

# Contents

Parallel Programming in Visual C++

Auto-Parallelization and Auto-Vectorization

Accelerated Multiprocessing (AMP)

C++ AMP (C++ Accelerated Massive Parallelism)

C++ AMP Overview

Using Tiles

Using C++ AMP in UWP Apps

Walkthrough: Matrix Multiplication

Walkthrough: Debugging a C++ AMP Application

Using Lambdas, Function Objects, and Restricted Functions

Graphics (C++ AMP)

Using accelerator and accelerator\_view Objects

Reference

Reference (C++ AMP)

Concurrency Namespace (C++ AMP)

Concurrency namespace functions (AMP)

Concurrency namespace enums (AMP)

Concurrency namespace operators (AMP)

Concurrency namespace constants (AMP)

accelerator Class

accelerator\_view Class

accelerator\_view\_removed Class

array Class

array\_view Class

completion\_future Class

extent Class (C++ AMP)

index Class

invalid\_compute\_domain Class

out\_of\_memory Class

runtime\_exception Class

tile\_barrier Class

tiled\_extent Class

tiled\_index Class

uninitialized\_object Class

unsupported\_feature Class

Concurrency::direct3d Namespace

Concurrency::direct3d namespace functions (AMP)

adopt\_d3d\_access\_lock\_t Structure

scoped\_d3d\_access\_lock Class

Concurrency::fast\_math Namespace

Concurrency::fast\_math namespace functions

Concurrency::graphics Namespace

Concurrency::graphics::direct3d Namespace

Concurrency::graphics::direct3d namespace functions

Concurrency::graphics namespace functions

Concurrency::graphics namespace enums

double\_2 Class

double\_3 Class

double\_4 Class

float\_2 Class

float\_3 Class

float\_4 Class

int\_2 Class

int\_3 Class

int\_4 Class

norm Class

norm\_2 Class

norm\_3 Class

norm\_4 Class

sampler Class

short\_vector Structure

[short\\_vector\\_traits Structure](#)

[texture Class](#)

[texture\\_view Class](#)

[writable\\_texture\\_view Class](#)

[uint\\_2 Class](#)

[uint\\_3 Class](#)

[uint\\_4 Class](#)

[unorm Class](#)

[unorm\\_2 Class](#)

[unorm\\_3 Class](#)

[unorm\\_4 Class](#)

[Concurrency::precise\\_math Namespace](#)

[Concurrency::precise\\_math namespace functions](#)

[Concurrency Runtime \(ConCRT\)](#)

[Concurrency Runtime](#)

[Overview of the Concurrency Runtime](#)

[Exception Handling in the Concurrency Runtime](#)

[Parallel Diagnostic Tools \(Concurrency Runtime\)](#)

[Creating Asynchronous Operations in C++ for UWP Apps](#)

[Comparing the Concurrency Runtime to Other Concurrency Models](#)

[Migrating from OpenMP to the Concurrency Runtime](#)

[How to: Convert an OpenMP parallel for Loop to Use the Concurrency Runtime](#)

[How to: Convert an OpenMP Loop that Uses Cancellation to Use the Concurrency Runtime](#)

[How to: Convert an OpenMP Loop that Uses Exception Handling to Use the Concurrency Runtime](#)

[How to: Convert an OpenMP Loop that Uses a Reduction Variable to Use the Concurrency Runtime](#)

[Parallel Patterns Library \(PPL\)](#)

[Task Parallelism \(Concurrency Runtime\)](#)

[How to: Use parallel\\_invoke to Write a Parallel Sort Routine](#)

[How to: Use parallel\\_invoke to Execute Parallel Operations](#)

[How to: Create a Task that Completes After a Delay](#)

## Parallel Algorithms

[How to: Write a parallel\\_for Loop](#)

[How to: Write a parallel\\_for\\_each Loop](#)

[How to: Perform Map and Reduce Operations in Parallel](#)

## Parallel Containers and Objects

[How to: Use Parallel Containers to Increase Efficiency](#)

[How to: Use combinable to Improve Performance](#)

[How to: Use combinable to Combine Sets](#)

## Cancellation in the PPL

[How to: Use Cancellation to Break from a Parallel Loop](#)

[How to: Use Exception Handling to Break from a Parallel Loop](#)

## Asynchronous Agents Library

[Asynchronous Agents](#)

[Asynchronous Message Blocks](#)

[Message Passing Functions](#)

[How to: Implement Various Producer-Consumer Patterns](#)

[How to: Provide Work Functions to the call and transformer Classes](#)

[How to: Use transformer in a Data Pipeline](#)

[How to: Select Among Completed Tasks](#)

[How to: Send a Message at a Regular Interval](#)

[How to: Use a Message Block Filter](#)

## Synchronization Data Structures

[Comparing Synchronization Data Structures to the Windows API](#)

## Task Scheduler (Concurrency Runtime)

[Scheduler Instances](#)

[How to: Manage a Scheduler Instance](#)

[Scheduler Policies](#)

[How to: Specify Specific Scheduler Policies](#)

[How to: Create Agents that Use Specific Scheduler Policies](#)

[Schedule Groups](#)

[How to: Use Schedule Groups to Influence Order of Execution](#)

[Lightweight Tasks](#)

## Contexts

[How to: Use the Context Class to Implement a Cooperative Semaphore](#)

[How to: Use Oversubscription to Offset Latency](#)

## Memory Management Functions

[How to: Use Alloc and Free to Improve Memory Performance](#)

## Concurrency Runtime Walkthroughs

[Walkthrough: Connecting Using Tasks and XML HTTP Requests](#)

[Walkthrough: Creating an Agent-Based Application](#)

[Walkthrough: Creating a Dataflow Agent](#)

[Walkthrough: Creating an Image-Processing Network](#)

[Walkthrough: Implementing Futures](#)

[Walkthrough: Using join to Prevent Deadlock](#)

[Walkthrough: Removing Work from a User-Interface Thread](#)

[Walkthrough: Using the Concurrency Runtime in a COM-Enabled Application](#)

[Walkthrough: Adapting Existing Code to Use Lightweight Tasks](#)

[Walkthrough: Creating a Custom Message Block](#)

## Concurrency Runtime Best Practices

[Best Practices in the Parallel Patterns Library](#)

[Best Practices in the Asynchronous Agents Library](#)

[General Best Practices in the Concurrency Runtime](#)

## Reference

[Reference \(Concurrency Runtime\)](#)

[concurrency Namespace](#)

[concurrency namespace functions](#)

[concurrency namespace Operators](#)

[concurrency namespace constants1](#)

[concurrency namespace enums](#)

[affinity\\_partitioner Class](#)

[agent Class](#)

[auto\\_partitioner Class](#)

[bad\\_target Class](#)

[call Class](#)

cancellation\_token Class  
cancellation\_token\_registration Class  
cancellation\_token\_source Class  
choice Class  
combinable Class  
concurrent\_priority\_queue Class  
concurrent\_queue Class  
concurrent\_unordered\_map Class  
concurrent\_unordered\_multimap Class  
concurrent\_unordered\_multiset Class  
concurrent\_unordered\_set Class  
concurrent\_vector Class  
Context Class  
context\_self\_unblock Class  
context\_unblock\_unbalanced Class  
critical\_section Class  
CurrentScheduler Class  
default\_scheduler\_exists Class  
DispatchState Structure  
event Class  
IExecutionContext Structure  
IExecutionResource Structure  
improper\_lock Class  
improper\_scheduler\_attach Class  
improper\_scheduler\_detach Class  
improper\_scheduler\_reference Class  
invalid\_link\_target Class  
invalid\_multiple\_scheduling Class  
invalid\_operation Class  
invalid\_oversubscribe\_operation Class  
invalid\_scheduler\_policy\_key Class  
invalid\_scheduler\_policy\_thread\_specification Class

invalid\_scheduler\_policy\_value Class  
IResourceManager Structure  
IScheduler Structure  
ISchedulerProxy Structure  
ISource Class  
ITarget Class  
IThreadProxy Structure  
ITopologyExecutionResource Structure  
ITopologyNode Structure  
IUMSCompletionList Structure  
IUMSScheduler Structure  
IUMSThreadProxy Structure  
IUMSUnblockNotification Structure  
IVirtualProcessorRoot Structure  
join Class  
location Class  
message Class  
message\_not\_found Class  
message\_processor Class  
missing\_wait Class  
multi\_link\_registry Class  
multitype\_join Class  
nested\_scheduler\_missing\_detach Class  
network\_link\_registry Class  
operation\_timed\_out Class  
ordered\_message\_processor Class  
overwrite\_buffer Class  
progress\_reporter Class  
propagator\_block Class  
reader\_writer\_lock Class  
ScheduleGroup Class  
Scheduler Class

scheduler\_interface Structure  
scheduler\_not\_attached Class  
scheduler\_ptr Structure (Concurrency Runtime)  
scheduler\_resource\_allocation\_error Class  
scheduler\_worker\_creation\_error Class  
SchedulerPolicy Class  
simple\_partitioner Class  
single\_assignment Class  
single\_link\_registry Class  
source\_block Class  
source\_link\_manager Class  
static\_partitioner Class  
structured\_task\_group Class  
target\_block Class  
task Class (Concurrency Runtime)  
task\_canceled Class  
task\_completion\_event Class  
task\_continuation\_context Class  
task\_group Class  
task\_handle Class  
task\_options Class (Concurrency Runtime)  
timer Class  
transformer Class  
unbounded\_buffer Class  
unsupported\_os Class  
std Namespace  
    make\_exception\_ptr Function  
stdx Namespace  
    declval Function

## OpenMP

OpenMP in Visual C++

SIMD Extension

## OpenMP C and C++ Application Program Interface

Introduction

Directives

Run-time library functions

Environment variables

Appendices

Examples

Stubs for run-time library functions

OpenMP C and C++ grammar

The schedule clause

Implementation-defined behaviors in OpenMP C/C++

New features and clarifications in version 2.0

## OpenMP Library Reference

Directives

Clauses

Functions

Environment Variables

## Multithreading Support for Older Code (Visual C++)

### Multithreading with C and Win32

Multithread Programs

Library Support for Multithreading

Include Files for Multithreading

C Run-Time Library Functions for Thread Control

Sample Multithread C Program

Writing a Multithreaded Win32 Program

Compiling and Linking Multithread Programs

Avoiding Problem Areas with Multithread Programs

Thread Local Storage (TLS)

### Multithreading with C++ and MFC

Multithreading: Creating User-Interface Threads

Multithreading: Creating Worker Threads

Multithreading: When to Use the Synchronization Classes

[Multithreading: How to Use the Synchronization Classes](#)

[Multithreading: Terminating Threads](#)

[Multithreading: Programming Tips](#)

[Multithreading and Locales](#)

# Parallel Programming in Visual C++

5/15/2019 • 2 minutes to read • [Edit Online](#)

Visual C++ provides the following technologies to help you create multi-threaded and parallel programs that take advantage of multiple cores and use the GPU for general purpose programming.

## Related Articles

TITLE	DESCRIPTION
<a href="#">Auto-Parallelization and Auto-Vectorization</a>	Compiler optimizations that speed up code.
<a href="#">Concurrency Runtime</a>	Classes that simplify the writing of programs that use data parallelism or task parallelism.
<a href="#">C++ AMP (C++ Accelerated Massive Parallelism)</a>	Classes that enable the use of modern graphics processors for general purpose programming.
<a href="#">Multithreading Support for Older Code (Visual C++)</a>	Older technologies that may be useful in older applications. For new apps, use the Concurrency Runtime or C++ AMP.
<a href="#">OpenMP</a>	The Microsoft implementation of the OpenMP API.
<a href="#">C++ in Visual Studio</a>	This section of the documentation contains information about most of the features of Visual C++.

# Auto-Parallelization and Auto-Vectorization

3/4/2019 • 3 minutes to read • [Edit Online](#)

Auto-Parallelizer and Auto-Vectorizer are designed to provide automatic performance gains for loops in your code.

## Auto-Parallelizer

The `/Qpar` compiler switch enables *automatic parallelization* of loops in your code. When you specify this flag without changing your existing code, the compiler evaluates the code to find loops that might benefit from parallelization. Because it might find loops that don't do much work and therefore won't benefit from parallelization, and because every unnecessary parallelization can engender the spawning of a thread pool, extra synchronization, or other processing that would tend to slow performance instead of improving it, the compiler is conservative in selecting the loops that it parallelizes. For example, consider the following example in which the upper bound of the loop is not known at compile time:

```
void loop_test(int u) {
    for (int i=0; i<u; ++i)
        A[i] = B[i] * C[i];
}
```

Because `u` could be a small value, the compiler won't automatically parallelize this loop. However, you might still want it parallelized because you know that `u` will always be large. To enable the auto-parallelization, specify `#pragma loop(hint_parallel(n))`, where `n` is the number of threads to parallelize across. In the following example, the compiler will attempt to parallelize the loop across 8 threads.

```
void loop_test(int u) {
    #pragma loop(hint_parallel(8))
    for (int i=0; i<u; ++i)
        A[i] = B[i] * C[i];
}
```

As with all [pragma directives](#), the alternate pragma syntax `__pragma(loop(hint_parallel(n)))` is also supported.

There are some loops that the compiler can't parallelize even if you want it to. Here's an example:

```
#pragma loop(hint_parallel(8))
for (int i=0; i<upper_bound(); ++i)
    A[i] = B[i] * C[i];
```

The function `upper_bound()` might change every time it's called. Because the upper bound cannot be known, the compiler can emit a diagnostic message that explains why it can't parallelize this loop. The following example demonstrates a loop that can be parallelized, a loop that cannot be parallelized, the compiler syntax to use at the command prompt, and the compiler output for each command line option:

```

int A[1000];
void test() {
#pragma loop(hint_parallel(0))
    for (int i=0; i<1000; ++i) {
        A[i] = A[i] + 1;
    }

    for (int i=1000; i<2000; ++i) {
        A[i] = A[i] + 1;
    }
}

```

Compiling by using this command:

```
cl d:\myproject\mylooptest.cpp /O2 /Qpar /Qpar-report:1
```

yields this output:

```

--- Analyzing function: void __cdecl test(void)
d:\myproject\mytest.cpp(4) : loop parallelized

```

Compiling by using this command:

```
cl d:\myproject\mylooptest.cpp /O2 /Qpar /Qpar-report:2
```

yields this output:

```

--- Analyzing function: void __cdecl test(void)
d:\myproject\mytest.cpp(4) : loop parallelized
d:\myproject\mytest.cpp(4) : loop not parallelized due to reason '1008'

```

Notice the difference in output between the two different [/Qpar-report \(Auto-Parallelizer Reporting Level\)](#) options.

`/Qpar-report:1` outputs parallelizer messages only for loops that are successfully parallelized. `/Qpar-report:2` outputs parallelizer messages for both successful and unsuccessful loop parallelizations.

For more information about reason codes and messages, see [Vectorizer and Parallelizer Messages](#).

## Auto-Vectorizer

The Auto-Vectorizer analyzes loops in your code, and uses the vector registers and instructions on the target computer to execute them, if it can. This can improve the performance of your code. The compiler targets the SSE2, AVX, and AVX2 instructions in Intel or AMD processors, or the NEON instructions on ARM processors, according to the `/arch` switch.

The Auto-Vectorizer may generate different instructions than specified by the `/arch` switch. These instructions are guarded by a runtime check to make sure that code still runs correctly. For example, when you compile `/arch:SSE2`, SSE4.2 instructions may be emitted. A runtime check verifies that SSE4.2 is available on the target processor and jumps to a non-SSE4.2 version of the loop if the processor does not support those instructions.

By default, the Auto-Vectorizer is enabled. If you want to compare the performance of your code under vectorization, you can use `#pragma loop(no_vector)` to disable vectorization of any given loop.

```

#pragma loop(no_vector)
for (int i = 0; i < 1000; ++i)
    A[i] = B[i] + C[i];

```

As with all [pragma directives](#), the alternate pragma syntax `__pragma(loop(no_vector))` is also supported.

As with the Auto-Parallelizer, you can specify the [/Qvec-report \(Auto-Vectorizer Reporting Level\)](#) command-line option to report either successfully vectorized loops only—`/Qvec-report:1`—or both successfully and unsuccessfully vectorized loops—`/Qvec-report:2`).

For more information about reason codes and messages, see [Vectorizer and Parallelizer Messages](#).

For an example showing how the vectorizer works in practice, see [Project Austin Part 2 of 6: Page Curling](#)

## See also

[loop](#)

[Parallel Programming in Native Code](#)

[/Qpar \(Auto-Parallelizer\)](#)

[/Qpar-report \(Auto-Parallelizer Reporting Level\)](#)

[/Qvec-report \(Auto-Vectorizer Reporting Level\)](#)

[Vectorizer and Parallelizer Messages](#)

# C++ AMP (C++ Accelerated Massive Parallelism)

10/31/2018 • 2 minutes to read • [Edit Online](#)

C++ AMP (C++ Accelerated Massive Parallelism) accelerates the execution of your C++ code by taking advantage of the data-parallel hardware that's commonly present as a graphics processing unit (GPU) on a discrete graphics card. The C++ AMP programming model includes support for multidimensional arrays, indexing, memory transfer, and tiling. It also includes a mathematical function library. You can use C++ AMP language extensions to control how data is moved from the CPU to the GPU and back.

## Related Topics

TITLE	DESCRIPTION
<a href="#">C++ AMP Overview</a>	Describes the key features of C++ AMP and the mathematical library.
<a href="#">Using Lambdas, Function Objects, and Restricted Functions</a>	Describes how to use lambda expressions, function objects, and restricted functions in calls to the <a href="#">parallel_for_each</a> method.
<a href="#">Using Tiles</a>	Describes how to use tiles to accelerate your C++ AMP code.
<a href="#">Using accelerator and accelerator_view Objects</a>	Describes how to use accelerators to customize execution of your code on the GPU.
<a href="#">Using C++ AMP in UWP Apps</a>	Describes how to use C++ AMP in Universal Windows Platform (UWP) apps that use Windows Runtime types.
<a href="#">Graphics (C++ AMP)</a>	Describes how to use the C++ AMP graphics library.
<a href="#">Walkthrough: Matrix Multiplication</a>	Demonstrates matrix multiplication using C++ AMP code and tiling.
<a href="#">Walkthrough: Debugging a C++ AMP Application</a>	Explains how to create and debug an application that uses parallel reduction to sum up a large array of integers.

## Reference

[Reference \(C++ AMP\)](#)

[tile\\_static](#) Keyword

[restrict \(C++ AMP\)](#)

## Other Resources

[Parallel Programming in Native Code Blog](#)

[C++ AMP sample projects for download](#)

[Analyzing C++ AMP Code with the Concurrency Visualizer](#)

# C++ AMP Overview

3/22/2019 • 18 minutes to read • [Edit Online](#)

C++ Accelerated Massive Parallelism (C++ AMP) accelerates execution of C++ code by taking advantage of data-parallel hardware such as a graphics processing unit (GPU) on a discrete graphics card. By using C++ AMP, you can code multi-dimensional data algorithms so that execution can be accelerated by using parallelism on heterogeneous hardware. The C++ AMP programming model includes multidimensional arrays, indexing, memory transfer, tiling, and a mathematical function library. You can use C++ AMP language extensions to control how data is moved from the CPU to the GPU and back, so that you can improve performance.

## System Requirements

- Windows 7 or later
- Windows Server 2008 R2 or later
- DirectX 11 Feature Level 11.0 or later hardware
- For debugging on the software emulator, Windows 8 or Windows Server 2012 is required. For debugging on the hardware, you must install the drivers for your graphics card. For more information, see [Debugging GPU Code](#).
- Note: AMP is currently not supported on ARM64.

## Introduction

The following two examples illustrate the primary components of C++ AMP. Assume that you want to add the corresponding elements of two one-dimensional arrays. For example, you might want to add `{1, 2, 3, 4, 5}` and `{6, 7, 8, 9, 10}` to obtain `{7, 9, 11, 13, 15}`. Without using C++ AMP, you might write the following code to add the numbers and display the results.

```
#include <iostream>

void StandardMethod() {

    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[5];

    for (int idx = 0; idx < 5; idx++)
    {
        sumCPP[idx] = aCPP[idx] + bCPP[idx];
    }

    for (int idx = 0; idx < 5; idx++)
    {
        std::cout << sumCPP[idx] << "\n";
    }
}
```

The important parts of the code are as follows:

- Data: The data consists of three arrays. All have the same rank (one) and length (five).
- Iteration: The first `for` loop provides a mechanism for iterating through the elements in the arrays. The

code that you want to execute to compute the sums is contained in the first `for` block.

- Index: The `idx` variable accesses the individual elements of the arrays.

Using C++ AMP, you might write the following code instead.

```
#include <amp.h>
#include <iostream>
using namespace concurrency;

const int size = 5;

void CppAmpMethod() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[size];

    // Create C++ AMP objects.
    array_view<const int, 1> a(size, aCPP);
    array_view<const int, 1> b(size, bCPP);
    array_view<int, 1> sum(size, sumCPP);
    sum.discard_data();

    parallel_for_each(
        // Define the compute domain, which is the set of threads that are created.
        sum.extent,
        // Define the code to run on each thread on the accelerator.
        [=](index<1> idx) restrict(amp) {
            sum[idx] = a[idx] + b[idx];
        }
    );

    // Print the results. The expected output is "7, 9, 11, 13, 15".
    for (int i = 0; i < size; i++) {
        std::cout << sum[i] << "\n";
    }
}
```

The same basic elements are present, but C++ AMP constructs are used:

- Data: You use C++ arrays to construct three C++ AMP `array_view` objects. You supply four values to construct an `array_view` object: the data values, the rank, the element type, and the length of the `array_view` object in each dimension. The rank and type are passed as type parameters. The data and length are passed as constructor parameters. In this example, the C++ array that is passed to the constructor is one-dimensional. The rank and length are used to construct the rectangular shape of the data in the `array_view` object, and the data values are used to fill the array. The runtime library also includes the `array Class`, which has an interface that resembles the `array_view` class and is discussed later in this article.
- Iteration: The `parallel_for_each Function (C++ AMP)` provides a mechanism for iterating through the data elements, or *compute domain*. In this example, the compute domain is specified by `sum.extent`. The code that you want to execute is contained in a lambda expression, or *kernel function*. The `restrict(amp)` indicates that only the subset of the C++ language that C++ AMP can accelerate is used.
- Index: The `index Class` variable, `idx`, is declared with a rank of one to match the rank of the `array_view` object. By using the index, you can access the individual elements of the `array_view` objects.

## Shaping and Indexing Data: index and extent

You must define the data values and declare the shape of the data before you can run the kernel code. All data is defined to be an array (rectangular), and you can define the array to have any rank (number of dimensions). The data can be any size in any of the dimensions.

## index Class

The [index Class](#) specifies a location in the `array` or `array_view` object by encapsulating the offset from the origin in each dimension into one object. When you access a location in the array, you pass an `index` object to the indexing operator, `[]`, instead of a list of integer indexes. You can access the elements in each dimension by using the `array::operator()` [Operator](#) or the `array_view::operator()` [Operator](#).

The following example creates a one-dimensional index that specifies the third element in a one-dimensional `array_view` object. The index is used to print the third element in the `array_view` object. The output is 3.

```
int aCPP[] = {1, 2, 3, 4, 5};
array_view<int, 1> a(5, aCPP);

index<1> idx(2);

std::cout << a[idx] << "\n";
// Output: 3
```

The following example creates a two-dimensional index that specifies the element where the row = 1 and the column = 2 in a two-dimensional `array_view` object. The first parameter in the `index` constructor is the row component, and the second parameter is the column component. The output is 6.

```
int aCPP[] = {1, 2, 3, 4, 5, 6};
array_view<int, 2> a(2, 3, aCPP);

index<2> idx(1, 2);

std::cout << a[idx] << "\n";
// Output: 6
```

The following example creates a three-dimensional index that specifies the element where the depth = 0, the row = 1, and the column = 3 in a three-dimensional `array_view` object. Notice that the first parameter is the depth component, the second parameter is the row component, and the third parameter is the column component. The output is 8.

```
int aCPP[] = {
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

array_view<int, 3> a(2, 3, 4, aCPP);

// Specifies the element at 3, 1, 0.
index<3> idx(0, 1, 3);

std::cout << a[idx] << "\n";
// Output: 8
```

## extent Class

The [extent Class](#) specifies the length of the data in each dimension of the `array` or `array_view` object. You can create an extent and use it to create an `array` or `array_view` object. You can also retrieve the extent of an existing `array` or `array_view` object. The following example prints the length of the extent in each dimension of an `array_view` object.

```

int aCPP[] = {
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
// There are 3 rows and 4 columns, and the depth is two.
array_view<int, 3> a(2, 3, 4, aCPP);

std::cout << "The number of columns is " << a.extent[2] << "\n";
std::cout << "The number of rows is " << a.extent[1] << "\n";
std::cout << "The depth is " << a.extent[0] << "\n";
std::cout << "Length in most significant dimension is " << a.extent[0] << "\n";

```

The following example creates an `array_view` object that has the same dimensions as the object in the previous example, but this example uses an `extent` object instead of using explicit parameters in the `array_view` constructor.

```

int aCPP[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24};
extent<3> e(2, 3, 4);

array_view<int, 3> a(e, aCPP);

std::cout << "The number of columns is " << a.extent[2] << "\n";
std::cout << "The number of rows is " << a.extent[1] << "\n";
std::cout << "The depth is " << a.extent[0] << "\n";

```

## Moving Data to the Accelerator: `array` and `array_view`

Two data containers used to move data to the accelerator are defined in the runtime library. They are the `array Class` and the `array_view Class`. The `array` class is a container class that creates a deep copy of the data when the object is constructed. The `array_view` class is a wrapper class that copies the data when the kernel function accesses the data. When the data is needed on the source device the data is copied back.

### `array Class`

When an `array` object is constructed, a deep copy of the data is created on the accelerator if you use a constructor that includes a pointer to the data set. The kernel function modifies the copy on the accelerator. When the execution of the kernel function is finished, you must copy the data back to the source data structure. The following example multiplies each element in a vector by 10. After the kernel function is finished, the `vector conversion operator` is used to copy the data back into the vector object.

```

std::vector<int> data(5);

for (int count = 0; count <5; count++)
{
    data[count] = count;
}

array<int, 1> a(5, data.begin(), data.end());

parallel_for_each(
    a.extent,
    [=, &a](index<1> idx) restrict(amp) {
        a[idx] = a[idx]* 10;
    });

data = a;
for (int i = 0; i < 5; i++)
{
    std::cout << data[i] << "\n";
}

```

## array\_view Class

The `array_view` has nearly the same members as the `array` class, but the underlying behavior is not the same. Data passed to the `array_view` constructor is not replicated on the GPU as it is with an `array` constructor. Instead, the data is copied to the accelerator when the kernel function is executed. Therefore, if you create two `array_view` objects that use the same data, both `array_view` objects refer to the same memory space. When you do this, you have to synchronize any multithreaded access. The main advantage of using the `array_view` class is that data is moved only if it is necessary.

## Comparison of array and array\_view

The following table summarizes the similarities and differences between the `array` and `array_view` classes.

DESCRIPTION	ARRAY CLASS	ARRAY_VIEW CLASS
When rank is determined	At compile time.	At compile time.
When extent is determined	At run time.	At run time.
Shape	Rectangular.	Rectangular.
Data storage	Is a data container.	Is a data wrapper.
Copy	Explicit and deep copy at definition.	Implicit copy when it is accessed by the kernel function.
Data retrieval	By copying the array data back to an object on the CPU thread.	By direct access of the <code>array_view</code> object or by calling the <a href="#">array_view::synchronize Method</a> to continue accessing the data on the original container.

## Shared memory with array and array\_view

Shared memory is memory that can be accessed by both the CPU and the accelerator. The use of shared memory eliminates or significantly reduces the overhead of copying data between the CPU and the accelerator. Although the memory is shared, it cannot be accessed concurrently by both the CPU and the accelerator, and doing so causes undefined behavior.

`array` objects can be used to specify fine-grained control over the use of shared memory if the associated accelerator supports it. Whether an accelerator supports shared memory is determined by the accelerator's [supports\\_cpu\\_shared\\_memory](#) property, which returns **true** when shared memory is supported. If shared memory is supported, the default [access\\_type Enumeration](#) for memory allocations on the accelerator is determined by the `default_cpu_access_type` property. By default, `array` and `array_view` objects take on the same `access_type` as the primary associated `accelerator`.

By setting the [array::cpu\\_access\\_type Data Member](#) property of an `array` explicitly, you can exercise fine-grained control over how shared memory is used, so that you can optimize the app for the hardware's performance characteristics, based on the memory access patterns of its computation kernels. An `array_view` reflects the same `cpu_access_type` as the `array` that it's associated with; or, if the `array_view` is constructed without a data source, its `access_type` reflects the environment that first causes it to allocate storage. That is, if it's first accessed by the host (CPU), then it behaves as if it were created over a CPU data source and shares the `access_type` of the `accelerator_view` associated by capture; however, if it's first accessed by an `accelerator_view`, then it behaves as if it were created over an `array` created on that `accelerator_view` and shares the `array`'s `access_type`.

The following code example shows how to determine whether the default accelerator supports shared memory, and then creates several arrays that have different `cpu_access_type` configurations.

```

#include <amp.h>
#include <iostream>

using namespace Concurrency;

int main()
{
    accelerator acc = accelerator(accelerator::default_accelerator);

    // Early out if the default accelerator doesn't support shared memory.
    if (!acc.supports_cpu_shared_memory)
    {
        std::cout << "The default accelerator does not support shared memory" << std::endl;
        return 1;
    }

    // Override the default CPU access type.
    acc.default_cpu_access_type = access_type_read_write

    // Create an accelerator_view from the default accelerator. The
    // accelerator_view inherits its default_cpu_access_type from acc.
    accelerator_view acc_v = acc.default_view;

    // Create an extent object to size the arrays.
    extent<1> ex(10);

    // Input array that can be written on the CPU.
    array<int, 1> arr_w(ex, acc_v, access_type_write);

    // Output array that can be read on the CPU.
    array<int, 1> arr_r(ex, acc_v, access_type_read);

    // Read-write array that can be both written to and read from on the CPU.
    array<int, 1> arr_rw(ex, acc_v, access_type_read_write);
}

```

## Executing Code over Data: `parallel_for_each`

The `parallel_for_each` function defines the code that you want to run on the accelerator against the data in the `array` or `array_view` object. Consider the following code from the introduction of this topic.

```

#include <amp.h>
#include <iostream>
using namespace concurrency;

void AddArrays() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[5] = {0, 0, 0, 0, 0};

    array_view<int, 1> a(5, aCPP);
    array_view<int, 1> b(5, bCPP);
    array_view<int, 1> sum(5, sumCPP);

    parallel_for_each(
        sum.extent,
        [=](index<1> idx) restrict(amp)
        {
            sum[idx] = a[idx] + b[idx];
        }
    );

    for (int i = 0; i < 5; i++) {
        std::cout << sum[i] << "\n";
    }
}

```

The `parallel_for_each` method takes two arguments, a compute domain and a lambda expression.

The *compute domain* is an `extent` object or a `tiled_extent` object that defines the set of threads to create for parallel execution. One thread is generated for each element in the compute domain. In this case, the `extent` object is one-dimensional and has five elements. Therefore, five threads are started.

The *lambda expression* defines the code to run on each thread. The capture clause, `[=]`, specifies that the body of the lambda expression accesses all captured variables by value, which in this case are `a`, `b`, and `sum`. In this example, the parameter list creates a one-dimensional `index` variable named `idx`. The value of the `idx[0]` is 0 in the first thread and increases by one in each subsequent thread. The `restrict(amp)` indicates that only the subset of the C++ language that C++ AMP can accelerate is used. The limitations on functions that have the `restrict` modifier are described in [restrict \(C++ AMP\)](#). For more information, see, [Lambda Expression Syntax](#).

The lambda expression can include the code to execute or it can call a separate kernel function. The kernel function must include the `restrict(amp)` modifier. The following example is equivalent to the previous example, but it calls a separate kernel function.

```

#include <amp.h>
#include <iostream>
using namespace concurrency;

void AddElements(
    index<1> idx,
    array_view<int, 1> sum,
    array_view<int, 1> a,
    array_view<int, 1> b) restrict(amp) {
    sum[idx] = a[idx] + b[idx];
}

void AddArraysWithFunction() {

    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[5] = {0, 0, 0, 0, 0};

    array_view<int, 1> a(5, aCPP);
    array_view<int, 1> b(5, bCPP);
    array_view<int, 1> sum(5, sumCPP);

    parallel_for_each(
        sum.extent,
        [=](index<1> idx) restrict(amp) {
            AddElements(idx, sum, a, b);
        }
    );

    for (int i = 0; i < 5; i++) {
        std::cout << sum[i] << "\n";
    }
}

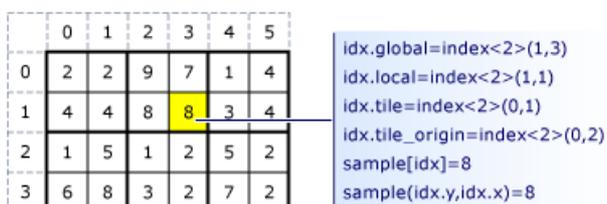
```

## Accelerating Code: Tiles and Barriers

You can gain additional acceleration by using tiling. Tiling divides the threads into equal rectangular subsets or *tiles*. You determine the appropriate tile size based on your data set and the algorithm that you are coding. For each thread, you have access to the *global* location of a data element relative to the whole `array` or `array_view` and access to the *local* location relative to the tile. Using the local index value simplifies your code because you don't have to write the code to translate index values from global to local. To use tiling, call the `extent::tile Method` on the compute domain in the `parallel_for_each` method, and use a `tiled_index` object in the lambda expression.

In typical applications, the elements in a tile are related in some way, and the code has to access and keep track of values across the tile. Use the `tile_static Keyword` keyword and the `tile_barrier::wait Method` to accomplish this. A variable that has the `tile_static` keyword has a scope across an entire tile, and an instance of the variable is created for each tile. You must handle synchronization of tile-thread access to the variable. The `tile_barrier::wait Method` stops execution of the current thread until all the threads in the tile have reached the call to `tile_barrier::wait`. So you can accumulate values across the tile by using `tile_static` variables. Then you can finish any computations that require access to all the values.

The following diagram represents a two-dimensional array of sampling data that is arranged in tiles.



The following code example uses the sampling data from the previous diagram. The code replaces each value in

the tile by the average of the values in the tile.

```
// Sample data:
int sampledata[] = {
    2, 2, 9, 7, 1, 4,
    4, 4, 8, 8, 3, 4,
    1, 5, 1, 2, 5, 2,
    6, 8, 3, 2, 7, 2};

// The tiles:
// 2 2   9 7   1 4
// 4 4   8 8   3 4
//
// 1 5   1 2   5 2
// 6 8   3 2   7 2

// Averages:
int averagedata[] = {
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
};

array_view<int, 2> sample(4, 6, sampledata);

array_view<int, 2> average(4, 6, averagedata);

parallel_for_each(
    // Create threads for sample.extent and divide the extent into 2 x 2 tiles.
    sample.extent.tile<2,2>(),
    [=](tiled_index<2,2> idx) restrict(amp) {
        // Create a 2 x 2 array to hold the values in this tile.
        tile_static int nums[2][2];

        // Copy the values for the tile into the 2 x 2 array.
        nums[idx.local[1]][idx.local[0]] = sample[idx.global];

        // When all the threads have executed and the 2 x 2 array is complete, find the average.
        idx.barrier.wait();
        int sum = nums[0][0] + nums[0][1] + nums[1][0] + nums[1][1];

        // Copy the average into the array_view.
        average[idx.global] = sum / 4;
    });

for (int i = 0; i <4; i++) {
    for (int j = 0; j <6; j++) {
        std::cout << average(i,j) << " ";
    }
    std::cout << "\n";
}

// Output:
// 3 3 8 8 3 3
// 3 3 8 8 3 3
// 5 5 2 2 4 4
// 5 5 2 2 4 4
```

## Math Libraries

C++ AMP includes two math libraries. The double-precision library in the [Concurrency::precise\\_math Namespace](#) provides support for double-precision functions. It also provides support for single-precision functions, although double-precision support on the hardware is still required. It complies with the [C99 Specification \(ISO/IEC 9899\)](#).

The accelerator must support full double precision. You can determine whether it does by checking the value of the [accelerator::supports\\_double\\_precision Data Member](#). The fast math library, in the [Concurrency::fast\\_math Namespace](#), contains another set of math functions. These functions, which support only `float` operands, execute more quickly but aren't as precise as those in the double-precision math library. The functions are contained in the `<amp_math.h>` header file and all are declared with `restrict(amp)`. The functions in the `<cmath>` header file are imported into both the `fast_math` and `precise_math` namespaces. The **restrict** keyword is used to distinguish the `<cmath>` version and the C++ AMP version. The following code calculates the base-10 logarithm, using the fast method, of each value that is in the compute domain.

```
#include <amp.h>
#include <amp_math.h>
#include <iostream>
using namespace concurrency;

void MathExample() {

    double numbers[] = { 1.0, 10.0, 60.0, 100.0, 600.0, 1000.0 };
    array_view<double, 1> logs(6, numbers);

    parallel_for_each(
        logs.extent,
        [=] (index<1> idx) restrict(amp) {
            logs[idx] = concurrency::fast_math::log10(numbers[idx]);
        }
    );

    for (int i = 0; i < 6; i++) {
        std::cout << logs[i] << "\n";
    }
}
```

## Graphics Library

C++ AMP includes a graphics library that is designed for accelerated graphics programming. This library is used only on devices that support native graphics functionality. The methods are in the [Concurrency::graphics Namespace](#) and are contained in the `<amp_graphics.h>` header file. The key components of the graphics library are:

- [texture Class](#): You can use the texture class to create textures from memory or from a file. Textures resemble arrays because they contain data, and they resemble containers in the C++ Standard Library with respect to assignment and copy construction. For more information, see [C++ Standard Library Containers](#). The template parameters for the `texture` class are the element type and the rank. The rank can be 1, 2, or 3. The element type can be one of the short vector types that are described later in this article.
- [writeonly\\_texture\\_view Class](#): Provides write-only access to any texture.
- [Short Vector Library](#): Defines a set of short vector types of length 2, 3, and 4 that are based on `int`, `uint`, `float`, `double`, `norm`, or `unorm`.

## Universal Windows Platform (UWP) Apps

Like other C++ libraries, you can use C++ AMP in your UWP apps. These articles describe how to include C++ AMP code in apps that is created by using C++, C#, Visual Basic, or JavaScript:

- [Using C++ AMP in UWP Apps](#)
- [Walkthrough: Creating a basic Windows Runtime component in C++ and calling it from JavaScript](#)
- [Bing Maps Trip Optimizer, a Window Store app in JavaScript and C++](#)

- [How to use C++ AMP from C# using the Windows Runtime](#)
- [How to use C++ AMP from C#](#)
- [Calling Native Functions from Managed Code](#)

## C++ AMP and Concurrency Visualizer

The Concurrency Visualizer includes support for analyzing performance of C++ AMP code. These articles describe these features:

- [GPU Activity Graph](#)
- [GPU Activity \(Paging\)](#)
- [GPU Activity \(This Process\)](#)
- [GPU Activity \(Other Processes\)](#)
- [Channels \(Threads View\)](#)
- [Analyzing C++ AMP Code with the Concurrency Visualizer](#)

## Performance Recommendations

Modulus and division of unsigned integers have significantly better performance than modulus and division of signed integers. We recommend that you use unsigned integers when possible.

## See also

[C++ AMP \(C++ Accelerated Massive Parallelism\)](#)

[Lambda Expression Syntax](#)

[Reference \(C++ AMP\)](#)

[Parallel Programming in Native Code Blog](#)

# Using Tiles

11/20/2018 • 11 minutes to read • [Edit Online](#)

You can use tiling to maximize the acceleration of your app. Tiling divides threads into equal rectangular subsets or *tiles*. If you use an appropriate tile size and tiled algorithm, you can get even more acceleration from your C++ AMP code. The basic components of tiling are:

- `tile_static` variables. The primary benefit of tiling is the performance gain from `tile_static` access. Access to data in `tile_static` memory can be significantly faster than access to data in the global space ( `array` or `array_view` objects). An instance of a `tile_static` variable is created for each tile, and all threads in the tile have access to the variable. In a typical tiled algorithm, data is copied into `tile_static` memory once from global memory and then accessed many times from the `tile_static` memory.
- `tile_barrier::wait` Method. A call to `tile_barrier::wait` suspends execution of the current thread until all of the threads in the same tile reach the call to `tile_barrier::wait`. You cannot guarantee the order that the threads will run in, only that no threads in the tile will execute past the call to `tile_barrier::wait` until all of the threads have reached the call. This means that by using the `tile_barrier::wait` method, you can perform tasks on a tile-by-tile basis rather than a thread-by-thread basis. A typical tiling algorithm has code to initialize the `tile_static` memory for the whole tile followed by a call to `tile_barrier::wait`. Code that follows `tile_barrier::wait` contains computations that require access to all the `tile_static` values.
- Local and global indexing. You have access to the index of the thread relative to the entire `array_view` or `array` object and the index relative to the tile. Using the local index can make your code easier to read and debug. Typically, you use local indexing to access `tile_static` variables, and global indexing to access `array` and `array_view` variables.
- `tiled_extent` Class and `tiled_index` Class. You use a `tiled_extent` object instead of an `extent` object in the `parallel_for_each` call. You use a `tiled_index` object instead of an `index` object in the `parallel_for_each` call.

To take advantage of tiling, your algorithm must partition the compute domain into tiles and then copy the tile data into `tile_static` variables for faster access.

## Example of Global, Tile, and Local Indices

The following diagram represents an 8x9 matrix of data that is arranged in 2x3 tiles.

	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15	16	17
2	18	19	20	21	22	23	24	25	26
3	27	28	29	30	31	32	33	34	35
4	36	37	38	39	40	41	42	43	44
5	45	46	47	48	49	50	51	52	53
6	54	55	56	57	58	59	60	61	62
7	63	64	65	66	67	68	69	70	71

```
descriptions[t_idx]=25
descriptions(t_idx.global[0], t_idx.global[1]) = 25
t_idx.global[0] = 2
t_idx.global[1] = 7
t_idx.tile[0] = 1
t_idx.tile[1] = 2
t_idx.local[0] = 0
t_idx.local[1] = 1
```

The following example displays the global, tile, and local indices of this tiled matrix. An `array_view` object is created by using elements of type `Description`. The `Description` holds the global, tile, and local indices of the element in the matrix. The code in the call to `parallel_for_each` sets the values of the global, tile, and local indices of each element. The output displays the values in the `Description` structures.

```
#include <iostream>
#include <iomanip>
#include <Windows.h>
#include <amp.h>
using namespace concurrency;

const int ROWS = 8;
const int COLS = 9;

// tileRow and tileColumn specify the tile that each thread is in.
// globalRow and globalColumn specify the location of the thread in the array_view.
// localRow and localColumn specify the location of the thread relative to the tile.
struct Description {
    int value;
    int tileRow;
    int tileColumn;
    int globalRow;
    int globalColumn;
    int localRow;
    int localColumn;
};

// A helper function for formatting the output.
void SetConsoleColor(int color) {
    int colorValue = (color == 0) ? 4 : 2;
    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), colorValue);
}

// A helper function for formatting the output.
void SetConsoleSize(int height, int width) {
    COORD coord;

    coord.X = width;
    coord.Y = height;
    SetConsoleScreenBufferSize(GetStdHandle(STD_OUTPUT_HANDLE), coord);

    SMALL_RECT* rect = new SMALL_RECT();
    rect->Left = 0;
    rect->Top = 0;
    rect->Right = width;
    rect->Bottom = height;
    SetConsoleWindowInfo(GetStdHandle(STD_OUTPUT_HANDLE), true, rect);
}

// This method creates an 8x9 matrix of Description structures.
// In the call to parallel_for_each, the structure is updated
// with tile, global, and local indices.
void TilingDescription() {
    // Create 72 (8x9) Description structures.
    std::vector<Description> desc;
    for (int i = 0; i < ROWS * COLS; i++) {
        Description d = {i, 0, 0, 0, 0, 0, 0};
        desc.push_back(d);
    }

    // Create an array_view from the Description structures.
    extent<2> matrix(ROWS, COLS);
    array_view<Description, 2> descriptions(matrix, desc);

    // Update each Description with the tile, global, and local indices.
    parallel_for_each(descriptions.extent.tile< 2, 3>(),
        [=] (tiled index< 2, 3> + idx) restrict(amp)

```

```

    [T] ( tiled_index< 2, 3> t_idx) restrict(amp)
    {
        descriptions[t_idx].globalRow = t_idx.global[0];
        descriptions[t_idx].globalColumn = t_idx.global[1];
        descriptions[t_idx].tileRow = t_idx.tile[0];
        descriptions[t_idx].tileColumn = t_idx.tile[1];
        descriptions[t_idx].localRow = t_idx.local[0];
        descriptions[t_idx].localColumn= t_idx.local[1];
    });

// Print out the Description structure for each element in the matrix.
// Tiles are displayed in red and green to distinguish them from each other.
SetConsoleSize(100, 150);
for (int row = 0; row < ROWS; row++) {
    for (int column = 0; column < COLS; column++) {
        SetConsoleColor((descriptions(row, column).tileRow + descriptions(row, column).tileColumn) % 2);
        std::cout << "Value: " << std::setw(2) << descriptions(row, column).value << " ";
    }
    std::cout << "\n";

    for (int column = 0; column < COLS; column++) {
        SetConsoleColor((descriptions(row, column).tileRow + descriptions(row, column).tileColumn) % 2);
        std::cout << "Tile:  " << "(" << descriptions(row, column).tileRow << "," << descriptions(row,
column).tileColumn << ") ";
    }
    std::cout << "\n";

    for (int column = 0; column < COLS; column++) {
        SetConsoleColor((descriptions(row, column).tileRow + descriptions(row, column).tileColumn) % 2);
        std::cout << "Global: " << "(" << descriptions(row, column).globalRow << "," << descriptions(row,
column).globalColumn << ") ";
    }
    std::cout << "\n";

    for (int column = 0; column < COLS; column++) {
        SetConsoleColor((descriptions(row, column).tileRow + descriptions(row, column).tileColumn) % 2);
        std::cout << "Local:  " << "(" << descriptions(row, column).localRow << "," << descriptions(row,
column).localColumn << ") ";
    }
    std::cout << "\n";
    std::cout << "\n";
}
}

void main() {
    TilingDescription();
    char wait;
    std::cin >> wait;
}

```

The main work of the example is in the definition of the `array_view` object and the call to `parallel_for_each`.

1. The vector of `Description` structures is copied into an 8x9 `array_view` object.
2. The `parallel_for_each` method is called with a `tiled_extent` object as the compute domain. The `tiled_extent` object is created by calling the `extent::tile()` method of the `descriptions` variable. The type parameters of the call to `extent::tile()`, `<2,3>`, specify that 2x3 tiles are created. Thus, the 8x9 matrix is tiled into 12 tiles, four rows and three columns.
3. The `parallel_for_each` method is called by using a `tiled_index<2,3>` object (`t_idx`) as the index. The type parameters of the index (`t_idx`) must match the type parameters of the compute domain (`descriptions.extent.tile< 2, 3>()`).
4. When each thread is executed, the index `t_idx` returns information about which tile the thread is in (`tiled_index::tile` property) and the location of the thread within the tile (`tiled_index::local` property).

# Tile Synchronization—`tile_static` and `tile_barrier::wait`

The previous example illustrates the tile layout and indices, but is not in itself very useful. Tiling becomes useful when the tiles are integral to the algorithm and exploit `tile_static` variables. Because all threads in a tile have access to `tile_static` variables, calls to `tile_barrier::wait` are used to synchronize access to the `tile_static` variables. Although all of the threads in a tile have access to the `tile_static` variables, there is no guaranteed order of execution of threads in the tile. The following example shows how to use `tile_static` variables and the `tile_barrier::wait` method to calculate the average value of each tile. Here are the keys to understanding the example:

1. The `rawData` is stored in an 8x8 matrix.
2. The tile size is 2x2. This creates a 4x4 grid of tiles and the averages can be stored in a 4x4 matrix by using an `array` object. There are only a limited number of types that you can capture by reference in an AMP-restricted function. The `array` class is one of them.
3. The matrix size and sample size are defined by using `#define` statements, because the type parameters to `array`, `array_view`, `extent`, and `tiled_index` must be constant values. You can also use `const int static` declarations. As an additional benefit, it is trivial to change the sample size to calculate the average over 4x4 tiles.
4. A `tile_static` 2x2 array of float values is declared for each tile. Although the declaration is in the code path for every thread, only one array is created for each tile in the matrix.
5. There is a line of code to copy the values in each tile to the `tile_static` array. For each thread, after the value is copied to the array, execution on the thread stops due to the call to `tile_barrier::wait`.
6. When all of the threads in a tile have reached the barrier, the average can be calculated. Because the code executes for every thread, there is an `if` statement to only calculate the average on one thread. The average is stored in the `averages` variable. The barrier is essentially the construct that controls calculations by tile, much as you might use a `for` loop.
7. The data in the `averages` variable, because it is an `array` object, must be copied back to the host. This example uses the vector conversion operator.
8. In the complete example, you can change `SAMPLESIZE` to 4 and the code executes correctly without any other changes.

```
#include <iostream>
#include <amp.h>
using namespace concurrency;

#define SAMPLESIZE 2
#define MATRIXSIZE 8
void SamplingExample() {

    // Create data and array_view for the matrix.
    std::vector<float> rawData;
    for (int i = 0; i < MATRIXSIZE * MATRIXSIZE; i++) {
        rawData.push_back((float)i);
    }
    extent<2> dataExtent(MATRIXSIZE, MATRIXSIZE);
    array_view<float, 2> matrix(dataExtent, rawData);

    // Create the array for the averages.
    // There is one element in the output for each tile in the data.
    std::vector<float> outputData;
    int outputSize = MATRIXSIZE / SAMPLESIZE;
    for (int j = 0; j < outputSize * outputSize; j++) {
        outputData.push_back((float)0);
    }
}
```

```

}
extent<2> outputExtent(MATRIXSIZE / SAMPLESIZE, MATRIXSIZE / SAMPLESIZE);
array<float, 2> averages(outputExtent, outputData.begin(), outputData.end());

// Use tiles that are SAMPLESIZE x SAMPLESIZE.
// Find the average of the values in each tile.
// The only reference-type variable you can pass into the parallel_for_each call
// is a concurrency::array.
parallel_for_each(matrix.extent.tile<SAMPLESIZE, SAMPLESIZE>(),
    [=, &averages] (tiled_index<SAMPLESIZE, SAMPLESIZE> t_idx) restrict(amp)
    {
        // Copy the values of the tile into a tile-sized array.
        tile_static float tileValues[SAMPLESIZE][SAMPLESIZE];
        tileValues[t_idx.local[0]][t_idx.local[1]] = matrix[t_idx];

        // Wait for the tile-sized array to load before you calculate the average.
        t_idx.barrier.wait();

        // If you remove the if statement, then the calculation executes for every
        // thread in the tile, and makes the same assignment to averages each time.
        if (t_idx.local[0] == 0 && t_idx.local[1] == 0) {
            for (int trow = 0; trow < SAMPLESIZE; trow++) {
                for (int tcol = 0; tcol < SAMPLESIZE; tcol++) {
                    averages(t_idx.tile[0],t_idx.tile[1]) += tileValues[trow][tcol];
                }
            }
            averages(t_idx.tile[0],t_idx.tile[1]) /= (float) (SAMPLESIZE * SAMPLESIZE);
        }
    }
});

// Print out the results.
// You cannot access the values in averages directly. You must copy them
// back to a CPU variable.
outputData = averages;
for (int row = 0; row < outputSize; row++) {
    for (int col = 0; col < outputSize; col++) {
        std::cout << outputData[row*outputSize + col] << " ";
    }
    std::cout << "\n";
}

// Output for SAMPLESIZE = 2 is:
// 4.5 6.5 8.5 10.5
// 20.5 22.5 24.5 26.5
// 36.5 38.5 40.5 42.5
// 52.5 54.5 56.5 58.5

// Output for SAMPLESIZE = 4 is:
// 13.5 17.5
// 45.5 49.5
}

int main() {
    SamplingExample();
}

```

## Race Conditions

It might be tempting to create a `tile_static` variable named `total` and increment that variable for each thread, like this:

```
// Do not do this.
tile_static float total;
total += matrix[t_idx];
t_idx.barrier.wait();

averages(t_idx.tile[0],t_idx.tile[1]) /= (float) (SAMPLESIZE* SAMPLESIZE);
```

The first problem with this approach is that `tile_static` variables cannot have initializers. The second problem is that there is a race condition on the assignment to `total`, because all of the threads in the tile have access to the variable in no particular order. You could program an algorithm to only allow one thread to access the total at each barrier, as shown next. However, this solution is not extensible.

```
// Do not do this.
tile_static float total;
if (t_idx.local[0] == 0 && t_idx.local[1] == 0) {
    total = matrix[t_idx];
}
t_idx.barrier.wait();

if (t_idx.local[0] == 0 && t_idx.local[1] == 1) {
    total += matrix[t_idx];
}
t_idx.barrier.wait();

// etc.
```

## Memory Fences

There are two kinds of memory accesses that must be synchronized—global memory access and `tile_static` memory access. A `concurrency::array` object allocates only global memory. A `concurrency::array_view` can reference global memory, `tile_static` memory, or both, depending on how it was constructed. There are two kinds of memory that must be synchronized:

- global memory
- `tile_static`

A *memory fence* ensures that memory accesses are available to other threads in the thread tile, and that memory accesses are executed according to program order. To ensure this, compilers and processors do not reorder reads and writes across the fence. In C++ AMP, a memory fence is created by a call to one of these methods:

- [tile\\_barrier::wait Method](#): Creates a fence around both global and `tile_static` memory.
- [tile\\_barrier::wait\\_with\\_all\\_memory\\_fence Method](#): Creates a fence around both global and `tile_static` memory.
- [tile\\_barrier::wait\\_with\\_global\\_memory\\_fence Method](#): Creates a fence around only global memory.
- [tile\\_barrier::wait\\_with\\_tile\\_static\\_memory\\_fence Method](#): Creates a fence around only `tile_static` memory.

Calling the specific fence that you require can improve the performance of your app. The barrier type affects how the compiler and the hardware reorder statements. For example, if you use a global memory fence, it applies only to global memory accesses and therefore, the compiler and the hardware might reorder reads and writes to `tile_static` variables on the two sides of the fence.

In the next example, the barrier synchronizes the writes to `tileValues`, a `tile_static` variable. In this example, `tile_barrier::wait_with_tile_static_memory_fence` is called instead of `tile_barrier::wait`.

```

// Using a tile_static memory fence.
parallel_for_each(matrix.extent.tile<SAMPLESIZE, SAMPLESIZE>(),
    [=, &averages] (tiled_index<SAMPLESIZE, SAMPLESIZE> t_idx) restrict(amp)
    {
        // Copy the values of the tile into a tile-sized array.
        tile_static float tileValues[SAMPLESIZE][SAMPLESIZE];
        tileValues[t_idx.local[0]][t_idx.local[1]] = matrix[t_idx];

        // Wait for the tile-sized array to load before calculating the average.
        t_idx.barrier.wait_with_tile_static_memory_fence();

        // If you remove the if statement, then the calculation executes
        // for every thread in the tile, and makes the same assignment to
        // averages each time.
        if (t_idx.local[0] == 0 && t_idx.local[1] == 0) {
            for (int trow = 0; trow < SAMPLESIZE; trow++) {
                for (int tcol = 0; tcol < SAMPLESIZE; tcol++) {
                    averages(t_idx.tile[0], t_idx.tile[1]) += tileValues[trow][tcol];
                }
            }
            averages(t_idx.tile[0], t_idx.tile[1]) /= (float) (SAMPLESIZE * SAMPLESIZE);
        }
    });

```

## See also

[C++ AMP \(C++ Accelerated Massive Parallelism\)](#)

[tile\\_static](#) Keyword

# Using C++ AMP in UWP Apps

3/4/2019 • 3 minutes to read • [Edit Online](#)

You can use C++ AMP (C++ Accelerated Massive Parallelism) in your Universal Windows Platform (UWP) app to perform calculations on the GPU (Graphics Processing Unit) or other computational accelerators. However, C++ AMP doesn't provide APIs for working directly with Windows Runtime types, and the Windows Runtime doesn't provide a wrapper for C++ AMP. When you use Windows Runtime types in your code—including those that you've created yourself—you must convert them to types that are compatible with C++ AMP.

## Performance considerations

If you're using Visual C++ component extensions C++/CX to create your Universal Windows Platform (UWP) app, we recommend that you use plain-old-data (POD) types together with contiguous storage—for example, `std::vector` or C-style arrays—for data that will be used with C++ AMP. This can help you achieve higher performance than by using non-POD types or Windows RT containers because no marshaling has to occur.

In a C++ AMP kernel, to access data that's stored in this way, just wrap the `std::vector` or array storage in a `concurrency::array_view` and then use the array view in a `concurrency::parallel_for_each` loop:

```
// simple vector addition example
std::vector<int> data0(1024, 1);
std::vector<int> data1(1024, 2);
std::vector<int> data_out(data0.size(), 0);

concurrency::array_view<int, 1> av0(data0.size(), data0);
concurrency::array_view<int, 1> av1(data1.size(), data1);
concurrency::array_view<int, 1> av2(data_out.size(), data2);

av2.discard_data();

concurrency::parallel_for_each(av0.extent, [=](concurrency::index<1> idx) restrict(amp)
{
    av2[idx] = av0[idx] + av1[idx];
});
```

## Marshaling Windows Runtime types

When you work with Windows Runtime APIs, you might want to use C++ AMP on data that's stored in a Windows Runtime container such as a `Platform::Array<T>^` or in complex data types such as classes or structs that are declared by using the **ref** keyword or the **value** keyword. In these situations, you have to do some extra work to make the data available to C++ AMP.

### **Platform::Array<T>^, where T is a POD type**

When you encounter a `Platform::Array<T>^` and T is a POD type, you can access its underlying storage just by using the `get` member function:

```
Platform::Array<float>^ arr; // Assume that this was returned by a Windows Runtime API
concurrency::array_view<float, 1> av(arr->Length, &arr->get(0));
```

If T is not a POD type, use the technique that's described in the following section to use the data with C++ AMP.

### **Windows Runtime types: ref classes and value classes**

C++ AMP doesn't support complex data types. This includes non-POD types and any types that are declared by using the **ref** keyword or the **value** keyword. If an unsupported type is used in a `restrict(amp)` context, a compile-time error is generated.

When you encounter an unsupported type, you can copy interesting parts of its data into a `concurrency::array` object. In addition to making the data available for C++ AMP to consume, this manual-copy approach can also improve performance by maximizing data locality, and by ensuring that data that won't be used isn't copied to the accelerator. You can improve performance further by using a *staging array*, which is a special form of `concurrency::array` that provides a hint to the AMP runtime that the array should be optimized for frequent transfer between it and other arrays on the specified accelerator.

```
// pixel_color.h
ref class pixel_color sealed
{
public:
    pixel_color(Platform::String^ color_name, int red, int green, int blue)
    {
        name = color_name;
        r = red;
        g = green;
        b = blue;
    }

    property Platform::String^ name;
    property int r;
    property int g;
    property int b;
};

// Some other file

std::vector<pixel_color^> pixels (256);

for (pixel_color ^pixel : pixels)
{
    pixels.push_back(ref new pixel_color("blue", 0, 0, 255));
}

// Create the accelerators
auto cpuAccelerator = concurrency::accelerator(concurrency::accelerator::cpu_accelerator);
auto devAccelerator = concurrency::accelerator(concurrency::accelerator::default_accelerator);

// Create the staging arrays
concurrency::array<float, 1> red_vec(256, cpuAccelerator.default_view, devAccelerator.default_view);
concurrency::array<float, 1> blue_vec(256, cpuAccelerator.default_view, devAccelerator.default_view);

// Extract data from the complex array of structs into staging arrays.
concurrency::parallel_for(0, 256, [&](int i)
{
    red_vec[i] = pixels[i]->r;
    blue_vec[i] = pixels[i]->b;
});

// Array views are still used to copy data to the accelerator
concurrency::array_view<float, 1> av_red(red_vec);
concurrency::array_view<float, 1> av_blue(blue_vec);

// Change all pixels from blue to red.
concurrency::parallel_for_each(av_red.extent, [=](index<1> idx) restrict(amp)
{
    av_red[idx] = 255;
    av_blue[idx] = 0;
});
```

## See also

[Create your first UWP app using C++](#)

[Creating Windows Runtime Components in C++](#)

# Walkthrough: Matrix Multiplication

4/29/2019 • 10 minutes to read • [Edit Online](#)

This step-by-step walkthrough demonstrates how to use C++ AMP to accelerate the execution of matrix multiplication. Two algorithms are presented, one without tiling and one with tiling.

## Prerequisites

Before you start:

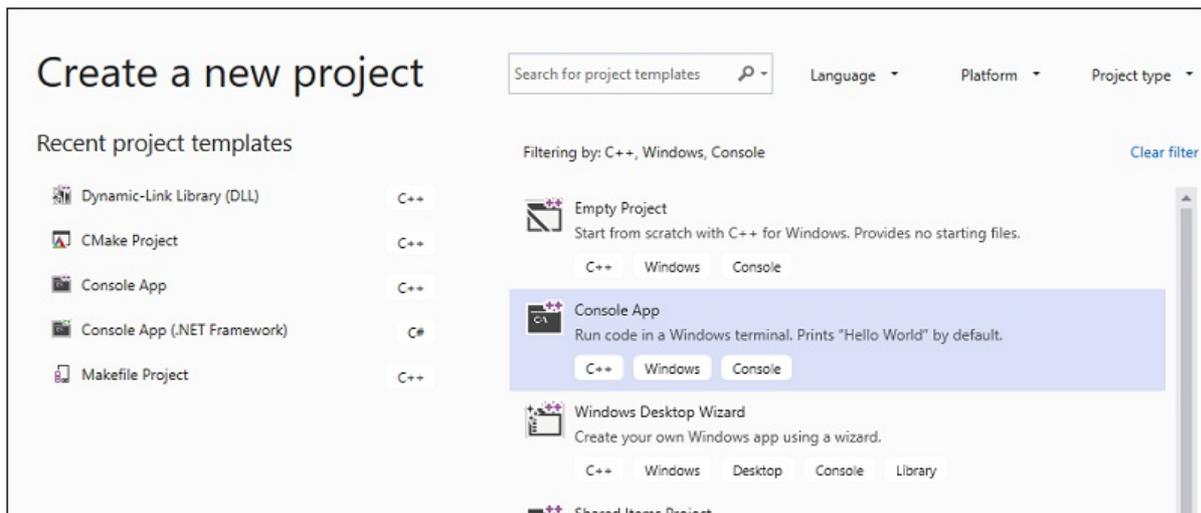
- Read [C++ AMP Overview](#).
- Read [Using Tiles](#).
- Make sure that you are running at least Windows 7, or Windows Server 2008 R2.

### To create the project

Instructions for creating a new project vary depending on which version of Visual Studio you have installed. Make sure you have the version selector in the upper left set to the correct version.

#### To create the project in Visual Studio 2019

1. On the menu bar, choose **File > New > Project** to open the **Create a New Project** dialog box.
2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Console**.
3. From the filtered list of project types, choose **Empty Project** then choose **Next**. In the next page, enter *MatrixMultiply* in the **Name** box to specify a name for the project, and specify the project location if desired.



4. Choose the **Create** button to create the client project.
5. In **Solution Explorer**, open the shortcut menu for **Source Files**, and then choose **Add > New Item**.
6. In the **Add New Item** dialog box, select **C++ File (.cpp)**, enter *MatrixMultiply.cpp* in the **Name** box, and then choose the **Add** button.

#### To create a project in Visual Studio 2017 or 2015

1. On the menu bar in Visual Studio, choose **File > New > Project**.
2. Under **Installed** in the templates pane, select **Visual C++**.

3. Select **Empty Project**, enter *MatrixMultiply* in the **Name** box, and then choose the **OK** button.
4. Choose the **Next** button.
5. In **Solution Explorer**, open the shortcut menu for **Source Files**, and then choose **Add > New Item**.
6. In the **Add New Item** dialog box, select **C++ File (.cpp)**, enter *MatrixMultiply.cpp* in the **Name** box, and then choose the **Add** button.

## Multiplication without tiling

In this section, consider the multiplication of two matrices, A and B, which are defined as follows:

$$A = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

$$B = \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

A is a 3-by-2 matrix and B is a 2-by-3 matrix. The product of multiplying A by B is the following 3-by-3 matrix. The product is calculated by multiplying the rows of A by the columns of B element by element.

$$\text{product} = \begin{bmatrix} 1 * 7 + 4 * 10 & 1 * 8 + 4 * 11 & 1 * 9 + 4 * 12 \\ 2 * 7 + 5 * 10 & 2 * 8 + 5 * 11 & 2 * 9 + 5 * 12 \\ 3 * 7 + 6 * 10 & 3 * 8 + 6 * 11 & 3 * 9 + 6 * 12 \end{bmatrix} = \begin{bmatrix} 47 & 52 & 57 \\ 64 & 71 & 78 \\ 81 & 90 & 99 \end{bmatrix}$$

### To multiply without using C++ AMP

1. Open *MatrixMultiply.cpp* and use the following code to replace the existing code.

```
#include <iostream>

void MultiplyWithoutAMP() {
    int aMatrix[3][2] = {{1, 4}, {2, 5}, {3, 6}};
    int bMatrix[2][3] = {{7, 8, 9}, {10, 11, 12}};
    int product[3][3] = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}};

    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 3; col++) {
            // Multiply the row of A by the column of B to get the row, column of product.
            for (int inner = 0; inner < 2; inner++) {
                product[row][col] += aMatrix[row][inner] * bMatrix[inner][col];
            }
            std::cout << product[row][col] << " ";
        }
        std::cout << "\n";
    }
}

void main() {
    MultiplyWithoutAMP();
    getchar();
}
```

The algorithm is a straightforward implementation of the definition of matrix multiplication. It does not use any parallel or threaded algorithms to reduce the computation time.

2. On the menu bar, choose **File > Save All**.
3. Choose the **F5** keyboard shortcut to start debugging and verify that the output is correct.

4. Choose **Enter** to exit the application.

### To multiply by using C++ AMP

1. In MatrixMultiply.cpp, add the following code before the `main` method.

```
void MultiplyWithAMP() {
    int aMatrix[] = { 1, 4, 2, 5, 3, 6 };
    int bMatrix[] = { 7, 8, 9, 10, 11, 12 };
    int productMatrix[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };

    array_view<int, 2> a(3, 2, aMatrix);

    array_view<int, 2> b(2, 3, bMatrix);

    array_view<int, 2> product(3, 3, productMatrix);

    parallel_for_each(product.extent,
        [=] (index<2> idx) restrict(amp) {
            int row = idx[0];
            int col = idx[1];
            for (int inner = 0; inner < 2; inner++) {
                product[idx] += a(row, inner)* b(inner, col);
            }
        });

    product.synchronize();

    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 3; col++) {
            //std::cout << productMatrix[row*3 + col] << " ";
            std::cout << product(row, col) << " ";
        }
        std::cout << "\n";
    }
}
```

The AMP code resembles the non-AMP code. The call to `parallel_for_each` starts one thread for each element in `product.extent`, and replaces the `for` loops for row and column. The value of the cell at the row and column is available in `idx`. You can access the elements of an `array_view` object by using either the `[]` operator and an index variable, or the `()` operator and the row and column variables. The example demonstrates both methods. The `array_view::synchronize` method copies the values of the `product` variable back to the `productMatrix` variable.

2. Add the following `include` and `using` statements at the top of MatrixMultiply.cpp.

```
#include <amp.h>
using namespace concurrency;
```

3. Modify the `main` method to call the `MultiplyWithAMP` method.

```
void main() {
    MultiplyWithOutAMP();
    MultiplyWithAMP();
    getchar();
}
```

4. Press the **Ctrl+F5** keyboard shortcut to start debugging and verify that the output is correct.

5. Press the **Spacebar** to exit the application.

# Multiplication with tiling

Tiling is a technique in which you partition data into equal-sized subsets, which are known as tiles. Three things change when you use tiling.

- You can create `tile_static` variables. Access to data in `tile_static` space can be many times faster than access to data in the global space. An instance of a `tile_static` variable is created for each tile, and all threads in the tile have access to the variable. The primary benefit of tiling is the performance gain due to `tile_static` access.
- You can call the `tile_barrier::wait` method to stop all of the threads in one tile at a specified line of code. You cannot guarantee the order that the threads will run in, only that all of the threads in one tile will stop at the call to `tile_barrier::wait` before they continue execution.
- You have access to the index of the thread relative to the entire `array_view` object and the index relative to the tile. By using the local index, you can make your code easier to read and debug.

To take advantage of tiling in matrix multiplication, the algorithm must partition the matrix into tiles and then copy the tile data into `tile_static` variables for faster access. In this example, the matrix is partitioned into submatrices of equal size. The product is found by multiplying the submatrices. The two matrices and their product in this example are:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

$$product = \begin{bmatrix} 34 & 44 & 54 & 64 \\ 82 & 108 & 134 & 160 \\ 34 & 44 & 54 & 64 \\ 82 & 108 & 134 & 160 \end{bmatrix}$$

The matrices are partitioned into four 2x2 matrices, which are defined as follows:

$$A = \begin{bmatrix} a = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} & b = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \\ c = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} & d = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \end{bmatrix}$$

$$B = \begin{bmatrix} e = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} & f = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \\ g = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} & h = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} \end{bmatrix}$$

The product of A and B can now be written and calculated as follows:

$$A * B = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

Because matrices `a` through `h` are 2x2 matrices, all of the products and sums of them are also 2x2 matrices. It also follows that the product of A and B is a 4x4 matrix, as expected. To quickly check the algorithm, calculate the value of the element in the first row, first column in the product. In the example, that would be the value of the element in the first row and first column of `ae + bg`. You only have to calculate the first column, first row of `ae`

and `bg` for each term. That value for `ae` is  $(1 * 1) + (2 * 5) = 11$ . The value for `bg` is  $(3 * 1) + (4 * 5) = 23$ . The final value is  $11 + 23 = 34$ , which is correct.

To implement this algorithm, the code:

- Uses a `tiled_extent` object instead of an `extent` object in the `parallel_for_each` call.
- Uses a `tiled_index` object instead of an `index` object in the `parallel_for_each` call.
- Creates `tile_static` variables to hold the submatrices.
- Uses the `tile_barrier::wait` method to stop the threads for the calculation of the products of the submatrices.

## To multiply by using AMP and tiling

1. In `MatrixMultiply.cpp`, add the following code before the `main` method.

```
void MultiplyWithTiling() {
    // The tile size is 2.
    static const int TS = 2;

    // The raw data.
    int aMatrix[] = { 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8 };
    int bMatrix[] = { 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8 };
    int productMatrix[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

    // Create the array_view objects.
    array_view<int, 2> a(4, 4, aMatrix);
    array_view<int, 2> b(4, 4, bMatrix);
    array_view<int, 2> product(4, 4, productMatrix);

    // Call parallel_for_each by using 2x2 tiles.
    parallel_for_each(product.extent.tile<TS, TS>(),
        [=] (tiled_index<TS, TS> t_idx) restrict(amp)
        {
            // Get the location of the thread relative to the tile (row, col)
            // and the entire array_view (rowGlobal, colGlobal).
            int row = t_idx.local[0];
            int col = t_idx.local[1];
            int rowGlobal = t_idx.global[0];
            int colGlobal = t_idx.global[1];
            int sum = 0;

            // Given a 4x4 matrix and a 2x2 tile size, this loop executes twice for each thread.
            // For the first tile and the first loop, it copies a into locA and e into locB.
            // For the first tile and the second loop, it copies b into locA and g into locB.
            for (int i = 0; i < 4; i += TS) {
                tile_static int locA[TS][TS];
                tile_static int locB[TS][TS];
                locA[row][col] = a(rowGlobal, col + i);
                locB[row][col] = b(row + i, colGlobal);
                // The threads in the tile all wait here until locA and locB are filled.
                t_idx.barrier.wait();

                // Return the product for the thread. The sum is retained across
                // both iterations of the loop, in effect adding the two products
                // together, for example, a*e.
                for (int k = 0; k < TS; k++) {
                    sum += locA[row][k] * locB[k][col];
                }

                // All threads must wait until the sums are calculated. If any threads
                // moved ahead, the values in locA and locB would change.
                t_idx.barrier.wait();
                // Now go on to the next iteration of the loop.
            }
        }
}
```

```

        // After both iterations of the loop, copy the sum to the product variable by using the
        global location.
        product[t_idx.global] = sum;
    });

    // Copy the contents of product back to the productMatrix variable.
    product.synchronize();

    for (int row = 0; row <4; row++) {
        for (int col = 0; col <4; col++) {
            // The results are available from both the product and productMatrix variables.
            //std::cout << productMatrix[row*3 + col] << " ";
            std::cout << product(row, col) << " ";
        }
        std::cout << "\n";
    }
}

```

This example is significantly different than the example without tiling. The code uses these conceptual steps:

- a. Copy the elements of tile[0,0] of `a` into `locA`. Copy the elements of tile[0,0] of `b` into `locB`. Notice that `product` is tiled, not `a` and `b`. Therefore, you use global indices to access `a`, `b`, and `product`. The call to `tile_barrier::wait` is essential. It stops all of the threads in the tile until both `locA` and `locB` are filled.
- b. Multiply `locA` and `locB` and put the results in `product`.
- c. Copy the elements of tile[0,1] of `a` into `locA`. Copy the elements of tile [1,0] of `b` into `locB`.
- d. Multiply `locA` and `locB` and add them to the results that are already in `product`.
- e. The multiplication of tile[0,0] is complete.
- f. Repeat for the other four tiles. There is no indexing specifically for the tiles and the threads can execute in any order. As each thread executes, the `tile_static` variables are created for each tile appropriately and the call to `tile_barrier::wait` controls the program flow.
- g. As you examine the algorithm closely, notice that each submatrix is loaded into a `tile_static` memory twice. That data transfer does take time. However, once the data is in `tile_static` memory, access to the data is much faster. Because calculating the products requires repeated access to the values in the submatrices, there is an overall performance gain. For each algorithm, experimentation is required to find the optimal algorithm and tile size.

In the non-AMP and non-tile examples, each element of A and B is accessed four times from the global memory to calculate the product. In the tile example, each element is accessed twice from the global memory and four times from the `tile_static` memory. That is not a significant performance gain. However, if the A and B were 1024x1024 matrices and the tile size were 16, there would be a significant performance gain. In that case, each element would be copied into `tile_static` memory only 16 times and accessed from `tile_static` memory 1024 times.

2. Modify the main method to call the `MultiplyWithTiling` method, as shown.

```

void main() {
    MultiplyWithoutAMP();
    MultiplyWithAMP();
    MultiplyWithTiling();
    getchar();
}

```

3. Press the **Ctrl+F5** keyboard shortcut to start debugging and verify that the output is correct.
4. Press the **Space** bar to exit the application.

## See also

[C++ AMP \(C++ Accelerated Massive Parallelism\)  
Walkthrough: Debugging a C++ AMP Application](#)

# Walkthrough: Debugging a C++ AMP Application

4/29/2019 • 13 minutes to read • [Edit Online](#)

This topic demonstrates how to debug an application that uses C++ Accelerated Massive Parallelism (C++ AMP) to take advantage of the graphics processing unit (GPU). It uses a parallel-reduction program that sums up a large array of integers. This walkthrough illustrates the following tasks:

- Launching the GPU debugger.
- Inspecting GPU threads in the GPU Threads window.
- Using the **Parallel Stacks** window to simultaneously observe the call stacks of multiple GPU threads.
- Using the **Parallel Watch** window to inspect values of a single expression across multiple threads at the same time.
- Flagging, freezing, thawing, and grouping GPU threads.
- Executing all the threads of a tile to a specific location in code.

## Prerequisites

Before you start this walkthrough:

- Read [C++ AMP Overview](#).
- Make sure that line numbers are displayed in the text editor. For more information, see [How to: Display Line Numbers in the Editor](#).
- Make sure you are running at least Windows 8 or Windows Server 2012 to support debugging on the software emulator.

### NOTE

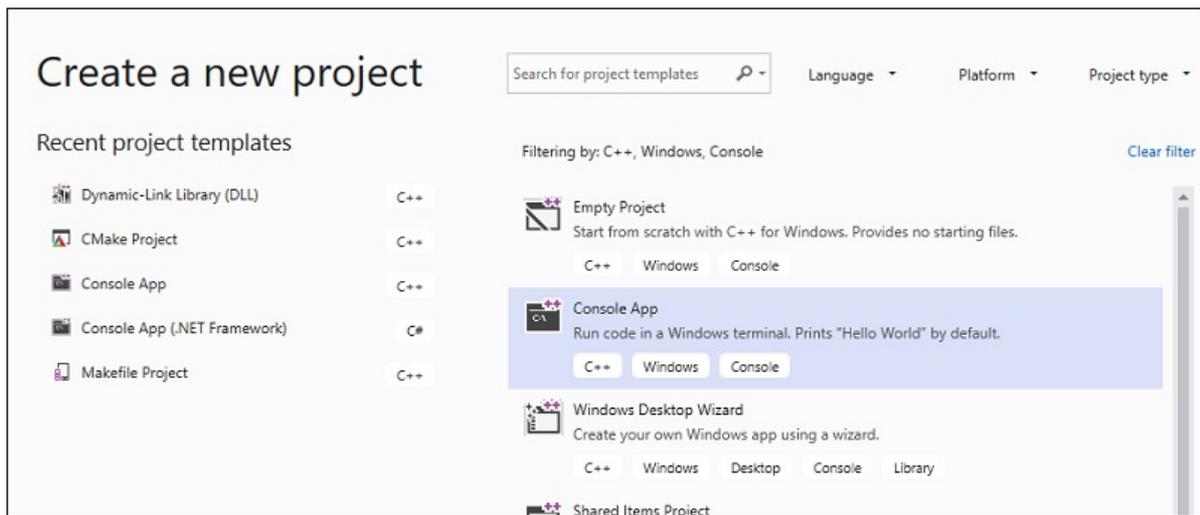
Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see [Personalizing the IDE](#).

### To create the sample project

The instructions for creating a project vary depending on which version of Visual Studio you are using. Make sure you have the correct version selected in the upper left of this page.

#### To create the sample project in Visual Studio 2019

1. On the menu bar, choose **File > New > Project** to open the **Create a New Project** dialog box.
2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Console**.
3. From the filtered list of project types, choose **Console App** then choose **Next**. In the next page, enter `AMPMAPReduce` in the **Name** box to specify a name for the project, and specify the project location if desired.



4. Choose the **Create** button to create the client project.

### To create the sample project in Visual Studio 2017 or Visual Studio 2015

1. Start Visual Studio.
2. On the menu bar, choose **File > New > Project**.
3. Under **Installed** in the templates pane, choose **Visual C++**.
4. Choose **Win32 Console Application**, type `AMPMAPReduce` in the **Name** box, and then choose the **OK** button.
5. Choose the **Next** button.
6. Clear the **Precompiled header** check box, and then choose the **Finish** button.
7. In **Solution Explorer**, delete `stdafx.h`, `targetver.h`, and `stdafx.cpp` from the project.
8. Open `AMPMAPReduce.cpp` and replace its content with the following code.

```
// AMPMAPReduce.cpp defines the entry point for the program.
// The program performs a parallel-sum reduction that computes the sum of an array of integers.

#include <stdio.h>
#include <tchar.h>
#include <amp.h>

const int BLOCK_DIM = 32;

using namespace concurrency;

void sum_kernel_tiled(tiled_index<BLOCK_DIM> t_idx, array<int, 1> &A, int stride_size) restrict(amp)
{
    tile_static int localA[BLOCK_DIM];

    index<1> globalIdx = t_idx.global * stride_size;
    index<1> localIdx = t_idx.local;

    localA[localIdx[0]] = A[globalIdx];

    t_idx.barrier.wait();

    // Aggregate all elements in one tile into the first element.
    for (int i = BLOCK_DIM / 2; i > 0; i /= 2)
    {
        if (localIdx[0] < i)
        {
```

```

        localA[localIdx[0]] += localA[localIdx[0] + i];
    }

    t_idx.barrier.wait();
}

if (localIdx[0] == 0)
{
    A[globalIdx] = localA[0];
}
}

int size_after_padding(int n)
{
    // The extent might have to be slightly bigger than num_stride to
    // be evenly divisible by BLOCK_DIM. You can do this by padding with zeros.
    // The calculation to do this is BLOCK_DIM * ceil(n / BLOCK_DIM)
    return ((n - 1) / BLOCK_DIM + 1) * BLOCK_DIM;
}

int reduction_sum_gpu_kernel(array<int, 1> input)
{
    int len = input.extent[0];

    //Tree-based reduction control that uses the CPU.
    for (int stride_size = 1; stride_size < len; stride_size *= BLOCK_DIM)
    {
        // Number of useful values in the array, given the current
        // stride size.
        int num_strides = len / stride_size;

        extent<1> e(size_after_padding(num_strides));

        // The sum kernel that uses the GPU.
        parallel_for_each(extent<1>(e).tile<BLOCK_DIM>(), [&input, stride_size] (tiled_index<BLOCK_DIM>
idx) restrict(amp)
        {
            sum_kernel_tiled(idx, input, stride_size);
        });
    }

    array_view<int, 1> output = input.section(extent<1>(1));
    return output[0];
}

int cpu_sum(const std::vector<int> &arr) {
    int sum = 0;
    for (size_t i = 0; i < arr.size(); i++) {
        sum += arr[i];
    }
    return sum;
}

std::vector<int> rand_vector(unsigned int size) {
    srand(2011);

    std::vector<int> vec(size);
    for (size_t i = 0; i < size; i++) {
        vec[i] = rand();
    }
    return vec;
}

array<int, 1> vector_to_array(const std::vector<int> &vec) {
    array<int, 1> arr(vec.size());
    copy(vec.begin(), vec.end(), arr);
    return arr;
}

```

```

int _tmain(int argc, _TCHAR* argv[])
{
    std::vector<int> vec = rand_vector(10000);
    array<int, 1> arr = vector_to_array(vec);

    int expected = cpu_sum(vec);
    int actual = reduction_sum_gpu_kernel(arr);

    bool passed = (expected == actual);
    if (!passed) {
        printf("Actual (GPU): %d, Expected (CPU): %d", actual, expected);
    }
    printf("sum: %s\n", passed ? "Passed!" : "Failed!");

    getchar();

    return 0;
}

```

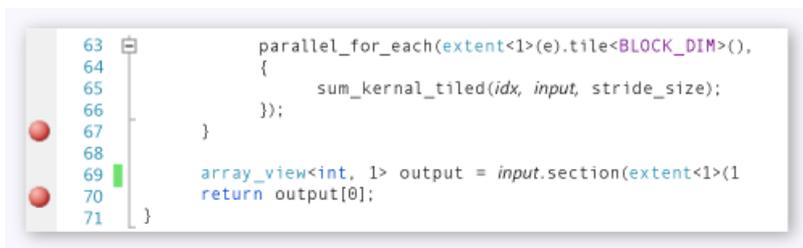
9. On the menu bar, choose **File > Save All**.
10. In **Solution Explorer**, open the shortcut menu for **AMPMapReduce**, and then choose **Properties**.
11. In the **Property Pages** dialog box, under **Configuration Properties**, choose **C/C++ > Precompiled Headers**.
12. For the **Precompiled Header** property, select **Not Using Precompiled Headers**, and then choose the **OK** button.
13. On the menu bar, choose **Build > Build Solution**.

## Debugging the CPU Code

In this procedure, you will use the Local Windows Debugger to make sure that the CPU code in this application is correct. The segment of the CPU code in this application that is especially interesting is the `for` loop in the `reduction_sum_gpu_kernel` function. It controls the tree-based parallel reduction that is run on the GPU.

### To debug the CPU code

1. In **Solution Explorer**, open the shortcut menu for **AMPMapReduce**, and then choose **Properties**.
2. In the **Property Pages** dialog box, under **Configuration Properties**, choose **Debugging**. Verify that **Local Windows Debugger** is selected in the **Debugger to launch** list.
3. Return to the **Code Editor**.
4. Set breakpoints on the lines of code shown in the following illustration (approximately lines 67 line 70).



CPU breakpoints

5. On the menu bar, choose **Debug > Start Debugging**.
6. In the **Locals** window, observe the value for `stride_size` until the breakpoint at line 70 is reached.
7. On the menu bar, choose **Debug > Stop Debugging**.

# Debugging the GPU Code

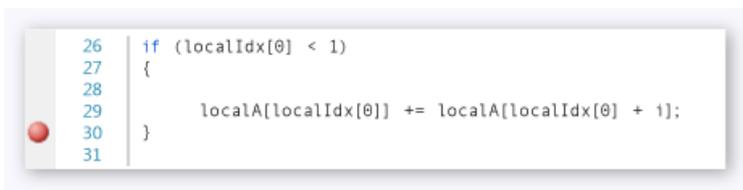
This section shows how to debug the GPU code, which is the code contained in the `sum_kernel_tiled` function. The GPU code computes the sum of integers for each "block" in parallel.

## To debug the GPU code

1. In **Solution Explorer**, open the shortcut menu for **AMPMapReduce**, and then choose **Properties**.
2. In the **Property Pages** dialog box, under **Configuration Properties**, choose **Debugging**.
3. In the **Debugger to launch** list, select **Local Windows Debugger**.
4. In the **Debugger Type** list, verify that **Auto** is selected.

**Auto** is the default value. Prior to Windows 10, **GPU Only** is the required value instead of **Auto**.

5. Choose the **OK** button.
6. Set a breakpoint at line 30, as shown in the following illustration.



GPU breakpoint

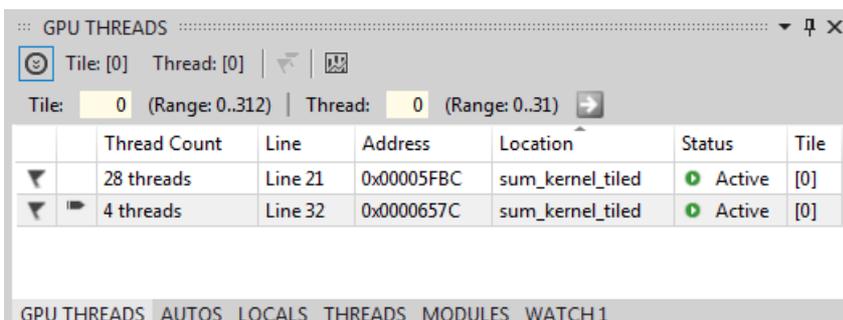
7. On the menu bar, choose **Debug > Start Debugging**. The breakpoints in the CPU code at lines 67 and 70 are not executed during GPU debugging because those lines of code are executed on the CPU.

## To use the GPU Threads window

1. To open the **GPU Threads** window, on the menu bar, choose **Debug > Windows > GPU Threads**.

You can inspect the state the GPU threads in the **GPU Threads** window that appears.

2. Dock the **GPU Threads** window at the bottom of Visual Studio. Choose the **Expand Thread Switch** button to display the tile and thread text boxes. The **GPU Threads** window shows the total number of active and blocked GPU threads, as shown in the following illustration.



GPU Threads window

There are 313 tiles allocated for this computation. Each tile contains 32 threads. Because local GPU debugging occurs on a software emulator, there are four active GPU threads. The four threads execute the instructions simultaneously and then move on together to the next instruction.

In the **GPU Threads** window, there are four GPU threads active and 28 GPU threads blocked at the `tile_barrier::wait` statement defined at about line 21 (`t_idx.barrier.wait();`). All 32 GPU threads belong to the first tile, `tile[0]`. An arrow points to the row that includes the current thread. To switch to a different thread, use one of the following methods:

- In the row for the thread to switch to in the **GPU Threads** window, open the shortcut menu and

choose **Switch To Thread**. If the row represents more than one thread, you will switch to the first thread according to the thread coordinates.

- Enter the tile and thread values of the thread in the corresponding text boxes and then choose the **Switch Thread** button.

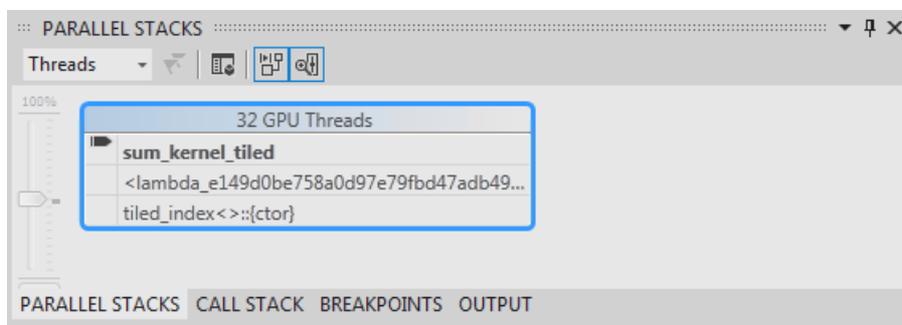
The **Call Stack** window displays the call stack of the current GPU thread.

### To use the Parallel Stacks window

1. To open the **Parallel Stacks** window, on the menu bar, choose **Debug > Windows > Parallel Stacks**.

You can use the **Parallel Stacks** window to simultaneously inspect the stack frames of multiple GPU threads.

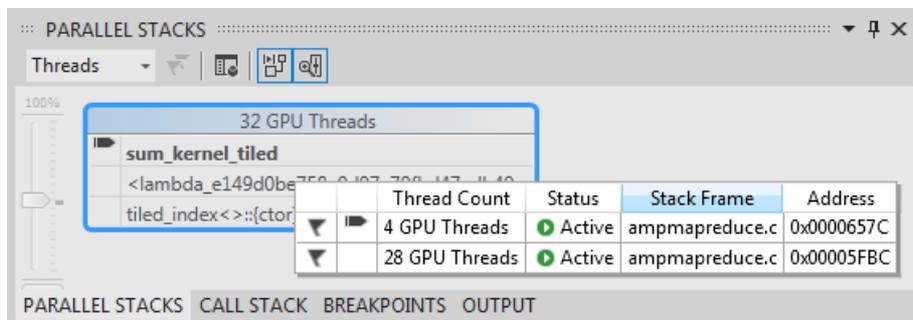
2. Dock the **Parallel Stacks** window at the bottom of Visual Studio.
3. Make sure that **Threads** is selected in the list in the upper-left corner. In the following illustration, the **Parallel Stacks** window shows a call-stack focused view of the GPU threads that you saw in the **GPU Threads** window.



Parallel Stacks window

32 threads went from `_kernel_stub` to the lambda statement in the `parallel_for_each` function call and then to the `sum_kernel_tiled` function, where the parallel reduction occurs. 28 out of the 32 threads have progressed to the `tile_barrier::wait` statement and remain blocked at line 22, whereas the other 4 threads remain active in the `sum_kernel_tiled` function at line 30.

You can inspect the properties of a GPU thread that are available in the **GPU Threads** window in the rich DataTip of the **Parallel Stacks** window. To do this, rest the mouse pointer on the stack frame of `sum_kernel_tiled`. The following illustration shows the DataTip.



GPU thread DataTip

For more information about the **Parallel Stacks** window, see [Using the Parallel Stacks Window](#).

### To use the Parallel Watch window

1. To open the **Parallel Watch** window, on the menu bar, choose **Debug > Windows > Parallel Watch > Parallel Watch 1**.

You can use the **Parallel Watch** window to inspect the values of an expression across multiple threads.

2. Dock the **Parallel Watch 1** window to the bottom of Visual Studio. There are 32 rows in the table of the **Parallel Watch** window. Each corresponds to a GPU thread that appeared in both the GPU Threads window and the **Parallel Stacks** window. Now, you can enter expressions whose values you want to inspect across all 32 GPU threads.
3. Select the **Add Watch** column header, enter `localIdx`, and then choose the **Enter** key.
4. Select the **Add Watch** column header again, type `globalIdx`, and then choose the **Enter** key.
5. Select the **Add Watch** column header again, type `localA[localIdx[0]]`, and then choose the **Enter** key.

You can sort by a specified expression by selecting its corresponding column header.

Select the `localA[localIdx[0]]` column header to sort the column. The following illustration shows the results of sorting by `localA[localIdx[0]]`.

[Tile][Thread]	localIdx	globalIdx	localA[localIdx[0]]	<Add Watch>
[0] [0]	(0)	(0)	10406	
[0] [1]	(1)	(1)	47454	
[0] [2]	(2)	(2)	46916	
[0] [3]	(3)	(3)	33992	
[0] [4]	(4)	(4)	9320	
[0] [5]	(5)	(5)	11363	
[0] [6]	(6)	(6)	8816	

Results of sort

You can export the content in the **Parallel Watch** window to Excel by choosing the **Excel** button and then choosing **Open in Excel**. If you have Excel installed on your development computer, this opens an Excel worksheet that contains the content.

6. In the upper-right corner of the **Parallel Watch** window, there's a filter control that you can use to filter the content by using Boolean expressions. Enter `localA[localIdx[0]] > 20000` in the filter control text box and then choose the **Enter** key.

The window now contains only threads on which the `localA[localIdx[0]]` value is greater than 20000. The content is still sorted by the `localA[localIdx[0]]` column, which is the sorting action you performed earlier.

## Flagging GPU Threads

You can mark specific GPU threads by flagging them in the **GPU Threads** window, the **Parallel Watch** window, or the DataTip in the **Parallel Stacks** window. If a row in the GPU Threads window contains more than one thread, flagging that row flags all threads that are contained in the row.

### To flag GPU threads

1. Select the **[Thread]** column header in the **Parallel Watch 1** window to sort by tile index and thread index.
2. On the menu bar, choose **Debug > Continue**, which causes the four threads that were active to progress to the next barrier (defined at line 32 of AMPMapReduce.cpp).
3. Choose the flag symbol on the left side of the row that contains the four threads that are now active.

The following illustration shows the four active flagged threads in the **GPU Threads** window.

	Thread Count	Line	Address	Location	Status	Tile
▼	24 threads	Line 21	0x00005FBC	sum_kernel_tiled	Active	[0]
▼	4 threads	Line 32	0x0000657C	sum_kernel_tiled	Active	[0]
▼	4 threads	Line 32	0x000065AC	sum_kernel_tiled	Blocked	[0]

Active threads in the GPU Threads window

The **Parallel Watch** window and the DataTip of the **Parallel Stacks** window both indicate the flagged threads.

4. If you want to focus on the four threads that you flagged, you can choose to show, in the **GPU Threads**, **Parallel Watch**, and **Parallel Stacks** windows, only the flagged threads.

Choose the **Show Flagged Only** button on any of the windows or on the **Debug Location** toolbar. The following illustration shows the **Show Flagged Only** button on the **Debug Location** toolbar.



**Show Flagged Only** button

Now the **GPU Threads**, **Parallel Watch**, and **Parallel Stacks** windows display only the flagged threads.

## Freezing and Thawing GPU Threads

You can freeze (suspend) and thaw (resume) GPU threads from either the **GPU Threads** window or the **Parallel Watch** window. You can freeze and thaw CPU threads the same way; for information, see [How to: Use the Threads Window](#).

### To freeze and thaw GPU threads

1. Choose the **Show Flagged Only** button to display all the threads.
2. On the menu bar, choose **Debug > Continue**.
3. Open the shortcut menu for the active row and then choose **Freeze**.

The following illustration of the **GPU Threads** window shows that all four threads are frozen.

	Thread Count	Line	Address	Location	Status	Tile
▼	4 threads	Line 32	0x0000657C	sum_kernel_tiled	Active	[0]
▼	20 threads	Line 21	0x00005FBC	sum_kernel_tiled	Active	[0]
▼	4 threads	Line 32	0x000065AC	sum_kernel_tiled	Blocked	[0]
▼	4 threads	Line 32	0x000065AC	sum_kernel_tiled	Blocked	[0]

Frozen threads in the **GPU Threads** window

Similarly, the **Parallel Watch** window shows that all four threads are frozen.

4. On the menu bar, choose **Debug > Continue** to allow the next four GPU threads to progress past the barrier at line 22 and to reach the breakpoint at line 30. The **GPU Threads** window shows that the four previously frozen threads remain frozen and in the active state.

- On the menu bar, choose **Debug, Continue**.
- From the **Parallel Watch** window, you can also thaw individual or multiple GPU threads.

### To group GPU threads

- On the shortcut menu for one of the threads in the **GPU Threads** window, choose **Group By, Address**.

The threads in the **GPU Threads** window are grouped by address. The address corresponds to the instruction in disassembly where each group of threads is located. 24 threads are at line 22 where the `tile_barrier::wait Method` is executed. 12 threads are at the instruction for the barrier at line 32. Four of these threads are flagged. Eight threads are at the breakpoint at line 30. Four of these threads are frozen. The following illustration shows the grouped threads in the **GPU Threads** window.

	Thread Count	Line	Address	Location	Status	Tile
<b>Address: 0x0000657C (8 Threads)</b>						
▼	4 threads	Line 32	0x0000657C	sum_kernel_tiled	● Active	[0]
▼	4 threads	Line 32	0x0000657C	sum_kernel_tiled	● Active	[0]
<b>Address: 0x00005FBC (12 Threads)</b>						
▼	12 threads	Line 21	0x00005FBC	sum_kernel_tiled	● Active	[0]
<b>Address: 0x000065AC (12 Threads)</b>						
▼	8 threads	Line 32	0x000065AC	sum_kernel_tiled	● Blocked	[0]
▼	4 threads	Line 32	0x000065AC	sum_kernel_tiled	● Blocked	[0]

Grouped threads in the **GPU Threads** window

- You can also perform the **Group By** operation by opening the shortcut menu for the data grid of the **Parallel Watch** window, choosing **Group By**, and then choosing the menu item that corresponds to how you want to group the threads.

## Running All Threads to a Specific Location in Code

You run all the threads in a given tile to the line that contains the cursor by using **Run Current Tile To Cursor**.

### To run all threads to the location marked by the cursor

- On the shortcut menu for the frozen threads, choose **Thaw**.
- In the **Code Editor**, put the cursor in line 30.
- On the shortcut menu for the **Code Editor**, choose **Run Current Tile To Cursor**.

The 24 threads that were previously blocked at the barrier at line 21 have progressed to line 32. This is shown in the **GPU Threads** window.

## See also

- [C++ AMP Overview](#)
- [Debugging GPU Code](#)
- [How to: Use the GPU Threads Window](#)
- [How to: Use the Parallel Watch Window](#)
- [Analyzing C++ AMP Code with the Concurrency Visualizer](#)

# Using Lambdas, Function Objects, and Restricted Functions

3/4/2019 • 3 minutes to read • [Edit Online](#)

The C++ AMP code that you want to run on the accelerator is specified as an argument in a call to the [parallel\\_for\\_each](#) method. You can provide either a lambda expression or a function object (functor) as that argument. Additionally, the lambda expression or function object can call a C++ AMP-restricted function. This topic uses an array addition algorithm to demonstrate lambdas, function objects, and restricted functions. The following example shows the algorithm without C++ AMP code. Two 1-dimensional arrays of equal length are created. The corresponding integer elements are added and stored in a third 1-dimensional array. C++ AMP is not used.

```
void CpuMethod() {  
  
    int aCPP[] = {1, 2, 3, 4, 5};  
    int bCPP[] = {6, 7, 8, 9, 10};  
    int sumCPP[5];  
  
    for (int idx = 0; idx <5; idx++)  
    {  
        sumCPP[idx] = aCPP[idx] + bCPP[idx];  
    }  
  
    for (int idx = 0; idx <5; idx++)  
    {  
        std::cout <<sumCPP[idx] <<"\n";  
    }  
}
```

## Lambda Expression

Using a lambda expression is the most direct way to use C++ AMP to rewrite the code.

```

void AddArraysWithLambda() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[5];

    array_view<const int, 1> a(5, aCPP);

    array_view<const int, 1> b(5, bCPP);

    array_view<int, 1> sum(5, sumCPP);

    sum.discard_data();

    parallel_for_each(
        sum.extent,
        [=](index<1> idx) restrict(amp)
        {
            sum[idx] = a[idx] + b[idx];
        });

    for (int i = 0; i < 5; i++) {
        std::cout << sum[i] << "\n";
    }
}

```

The lambda expression must include one indexing parameter and must include `restrict(amp)`. In the example, the `array_view` `sum` object has a rank of 1. Therefore, the parameter to the lambda statement is an `index` object that has rank 1. At runtime, the lambda expression is executed once for each element in the `array_view` object. For more information, see [Lambda Expression Syntax](#).

## Function Object

You can factor the accelerator code into a function object.

```

class AdditionFunctionObject
{
public:
    AdditionFunctionObject(const array_view<int, 1>& a,
        const array_view<int, 1>& b,
        const array_view<int, 1>& sum)
        : a(a), b(b), sum(sum)
        {
        }

    void operator()(index<1> idx) restrict(amp)
    {
        sum[idx] = a[idx] + b[idx];
    }

private:
    array_view<int, 1> a;
    array_view<int, 1> b;
    array_view<int, 1> sum;
};

void AddArraysWithFunctionObject() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[5];

    array_view<const int, 1> a(5, aCPP);

    array_view<const int, 1> b(5, bCPP);

    array_view<int, 1> sum(5, sumCPP);

    sum.discard_data();

    parallel_for_each(
        sum.extent,
        AdditionFunctionObject(a, b, sum));

    for (int i = 0; i < 5; i++) {
        std::cout <<sum[i] <<"\n";
    }
}

```

The function object must include a constructor and must include an overload of the function call operator. The function call operator must include one indexing parameter. An instance of the function object is passed as the second argument to the `parallel_for_each` method. In this example, three `array_view` objects are passed to the function object constructor. The `array_view` object `sum` has a rank of 1. Therefore, the parameter to the function call operator is an `index` object that has rank 1. At runtime, the function is executed once for each element in the `array_view` object. For more information, see [Function Call](#) and [Function Objects in the C++ Standard Library](#).

## C++ AMP-Restricted Function

You can further factor the accelerator code by creating a restricted function and calling it from a lambda expression or a function object. The following code example demonstrates how to call a restricted function from a lambda expression.

```

void AddElementsWithRestrictedFunction(index<1> idx, array_view<int, 1> sum, array_view<int, 1> a,
array_view<int, 1> b) restrict(amp)
{
    sum[idx] = a[idx] + b[idx];
}

void AddArraysWithFunction() {

    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[5];

    array_view<int, 1> a(5, aCPP);

    array_view<int, 1> b(5, bCPP);

    array_view<int, 1> sum(5, sumCPP);

    sum.discard_data();

    parallel_for_each(
        sum.extent,
        [=](index<1> idx) restrict(amp)
        {
            AddElementsWithRestrictedFunction(idx, sum, a, b);
        });

    for (int i = 0; i < 5; i++) {
        std::cout <<sum[i] <<"\n";
    }
}

```

The restricted function must include `restrict(amp)` and conform to the restrictions that are described in [restrict \(C++ AMP\)](#).

## See also

[C++ AMP \(C++ Accelerated Massive Parallelism\)](#)

[Lambda Expression Syntax](#)

[Function Call](#)

[Function Objects in the C++ Standard Library](#)

[restrict \(C++ AMP\)](#)

# Graphics (C++ AMP)

3/4/2019 • 17 minutes to read • [Edit Online](#)

C++ AMP contains several APIs in the [Concurrency::graphics](#) namespace that you can use to access the texture support on GPUs. Some common scenarios are:

- You can use the [texture](#) class as a data container for computation and exploit the *spatial locality* of the texture cache and layouts of GPU hardware. Spatial locality is the property of data elements being physically close to each other.
- The runtime provides efficient interoperability with non-compute shaders. Pixel, vertex, tessellation, and hull shaders frequently consume or produce textures that you can use in your C++ AMP computations.
- The graphics APIs in C++ AMP provide alternative ways to access sub-word packed buffers. Textures that have formats that represent *texels* (texture elements) that are composed of 8-bit or 16-bit scalars allow access to such packed data storage.

## The norm and unorm Types

The `norm` and `unorm` types are scalar types that limit the range of **float** values; this is known as *clamping*. These types can be explicitly constructed from other scalar types. In casting, the value is first cast to **float** and then clamped to the respective region that's allowed by norm [-1.0, 1.0] or unorm [0.0, 1.0]. Casting from +/- infinity returns +/-1. Casting from NaN is undefined. A norm can be implicitly constructed from a unorm and there is no loss of data. The implicit conversion operator to float is defined on these types. Binary operators are defined between these types and other built-in scalar types such as **float** and **int**: +, -, \*, /, ==, !=, >, <, >=, <=. The compound assignment operators are also supported: +=, -=, \*=, /=. The unary negation operator (-) is defined for norm types.

## Short Vector Library

The Short Vector Library provides some of the functionality of the [Vector Type](#) that's defined in HLSL and is typically used to define texels. A short vector is a data structure that holds one to four values of the same type. The supported types are **double**, **float**, **int**, `norm`, `uint`, and `unorm`. The type names are shown in the following table. For each type, there is also a corresponding **typedef** that doesn't have an underscore in the name. The types that have the underscores are in the [Concurrency::graphics Namespace](#). The types that don't have the underscores are in the [Concurrency::graphics::direct3d Namespace](#) so that they are clearly separated from the similarly-named fundamental types such as `__int8` and `__int16`.

	LENGTH 2	LENGTH 3	LENGTH 4
double	double_2	double_3	double_4
	double2	double3	double4
float	float_2	float_3	float_4
	float2	float3	float4
int	int_2	int_3	int_4
	int2	int3	int4

	LENGTH 2	LENGTH 3	LENGTH 4
norm	norm_2	norm_3	norm_4
	norm2	norm3	norm4
uint	uint_2	uint_3	uint_4
	uint2	uint3	uint4
unorm	unorm_2	unorm_3	unorm_4
	unorm2	unorm3	unorm4

## Operators

If an operator is defined between two short vectors, then it is also defined between a short vector and a scalar. Also, one of these must be true:

- The scalar's type must be the same as the short vector's element type.
- The scalar's type can be implicitly converted to the vector's element type by using only one user-defined conversion.

The operation is carried component-wise between each component of the short vector and the scalar. Here are the valid operators:

OPERATOR TYPE	VALID TYPES
Binary operators	Valid on all types: +, -, *, /, Valid on integer types: %, ^,  , &, <<, >> The two vectors must have the same size, and the result is a vector of the same size.
Relational operators	Valid on all types: == and !=
Compound assignment operator	Valid on all types: +=, -=, *=, /= Valid on integer types: %=, ^=,  =, &=, <<=, >>=
Increment and decrement operators	Valid on all types: ++, -- Both prefix and postfix are valid.
Bitwise NOT operator (~)	Valid on integer types.
Unary - operator	Valid on all types except <code>unorm</code> and <code>uint</code> .

## Swizzling Expressions

The Short Vector Library supports the `vector_type.identifier` accessor construct to access the components of a short vector. The `identifier`, which is known as a *swizzling expression*, specifies the components of the vector. The expression can be an l-value or an r-value. Individual characters in the identifier may be: x, y, z, and w; or r, g, b, and a. "x" and "r" mean the zero-th component, "y" and "g" mean the first component, and so on. (Notice that "x" and "r" cannot be used in the same identifier.) Therefore, "rgba" and "xyzw" return the same result. Single-component accessors such as "x" and "y" are scalar value types. Multi-component accessors are short vector types. For

example, if you construct an `int_4` vector that's named `fourInts` and has the values 2, 4, 6, and 8, then `fourInts.y` returns the integer 4 and `fourInts.rg` returns an `int_2` object that has the values 2 and 4.

## Texture Classes

Many GPUs have hardware and caches that are optimized to fetch pixels and texels and to render images and textures. The `texture<T,N>` class, which is a container class for texel objects, exposes the texture functionality of these GPUs. A texel can be:

- An **int**, `uint`, **float**, **double**, `norm`, or `unorm` scalar.
- A short vector that has two or four components. The only exception is `double_4`, which is not allowed.

The `texture` object can have a rank of 1, 2, or 3. The `texture` object can be captured only by reference in the lambda of a call to `parallel_for_each`. The texture is stored on the GPU as Direct3D texture objects. For more information about textures and texels in Direct3D, see [Introduction to Textures in Direct3D 11](#).

The texel type you use might be one of the many texture formats that are used in graphics programming. For example, an RGBA format could use 32 bits, with 8 bits each for the R, G, B, and A scalar elements. The texture hardware of a graphics card can access the individual elements based on the format. For example, if you are using the RGBA format, the texture hardware can extract each 8-bit element into a 32-bit form. In C++ AMP, you can set the bits per scalar element of your texel so that you can automatically access the individual scalar elements in the code without using bit-shifting.

### Instantiating Texture Objects

You can declare a texture object without initialization. The following code example declares several texture objects.

```
#include <amp.h>
#include <amp_graphics.h>
using namespace concurrency;
using namespace concurrency::graphics;

void declareTextures() {
    // Create a 16-texel texture of int.
    texture<int, 1> intTexture1(16);
    texture<int, 1> intTexture2(extent<1>(16));

    // Create a 16 x 32 texture of float_2.
    texture<float_2, 2> floatTexture1(16, 32);
    texture<float_2, 2> floatTexture2(extent<2>(16, 32));

    // Create a 2 x 4 x 8 texture of uint_4.
    texture<uint_4, 3> uintTexture1(2, 4, 8);
    texture<uint_4, 3> uintTexture2(extent<3>(2, 4, 8));
}
```

You can also use a constructor to declare and initialize a `texture` object. The following code example instantiates a `texture` object from a vector of `float_4` objects. The bits per scalar element is set to the default. You cannot use this constructor with `norm`, `unorm`, or the short vectors of `norm` and `unorm`, because they do not have a default bits per scalar element.

```

#include <amp.h>
#include <amp_graphics.h>
#include <vector>
using namespace concurrency;
using namespace concurrency::graphics;

void initializeTexture() {
    std::vector<int_4> texels;
    for (int i = 0; i < 768 * 1024; i++) {
        int_4 i4(i, i, i, i);
        texels.push_back(i4);
    }

    texture<int_4, 2> aTexture(768, 1024, texels.begin(), texels.end());
}

```

You can also declare and initialize a `texture` object by using a constructor overload that takes a pointer to the source data, the size of source data in bytes, and the bits per scalar element.

```

void createTextureWithBPC() { // Create the source data.
    float source[1024* 2];
    for (int i = 0; i <1024* 2; i++) {
        source[i] = (float)i;
    }
    // Initialize the texture by using the size of source in bytes // and bits per scalar element.
    texture<float_2, 1> floatTexture(1024, source, (unsigned int)sizeof(source), 32U);
}

```

The textures in these examples are created on the default view of the default accelerator. You can use other overloads of the constructor if you want to specify an `accelerator_view` object. You cannot create a texture object on a CPU accelerator.

There are limits on the size of each dimension of the `texture` object, as shown in the following table. A run-time error is generated if you exceed the limits.

TEXTURE	SIZE LIMITATION PER DIMENSION
<code>texture&lt;T,1&gt;</code>	16384
<code>texture&lt;T,2&gt;</code>	16384
<code>texture&lt;T,3&gt;</code>	2048

### Reading from Texture Objects

You can read from a `texture` object by using `texture::operator[]`, `texture::operator() Operator`, or `texture::get Method`. The two operators return a value, not a reference. Therefore, you cannot write to a `texture` object by using `texture::operator\[\]`.

```

void readTexture() {
    std::vector<int_2> src;
    for (int i = 0; i <16 *32; i++) {
        int_2 i2(i, i);

        src.push_back(i2);
    }

    std::vector<int_2> dst(16* 32);

    array_view<int_2, 2> arr(16, 32, dst);

    arr.discard_data();

    const texture<int_2, 2> tex9(16, 32, src.begin(), src.end());

    parallel_for_each(tex9.extent, [=, &tex9] (index<2> idx) restrict(amp) { // Use the subscript operator.
        arr[idx].x += tex9[idx].x; // Use the function () operator.
        arr[idx].x += tex9(idx).x; // Use the get method.
        arr[idx].y += tex9.get(idx).y; // Use the function () operator.
        arr[idx].y += tex9(idx[0], idx[1]).y;
    });

    arr.synchronize();
}

```

The following code example demonstrates how to store texture channels in a short vector, and then access the individual scalar elements as properties of the short vector.

```

void UseBitsPerScalarElement() { // Create the image data. // Each unsigned int (32-bit) represents four 8-bit
scalar elements(r,g,b,a values).
    const int image_height = 16;
    const int image_width = 16;
    std::vector<unsigned int> image(image_height* image_width);

    extent<2> image_extent(image_height, image_width);

    // By using uint_4 and 8 bits per channel, each 8-bit channel in the data source is // stored in one 32-
bit component of a uint_4.
    texture<uint_4, 2> image_texture(image_extent, image.data(), image_extent.size()* 4U, 8U);

    // Use can access the RGBA values of the source data by using swizzling expressions of the uint_4.
    parallel_for_each(image_extent,
        [&image_texture](index<2> idx) restrict(amp)
        { // 4 bytes are automatically extracted when reading.
            uint_4 color = image_texture[idx];
            unsigned int r = color.r;
            unsigned int g = color.g;
            unsigned int b = color.b;
            unsigned int a = color.a;
        });
}

```

The following table lists the valid bits per channel for each sort vector type.

TEXTURE DATA TYPE	VALID BITS PER SCALAR ELEMENT
int, int_2, int_4 uint, uint_2, uint_4	8, 16, 32
int_3, uint_3	32

TEXTURE DATA TYPE	VALID BITS PER SCALAR ELEMENT
float, float_2, float_4	16, 32
float_3	32
double, double_2	64
norm, norm_2, norm_4 unorm, unorm_2, unorm, 4	8, 16

### Writing to Texture Objects

Use the `texture::set` method to write to `texture` objects. A texture object can be readonly or read/write. For a texture object to be readable and writeable, the following conditions must be true:

- T has only one scalar component. (Short vectors are not allowed.)
- T is not **double**, `norm`, or `unorm`.
- The `texture::bits_per_scalar_element` property is 32.

If all three are not true, then the `texture` object is readonly. The first two conditions are checked during compilation. A compilation error is generated if you have code that tries to write to a `readonly` texture object. The condition for `texture::bits_per_scalar_element` is detected at run time, and the runtime generates the `unsupported_feature` exception if you try to write to a readonly `texture` object.

The following code example writes values to a texture object.

```
void writeTexture() {
    texture<int, 1> tex1(16);

    parallel_for_each(tex1.extent, [&tex1] (index<1> idx) restrict(amp) {
        tex1.set(idx, 0);
    });
}
```

### Copying Texture Objects

You can copy between texture objects by using the `copy` function or the `copy_async` function, as shown in the following code example.

```

void copyHostArrayToTexture() { // Copy from source array to texture object by using the copy function.
    float floatSource[1024* 2];
    for (int i = 0; i <1024* 2; i++) {
        floatSource[i] = (float)i;
    }
    texture<float_2, 1> floatTexture(1024);

    copy(floatSource, (unsigned int)sizeof(floatSource), floatTexture);

    // Copy from source array to texture object by using the copy function.
    char charSource[16* 16];
    for (int i = 0; i <16* 16; i++) {
        charSource[i] = (char)i;
    }
    texture<int, 2> charTexture(16, 16, 8U);

    copy(charSource, (unsigned int)sizeof(charSource), charTexture);
    // Copy from texture object to source array by using the copy function.
    copy(charTexture, charSource, (unsigned int)sizeof(charSource));
}

```

You can also copy from one texture to another by using the `texture::copy_to` method. The two textures can be on different accelerator\_views. When you copy to a `writeonly_texture_view` object, the data is copied to the underlying `texture` object. The bits per scalar element and the extent must be the same on the source and destination `texture` objects. If those requirements are not met, the runtime throws an exception.

## Texture View Classes

C++ AMP introduces the `texture_view Class` in Visual Studio 2013. Texture views support the same texel types and ranks as the `texture Class`, but unlike textures, they provide access to additional hardware features such as texture sampling and mipmaps. Texture views support read-only, write-only, and read-write access to the underlying texture data.

- Read-only access is provided by the `texture_view<const T, N>` template specialization, which supports elements that have 1, 2, or 4 components, texture sampling, and dynamic access to a range of mipmap levels that are determined when the view is instantiated.
- Write-only access is provided by the non-specialized template class `texture_view<T, N>`, which supports elements that have either 2 or 4 components and can access one mipmap level that's determined when the view is instantiated. It does not support sampling.
- Read-write access is provided by the non-specialized template class `texture_view<T, N>`, which, like textures, supports elements that have only one component; the view can access one mipmap level that's determined when it is instantiated. It does not support sampling.

Texture views are analogous to array views, but do not provide the automatic data management and movement functionality that the `array_view Class` provides over the `array class`. A `texture_view` can only be accessed on the accelerator view where the underlying texture data resides.

### **writeonly\_texture\_view Deprecated**

For Visual Studio 2013, C++ AMP introduces better support for hardware texture features such as sampling and mipmaps, which could not be supported by the `writeonly_texture_view Class`. The newly introduced `texture_view` class supports a superset of the functionality in `writeonly_texture_view`; as a result, `writeonly_texture_view` is deprecated.

We recommend—at least for new code—that you use `texture_view` to access functionality that was formerly provided by `writeonly_texture_view`. Compare the following two code examples that write to a texture object that has two components (`int_2`). Notice that in both cases, the view, `wo_tv4`, must be captured by value in the lambda

expression. Here is the example that uses the new `texture_view` class:

```
void write2ComponentTexture() {
    texture<int_2, 1> tex4(16);

    texture_view<int_2, 1> wo_tv4(tex4);

    parallel_for_each(extent<1>(16), [=] (index<1> idx) restrict(amp) {
        wo_tv4.set(idx, int_2(1, 1));
    });
}
```

And here is the deprecated `writeonly_texture_view` class:

```
void write2ComponentTexture() {
    texture<int_2, 1> tex4(16);

    writeonly_texture_view<int_2, 1> wo_tv4(tex4);

    parallel_for_each(extent<1>(16), [=] (index<1> idx) restrict(amp) {
        wo_tv4.set(idx, int_2(1, 1));
    });
}
```

As you can see, the two code examples are nearly identical when all you are doing is writing to the primary mipmap level. If you used `writeonly_texture_view` in existing code and you're not planning to enhance that code, you don't have to change it. However, if you're thinking about bringing that code forward, we suggest that you rewrite it to use `texture_view` because the enhancements in it support new hardware texture features. Read on for more information about these new capabilities.

For more information about the deprecation of `writeonly_texture_view`, see [Overview of the Texture View Design in C++ AMP](#) on the Parallel Programming in Native Code blog.

### Instantiating Texture View Objects

Declaring a `texture_view` is similar to declaring an `array_view` that's associated with an **array**. The following code example declares several `texture` objects and `texture_view` objects that are associated with them.

```
#include <amp.h>
#include <amp_graphics.h>
using namespace concurrency;
using namespace concurrency::graphics;

void declareTextureViews()
{
    // Create a 16-texel texture of int, with associated texture_views.
    texture<int, 1> intTexture(16);
    texture_view<const int, 1> intTextureViewRO(intTexture); // read-only
    texture_view<int, 1> intTextureViewRW(intTexture); // read-write

    // Create a 16 x 32 texture of float_2, with associated texture_views.
    texture<float_2, 2> floatTexture(16, 32);
    texture_view<const float_2, 2> floatTextureViewRO(floatTexture); // read-only
    texture_view<float_2, 2> floatTextureViewWO(floatTexture); // write-only

    // Create a 2 x 4 x 8 texture of uint_4, with associated texture_views.
    texture<uint_4, 3> uintTexture(2, 4, 8);
    texture_view<const uint_4, 3> uintTextureViewRO(uintTexture); // read-only
    texture_view<uint_4, 3> uintTextureViewWO(uintTexture); // write-only
}
```

Notice how a texture view whose element type is non-const and has one component is read-write, but a texture view whose element type is non-const but has more than one component are write-only. Texture views of const element types are always read-only, but if the element type is non-const, then the number of components in the element determines whether it is read-write (1 component) or write-only (multiple components).

The element type of a `texture_view`—its const-ness and also the number of components it has—also plays a role in determining whether the view supports texture sampling, and how mipmap levels can be accessed:

TYPE	COMPONENTS	READ	WRITE	SAMPLING	MIPMAP ACCESS
<code>texture_view&lt;const T, N&gt;</code>	1, 2, 4	Yes	No (1)	Yes	Yes, indexable. Range is determined at instantiation.
<code>Texture_view&lt;T, N&gt;</code>	1	Yes	Yes	No (1)	Yes, one level. Level is determined at instantiation.
	2, 4	No (2)	Yes	No (1)	Yes, one level. Level is determined at instantiation.

From this table, you can see that read-only texture views fully support the new capabilities in exchange for not being able to write to the view. Writable texture views are limited in that they can only access one mipmap level. Read-write texture views are even more specialized than writable ones, because they add the requirement that the element type of the texture view has only one component. Notice that sampling is not supported for writable texture views because it's a read-oriented operation.

### Reading from Texture View Objects

Reading unsampled texture data through a texture view is just like reading it from the texture itself, except that textures are captured by reference, whereas texture views are captured by value. The following two code examples demonstrate; first, by using `texture` only:

```
void write2ComponentTexture() {
    texture<int_2, 1> tex_data(16);

    parallel_for_each(extent<1>(16), [&] (index<1> idx) restrict(amp) {
        tex_data.set(idx, int_2(1, 1));
    });
}
```

And here is the same example, except it now uses the `texture_view` class:

```
void write2ComponentTexture() {
    texture<int_2, 1> tex_data(16);

    texture_view<int_2, 1> tex_view(tex_data);

    parallel_for_each(extent<1>(16), [=] (index<1> idx) restrict(amp) {
        tex_view.set(idx, int_2(1, 1));
    });
}
```

Texture views whose elements are based on floating-point types—for example, `float`, `float_2`, or `float_4`—can also be read by using texture sampling to take advantage of hardware support for various filtering modes and

addressing modes. C++ AMP supports the two filtering modes that are most common in compute scenarios—point-filtering (nearest-neighbor) and linear-filtering (weighted average)—and four addressing modes—wrapped, mirrored, clamped, and border. For more information about addressing modes, see [address\\_mode Enumeration](#).

In addition to modes that C++ AMP supports directly, you can access other filtering modes and addressing modes of the underlying platform by using the interop APIs to adopt a texture sampler that was created by using the platform APIs directly. For example, Direct3D supports other filtering modes such as anisotropic filtering, and can apply a different addressing mode to each dimension of a texture. You could create a texture sampler whose coordinates are wrapped vertically, mirrored horizontally, and sampled with anisotropic filtering by using the Direct3D APIs, and then leverage the sampler in your C++ AMP code by using the `make_sampler` interop API. For more information see [Texture Sampling in C++ AMP](#) on the Parallel Programming in Native Code blog.

Texture views also support the reading of mipmaps. Read-only texture views (those that have a const element type) offer the most flexibility because a range of mip-levels that is determined at instantiation can be dynamically sampled, and because elements that have 1, 2, or 4 components are supported. Read-write texture views that have elements that have one component also support mipmaps, but only of a level that's determined at instantiation. For more information, see [Texture with Mipmaps](#) on the Parallel Programming in Native Code blog.

### Writing to Texture View Objects

Use the `texture_view::get Method` to write to the underlying `texture` through the `texture_view` object. A texture view can be read-only, read-write, or write-only. For a texture view to be writable it must have an element type that is non-const; for a texture view to be readable and writable, its element type must also have only one component. Otherwise, the texture view is read-only. You can only access one mipmap level of a texture at a time through a texture view, and the level is specified when the view is instantiated.

This example shows how to write to the second-most detailed mipmap level of a texture that has 4 mipmap levels. The most detailed mipmap level is level 0.

```
// Create a texture that has 4 mipmap levels : 16x16, 8x8, 4x4, 2x2
texture<int, 2> tex(extent<2>(16, 16), 16U, 4);

// Create a writable texture view to the second mipmap level :4x4
texture_view<int, 2> w_view(tex, 1);

parallel_for_each(w_view.extent, [=](index<2> idx) restrict(amp)
{
    w_view.set(idx, 123);
});
```

## Interoperability

The C++ AMP runtime supports interoperability between `texture<T,1>` and the [ID3D11Texture1D interface](#), between `texture<T,2>` and the [ID3D11Texture2D interface](#), and between `texture<T,3>` and the [ID3D11Texture3D interface](#). The `get_texture` method takes a `texture` object and returns an `IUnknown` interface. The `make_texture` method takes an `IUnknown` interface and an `accelerator_view` object and returns a `texture` object.

## See also

[double\\_2 Class](#)

[double\\_3 Class](#)

[double\\_4 Class](#)

[float\\_2 Class](#)

[float\\_3 Class](#)

[float\\_4 Class](#)

[int\\_2 Class](#)

int\_3 Class  
int\_4 Class  
norm\_2 Class  
norm\_3 Class  
norm\_4 Class  
short\_vector Structure  
short\_vector\_traits Structure  
uint\_2 Class  
uint\_3 Class  
uint\_4 Class  
unorm\_2 Class  
unorm\_3 Class  
unorm\_4 Class

# Using accelerator and accelerator\_view Objects

3/4/2019 • 6 minutes to read • [Edit Online](#)

You can use the `accelerator` and `accelerator_view` classes to specify the device or emulator to run your C++ AMP code on. A system might have several devices or emulators that differ by amount of memory, shared memory support, debugging support, or double-precision support. C++ Accelerated Massive Parallelism (C++ AMP) provides APIs that you can use to examine the available accelerators, set one as the default, specify multiple `accelerator_views` for multiple calls to `parallel_for_each`, and perform special debugging tasks.

## Using the Default Accelerator

The C++ AMP runtime picks a default accelerator, unless you write code to pick a specific one. The runtime chooses the default accelerator as follows:

1. If the app is running in debug mode, an accelerator that supports debugging.
2. Otherwise, the accelerator that's specified by the `CPPAMP_DEFAULT_ACCELERATOR` environment variable, if it's set.
3. Otherwise, a non-emulated device.
4. Otherwise, the device that has the greatest amount of available memory.
5. Otherwise, a device that's not attached to the display.

Additionally, the runtime specifies an `access_type` of `access_type_auto` for the default accelerator. This means that the default accelerator uses shared memory if it's supported and if its performance characteristics (bandwidth and latency) are known to be the same as dedicated (non-shared) memory.

You can determine the properties of the default accelerator by constructing the default accelerator and examining its properties. The following code example prints the path, amount of accelerator memory, shared memory support, double-precision support, and limited double-precision support of the default accelerator.

```
void default_properties() {
    accelerator default_acc;
    std::wcout << default_acc.device_path << "\n";
    std::wcout << default_acc.dedicated_memory << "\n";
    std::wcout << (accs[i].supports_cpu_shared_memory ?
        "CPU shared memory: true" : "CPU shared memory: false") << "\n";
    std::wcout << (accs[i].supports_double_precision ?
        "double precision: true" : "double precision: false") << "\n";
    std::wcout << (accs[i].supports_limited_double_precision ?
        "limited double precision: true" : "limited double precision: false") << "\n";
}
```

### CPPAMP\_DEFAULT\_ACCELERATOR Environment Variable

You can set the `CPPAMP_DEFAULT_ACCELERATOR` environment variable to specify the

`accelerator::device_path` of the default accelerator. The path is hardware-dependent. The following code uses the `accelerator::get_all` function to retrieve a list of the available accelerators and then displays the path and characteristics of each accelerator.

```

void list_all_accelerators()
{
    std::vector<accelerator> accs = accelerator::get_all();

    for (int i = 0; i < accs.size(); i++) {
        std::wcout << accs[i].device_path << "\n";
        std::wcout << accs[i].dedicated_memory << "\n";
        std::wcout << (accs[i].supports_cpu_shared_memory ?
            "CPU shared memory: true" : "CPU shared memory: false") << "\n";
        std::wcout << (accs[i].supports_double_precision ?
            "double precision: true" : "double precision: false") << "\n";
        std::wcout << (accs[i].supports_limited_double_precision ?
            "limited double precision: true" : "limited double precision: false") << "\n";
    }
}

```

## Selecting an Accelerator

To select an accelerator, use the `accelerator::get_all` method to retrieve a list of the available accelerators and then select one based on its properties. This example shows how to pick the accelerator that has the most memory:

```

void pick_with_most_memory()
{
    std::vector<accelerator> accs = accelerator::get_all();
    accelerator acc_chosen = accs[0];

    for (int i = 0; i < accs.size(); i++) {
        if (accs[i].dedicated_memory > acc_chosen.dedicated_memory) {
            acc_chosen = accs[i];
        }
    }

    std::wcout << "The accelerator with the most memory is "
        << acc_chosen.device_path << "\n"
        << acc_chosen.dedicated_memory << ".\n";
}

```

### NOTE

One of the accelerators that are returned by `accelerator::get_all` is the CPU accelerator. You cannot execute code on the CPU accelerator. To filter out the CPU accelerator, compare the value of the `device_path` property of the accelerator that's returned by `accelerator::get_all` with the value of the `accelerator::cpu_accelerator`. For more information, see the "Special Accelerators" section in this article.

## Shared Memory

Shared memory is memory that can be accessed by both the CPU and the accelerator. The use of shared memory eliminates or significantly reduces the overhead of copying data between the CPU and the accelerator. Although the memory is shared, it cannot be accessed concurrently by both the CPU and the accelerator, and doing so causes undefined behavior. The accelerator property `supports_cpu_shared_memory` returns **true** if the accelerator supports shared memory, and the `default_cpu_access_type` property gets the default `access_type` for memory allocated on the `accelerator`—for example, **arrays** associated with the `accelerator`, or `array_view` objects accessed on the `accelerator`.

The C++ AMP runtime automatically chooses the best default `access_type` for each `accelerator`, but the performance characteristics (bandwidth and latency) of shared memory can be worse than those of dedicated (non-shared) accelerator memory when reading from the CPU, writing from the CPU, or both. If shared memory

performs as well as dedicated memory for reading and writing from the CPU, the runtime defaults to `access_type_read_write`; otherwise, the runtime chooses a more conservative default `access_type`, and allows the app to override it if the memory access patterns of its computation kernels benefit from a different `access_type`.

The following code example shows how to determine whether the default accelerator supports shared memory, and then overrides its default access type and creates an `accelerator_view` from it.

```
#include <amp.h>
#include <iostream>

using namespace Concurrency;

int main()
{
    accelerator acc = accelerator(accelerator::default_accelerator);

    // Early out if the default accelerator doesn't support shared memory.
    if (!acc.supports_cpu_shared_memory)
    {
        std::cout << "The default accelerator does not support shared memory" << std::endl;
        return 1;
    }

    // Override the default CPU access type.
    acc.set_default_cpu_access_type(access_type_read_write);

    // Create an accelerator_view from the default accelerator. The
    // accelerator_view reflects the default_cpu_access_type of the
    // accelerator it's associated with.
    accelerator_view acc_v = acc.default_view;
}
```

An `accelerator_view` always reflects the `default_cpu_access_type` of the `accelerator` it's associated with, and it provides no interface to override or change its `access_type`.

## Changing the Default Accelerator

You can change the default accelerator by calling the `accelerator::set_default` method. You can change the default accelerator only once per app execution and you must change it before any code is executed on the GPU. Any subsequent function calls to change the accelerator return **false**. If you want to use a different accelerator in a call to `parallel_for_each`, read the "Using Multiple Accelerators" section in this article. The following code example sets the default accelerator to one that is not emulated, is not connected to a display, and supports double-precision.

```

bool pick_accelerator()
{
    std::vector<accelerator> accs = accelerator::get_all();
    accelerator chosen_one;

    auto result = std::find_if(accs.begin(), accs.end(),
        [] (const accelerator& acc) {
            return !acc.is_emulated &&
                acc.supports_double_precision &&
                !acc.has_display;
        });

    if (result != accs.end()) {
        chosen_one = *(result);
    }

    std::wcout <<chosen_one.description <<std::endl;
    bool success = accelerator::set_default(chosen_one.device_path);
    return success;
}

```

## Using Multiple Accelerators

There are two ways to use multiple accelerators in your app:

- You can pass `accelerator_view` objects to the calls to the `parallel_for_each` method.
- You can construct an **array** object using a specific `accelerator_view` object. The C++ AMP runtime will pick up the `accelerator_view` object from the captured **array** object in the lambda expression.

## Special Accelerators

The device paths of three special accelerators are available as properties of the `accelerator` class:

- [`accelerator::direct3d\_ref` Data Member](#): This single-threaded accelerator uses software on the CPU to emulate a generic graphics card. It's used by default for debugging, but it's not useful in production because it's slower than the hardware accelerators. Additionally, it's available only in the DirectX SDK and the Windows SDK, and it's unlikely to be installed on your customers' computers. For more information, see [Debugging GPU Code](#).
- [`accelerator::direct3d\_warp` Data Member](#): This accelerator provides a fallback solution for executing C++ AMP code on multi-core CPUs that use Streaming SIMD Extensions (SSE).
- [`accelerator::cpu\_accelerator` Data Member](#): You can use this accelerator for setting up staging arrays. It cannot execute C++ AMP code. For more information, see the [Staging Arrays in C++ AMP](#) post on the Parallel Programming in Native Code blog.

## Interoperability

The C++ AMP runtime supports interoperability between the `accelerator_view` class and the `Direct3D ID3D11Device` interface. The `create_accelerator_view` method takes an `IUnknown` interface and returns an `accelerator_view` object. The `get_device` method takes an `accelerator_view` object and returns an `IUnknown` interface.

## See also

[C++ AMP \(C++ Accelerated Massive Parallelism\)](#)  
[Debugging GPU Code](#)

accelerator\_view Class

# Reference (C++ AMP)

3/4/2019 • 2 minutes to read • [Edit Online](#)

This section contains reference information for the C++ Accelerated Massive Parallelism (C++ AMP) runtime.

## NOTE

The C++ language standard reserves the use of identifiers that begin with an underscore (`_`) character for implementations such as libraries. Do not use names beginning with an underscore in your code. The behavior of code elements whose names follow this convention are not guaranteed and are subject to change in future releases. For these reasons, such code elements are omitted from this documentation.

## In This Section

### [Concurrency Namespace \(C++ AMP\)](#)

Provides classes and functions that enable the acceleration of C++ code on data parallel hardware.

### [Concurrency::direct3d Namespace](#)

Provides functions that support D3D interoperability. Enables seamless use of D3D resources for compute in AMP code and the use of resources created in AMP in D3D code, without creating redundant intermediate copies. You can use C++ AMP to incrementally accelerate the compute-intensive sections of your DirectX applications and use the D3D API on data produced from AMP computations.

### [Concurrency::fast\\_math Namespace](#)

Functions in the `fast_math` namespace are not C99-compliant. Only single-precision versions of each function are provided. These functions use the DirectX intrinsic functions, which are faster than the corresponding functions in the `precise_math` namespace and do not require extended double-precision support on the accelerator, but they are less accurate. There are two versions of each function for source-level compatibility with C99 code; both versions take and return single-precision values.

### [Concurrency::graphics Namespace](#)

Provides types and functions that are designed for graphics programming.

### [Concurrency::precise\\_math Namespace](#)

Functions in the `precise_math` namespace are C99 compliant. Both single-precision and double-precision versions of each function are included. These functions—this includes the single-precision functions—require extended double-precision support on the accelerator.

## Related Sections

### [C++ AMP \(C++ Accelerated Massive Parallelism\)](#)

C++ AMP accelerates the execution of your C++ code by taking advantage of the data-parallel hardware that's commonly present as a graphics processing unit (GPU) on a discrete graphics card.

# Concurrency Namespace (C++ AMP)

3/4/2019 • 5 minutes to read • [Edit Online](#)

Provides classes and functions that accelerate the execution of C++ code on data-parallel hardware. For more information, see [C++ AMP Overview](#)

## Syntax

```
namespace Concurrency;
```

## Members

### Namespaces

NAME	DESCRIPTION
<a href="#">Concurrency::direct3d Namespace</a>	Provides functions that support D3D interoperability. Enables seamless use of D3D resources for compute in AMP code and the use of resources created in AMP in D3D code, without creating redundant intermediate copies. You can use C++ AMP to incrementally accelerate the compute-intensive sections of your DirectX applications and use the D3D API on data produced from AMP computations.
<a href="#">Concurrency::fast_math Namespace</a>	Functions in the <code>fast_math</code> namespace are not C99-compliant. Only single-precision versions of each function are provided. These functions use the DirectX intrinsic functions, which are faster than the corresponding functions in the <code>precise_math</code> namespace and do not require extended double-precision support on the accelerator, but they are less accurate. There are two versions of each function for source-level compatibility with C99 code; both versions take and return single-precision values.
<a href="#">Concurrency::graphics Namespace</a>	Provides types and functions that are designed for graphics programming.
<a href="#">Concurrency::precise_math Namespace</a>	Functions in the <code>precise_math</code> namespace are C99 compliant. Both single-precision and double-precision versions of each function are included. These functions—this includes the single-precision functions—require extended double-precision support on the accelerator.

### Classes

NAME	DESCRIPTION
<a href="#">accelerator Class</a>	Represents an abstraction of a physical DP-optimized compute node.
<a href="#">accelerator_view Class</a>	Represents a virtual device abstraction on a C++ AMP data-parallel accelerator.

NAME	DESCRIPTION
<a href="#">accelerator_view_removed Class</a>	The exception that is thrown when an underlying DirectX call fails due to the Windows timeout-detection-and-recovery mechanism.
<a href="#">array Class</a>	A data aggregate on an <code>accelerator_view</code> in the grid domain. It is a collection of variables, one for each element in a grid domain. Each variable holds a value that corresponds to some C++ type.
<a href="#">array_view Class</a>	Represents a view into the data in an <code>array&lt;T,N&gt;</code> .
<a href="#">completion_future Class</a>	Represents a future that corresponds to a C++ AMP asynchronous operation.
<a href="#">extent Class</a>	Represents a vector of N integer values that specify the bounds of an N-dimensional space that has an origin of 0. The values in the coordinate vector are ordered from most significant to least significant. For example, in Cartesian 3-dimensional space, the extent vector (7,5,3) represents a space in which the z coordinate ranges from 0 to 7, the y coordinate ranges from 0 to 5, and the x coordinate ranges from 0 to 3.
<a href="#">index Class</a>	Defines an N-dimensional index point.
<a href="#">invalid_compute_domain Class</a>	The exception that's thrown when the runtime can't start a kernel by using the compute domain specified at the <code>parallel_for_each</code> call site.
<a href="#">out_of_memory Class</a>	The exception that is thrown when a method fails because of a lack of system or device memory.
<a href="#">runtime_exception Class</a>	The base type for exceptions in the C++ AMP library.
<a href="#">tile_barrier Class</a>	A capability class that is only creatable by the system and is passed to a tiled <code>parallel_for_each</code> lambda as part of the <code>tiled_index</code> parameter. It provides one method, <code>wait()</code> , whose purpose is to synchronize execution of threads that are running in the thread group (tile).
<a href="#">tiled_extent Class</a>	A <code>tiled_extent</code> object is an <code>extent</code> object of one to three dimensions that subdivides the extent space into one-dimensional, two-dimensional, or three-dimensional tiles.
<a href="#">tiled_index Class</a>	Provides an index into a <code>tiled_grid</code> object. This class has properties to access element relative to the local tile origin and relative to the global origin.
<a href="#">uninitialized_object Class</a>	The exception that is thrown when an uninitialized object is used.
<a href="#">unsupported_feature Class</a>	The exception that is thrown when an unsupported feature is used.

## Enumerations

NAME	DESCRIPTION
<a href="#">access_type Enumeration</a>	Specifies the data access type.
<a href="#">queuing_mode Enumeration</a>	Specifies the queuing modes that are supported on the accelerator.

## Operators

OPERATOR	DESCRIPTION
<a href="#">operator== Operator (C++ AMP)</a>	Determines whether the specified data structures are equal.
<a href="#">operator!= Operator (C++ AMP)</a>	Determines whether the specified data structures are unequal.
<a href="#">operator+ Operator (C++ AMP)</a>	Computes the component-wise sum of the specified arguments.
<a href="#">operator- Operator (C++ AMP)</a>	Computes the component-wise difference between the specified arguments.
<a href="#">operator* Operator (C++ AMP)</a>	Computes the component-wise product of the specified arguments.
<a href="#">operator/ Operator (C++ AMP)</a>	Computes the component-wise quotient of the specified arguments.
<a href="#">operator% Operator (C++ AMP)</a>	Computes the modulus of the first specified argument by the second specified argument.

## Functions

NAME	DESCRIPTION
<a href="#">all_memory_fence</a>	Blocks execution of all threads in a tile until all memory accesses have been completed.
<a href="#">amp_uninitialize</a>	Uninitializes the C++ AMP runtime.
<a href="#">atomic_compare_exchange</a>	Overloaded. If the value stored at the specified location compares equal to the first specified value, then the second specified value is stored in the same location as an atomic operation.
<a href="#">atomic_exchange</a>	Overloaded. Sets the value stored at the specified location to the specified value as an atomic operation.
<a href="#">atomic_fetch_add</a>	Overloaded. Sets the value stored at the specified location to the sum of that value and a specified value as an atomic operation.
<a href="#">atomic_fetch_and</a>	Overloaded. Sets the value stored at the specified location to the bitwise <code>and</code> of that value and a specified value as an atomic operation.

NAME	DESCRIPTION
<a href="#">atomic_fetch_dec</a>	Overloaded. Decrements the value stored at the specified location and stores the result in the same location as an atomic operation.
<a href="#">atomic_fetch_inc</a>	Overloaded. Increments the value stored at the specified location and stores the result in the same location as an atomic operation.
<a href="#">atomic_fetch_max</a>	Overloaded. Sets the value stored at the specified location to the larger of that value and a specified value as an atomic operation.
<a href="#">atomic_fetch_min</a>	Overloaded. Sets the value stored at the specified location to the smaller of that value and a specified value as an atomic operation.
<a href="#">atomic_fetch_or</a>	Overloaded. Sets the value stored at the specified location to the bitwise <code>or</code> of that value and a specified value as an atomic operation.
<a href="#">atomic_fetch_sub</a>	Overloaded. Sets the value stored at the specified location to the difference of that value and a specified value as an atomic operation.
<a href="#">atomic_fetch_xor</a>	Overloaded. Sets the value stored at the specified location to the bitwise <code>xor</code> of that value and a specified value as an atomic operation.
<a href="#">copy</a>	Copies a C++ AMP object. All synchronous data transfer requirements are met. Data can't be copied when code is running on an accelerator. The general form of this function is <code>copy(src, dest)</code> .
<a href="#">copy_async</a>	Copies a C++ AMP object and returns <a href="#">completion_future</a> that can be waited on. Data can't be copied when code is running on an accelerator. The general form of this function is <code>copy(src, dest)</code> .
<a href="#">direct3d_abort</a>	Aborts the execution of a function that has the <code>restrict(amp)</code> restriction clause.
<a href="#">direct3d_errorf</a>	Prints a formatted string to the Visual Studio <b>Output</b> window and raises a <a href="#">runtime_exception</a> exception that has the same formatting string.
<a href="#">direct3d_printf</a>	Prints a formatted string to the Visual Studio <b>Output</b> window. It is called from a function that has the <code>restrict(amp)</code> restriction clause.
<a href="#">global_memory_fence</a>	Blocks execution of all threads in a tile until all global memory accesses have been completed.
<a href="#">parallel_for_each Function (C++ AMP)</a>	Runs a function across the compute domain.

NAME	DESCRIPTION
<a href="#">tile_static_memory_fence</a>	Blocks execution of all threads in a tile until <code>tile_static</code> memory accesses have been completed.

## Constants

NAME	DESCRIPTION
<a href="#">HLSL_MAX_NUM_BUFFERS Constant</a>	The maximum number of buffers allowed by DirectX.
<a href="#">MODULENAME_MAX_LENGTH Constant</a>	Stores the maximum length of the module name. This value must be the same on the compiler and the runtime.

## Requirements

**Header:** `amp.h`

## See also

[Reference \(C++ AMP\)](#)

# Concurrency namespace functions (AMP)

3/4/2019 • 10 minutes to read • [Edit Online](#)

<a href="#">all_memory_fence</a>	<a href="#">amp_uninitialize</a>	<a href="#">atomic_compare_exchange</a>
<a href="#">atomic_exchange Function (C++ AMP)</a>	<a href="#">atomic_fetch_add Function (C++ AMP)</a>	<a href="#">atomic_fetch_and Function (C++ AMP)</a>
<a href="#">atomic_fetch_dec</a>	<a href="#">atomic_fetch_inc</a>	<a href="#">atomic_fetch_max</a>
<a href="#">atomic_fetch_min</a>	<a href="#">atomic_fetch_or Function (C++ AMP)</a>	<a href="#">atomic_fetch_sub Function (C++ AMP)</a>
<a href="#">atomic_fetch_xor Function (C++ AMP)</a>	<a href="#">copy</a>	<a href="#">copy_async</a>
<a href="#">direct3d_abort</a>	<a href="#">direct3d_errorf</a>	<a href="#">direct3d_printf</a>
<a href="#">global_memory_fence</a>	<a href="#">parallel_for_each Function (C++ AMP)</a>	<a href="#">tile_static_memory_fence</a>

## all\_memory\_fence

Blocks execution of all threads in a tile until all memory accesses have been completed. This ensures that all memory accesses are visible to other threads in the thread tile, and are executed in program order.

```
inline void all_memory_fence(const tile_barrier& _Barrier) restrict(amp);
```

### Parameters

*\_Barrier*

A `tile_barrier` object.

## amp\_uninitialize

Uninitializes the C++ AMP runtime. It is legal to call this function multiple times during an applications lifetime. Calling any C++ AMP API after calling this function will reinitialize the C++ AMP runtime. Note that it is illegal to use C++ AMP objects across calls to this function and doing so will result in undefined behavior. Also, concurrently calling this function and any other AMP APIs is illegal and would result in undefined behavior.

```
void __cdecl amp_uninitialize();
```

## atomic\_compare\_exchange

Atomically compares the value stored at a memory location specified in the first argument for equality with the value of the second specified argument, and if the values are the same, the value at the memory location is changed to that of the third specified argument.

```

inline bool atomic_compare_exchange(
    _Inout_int* _Dest,
    _Inout_int* _Expected_value,
    int value
) restrict(amp)

inline bool atomic_compare_exchange(
    _Inout_unsigned int* _Dest,
    _Inout_unsigned int* _Expected_value,
    unsigned int value
) restrict(amp)

```

### Parameters

*\_Dest*

The location from which one of the values to be compared is read, and to which the new value, if any, is to be stored.

*\_Expected\_value*

The location from which the second value to be compared is read.

*value*

The value to be stored to the memory location specified in by `_Dest` if `_Dest` is equal to `_Expected_value`.

### Return Value

**true** if the operation is successful; otherwise, **false**.

## atomic\_exchange Function (C++ AMP)

Sets the value of destination location as an atomic operation.

```

inline int atomic_exchange(
    _Inout_int* _Dest,
    int value
) restrict(amp)

inline unsigned int atomic_exchange(
    _Inout_unsigned int* _Dest,
    unsigned int value
) restrict(amp)

inline float atomic_exchange(
    _Inout_float* _Dest,
    float value
) restrict(amp)

```

### Parameters

*\_Dest*

Pointer to the destination location.

*value*

The new value.

### Return Value

The original value of the destination location.

## atomic\_fetch\_add Function (C++ AMP)

Atomically add a value to the value of a memory location.

```
inline int atomic_fetch_add(
    _Inout_int* _Dest,
    int value
) restrict(amp)

inline unsigned int atomic_fetch_add(
    _Inout_unsigned int* _Dest,
    unsigned int value
) restrict(amp)
```

### Parameters

*\_Dest*

Pointer to the memory location.

*value*

The value to be added.

### Return Value

The original value of the memory location.

## atomic\_fetch\_and Function (C++ AMP)

Atomically performs a bitwise AND operation of a value and the value of a memory location.

```
inline int atomic_fetch_and(
    _Inout_int* _Dest,
    int value
) restrict(amp)

inline unsigned int atomic_fetch_and(
    _Inout_unsigned int* _Dest,
    unsigned int value
) restrict(amp)
```

### Parameters

*\_Dest*

Pointer to the memory location.

*value*

The value to use in the bitwise AND calculation.

### Return Value

The original value of the memory location.

## atomic\_fetch\_dec

Atomically decrements the value stored at the specified memory location.

```
inline int atomic_fetch_dec(_Inout_int* _Dest
) restrict(amp)

inline unsigned int atomic_fetch_dec(_Inout_unsigned int* _Dest) restrict(amp);
```

### Parameters

*\_Dest*

The location in memory of the value to be decremented.

### Return Value

The original value stored at the memory location.

## atomic\_fetch\_inc

Atomically increments the value stored at the specified memory location.

```
inline int atomic_fetch_inc(_Inout_ int* _Dest) restrict(amp);

inline unsigned int atomic_fetch_inc(_Inout_ unsigned int* _Dest) restrict(amp);
```

### Parameters

*\_Dest*

The location in memory of the value to be incremented.

### Return Value

The original value stored at the memory location.

## atomic\_fetch\_max

Atomically computes the maximum value between the value stored at the memory location specified in the first argument and the value specified in the second argument, and stores it at the same memory location.

```
inline int atomic_fetch_max(
    _Inout_ int* _Dest,
    int value
) restrict(amp)

inline unsigned int atomic_fetch_max(
    _Inout_ unsigned int* _Dest,
    unsigned int value
) restrict(amp)
```

### Parameters

*\_Dest*

The location from which one of the values to be compared is read, and to which the maximum of the two values is to be stored.

*value*

The value to be compared to the value at the specified location.

### Return Value

The original value stored at the specified location location.

## atomic\_fetch\_min

Atomically computes the minimum value between the value stored at the memory location specified in the first argument and the value specified in the second argument, and stores it at the same memory location.

```
inline int atomic_fetch_min(
    _Inout_int* _Dest,
    int value
) restrict(amp)

inline unsigned int atomic_fetch_min(
    _Inout_unsigned int* _Dest,
    unsigned int value
) restrict(amp)
```

### Parameters

#### *\_Dest*

The location from which one of the values to be compared is read, and to which the minimum of the two values is to be stored.

#### *value*

The value to be compared to the value at the specified location.

### Return Value

The original value stored at the specified location location.

## atomic\_fetch\_or Function (C++ AMP)

Atomically performs a bitwise OR operation with a value and the value of a memory location.

```
inline int atomic_fetch_or(
    _Inout_int* _Dest,
    int value
) restrict(amp)

inline unsigned int atomic_fetch_or(
    _Inout_unsigned int* _Dest,
    unsigned int value
) restrict(amp)
```

### Parameters

#### *\_Dest*

Pointer to the memory location.

#### *value*

The value to use in the bitwise OR calculation.

### Return Value

The original value of the memory location.

## atomic\_fetch\_sub Function (C++ AMP)

Atomically subtracts a value from a memory location.

```
inline int atomic_fetch_sub(
    _Inout_int* _Dest,
    int value
) restrict(amp)

inline unsigned int atomic_fetch_sub(
    _Inout_unsigned int* _Dest,
    unsigned int value
) restrict(amp)
```

### Parameters

*\_Dest*

Pointer to the destination location.

*value*

The value to be subtracted.

### Return Value

The original value of the memory location.

## atomic\_fetch\_xor Function (C++ AMP)

Atomically performs an bitwise XOR operation of a value and a memory location.

```
inline int atomic_fetch_xor(
    _Inout_int* _Dest,
    int value
) restrict(amp)

inline unsigned int atomic_fetch_xor(
    _Inout_unsigned int* _Dest,
    unsigned int value
) restrict(amp)
```

### Parameters

*\_Dest*

Pointer to the memory location.

*value*

The value to use in the XOR calculation.

### Return Value

The original value of the memory location.

## copy

Copies a C++ AMP object. All synchronous data transfer requirements are met. You can't copy data when running code on an accelerator. The general form of this function is `copy(src, dest)`.

```

template <typename value_type, int _Rank>
void copy(
    const array<value_type, _Rank>& _Src,
    array<value_type, _Rank>& _Dest);

template <typename InputIterator, typename value_type, int _Rank>
void copy(
    InputIterator _SrcFirst,
    InputIterator _SrcLast,
    array<value_type, _Rank>& _Dest);

template <typename InputIterator, typename value_type, int _Rank>
void copy(
    InputIterator _SrcFirst,
    array<value_type, _Rank>& _Dest);

template <typename OutputIterator, typename value_type, int _Rank>
void copy(
    const array<value_type, _Rank>& _Src,
    OutputIterator _DestIter);

template <typename value_type, int _Rank>
void copy(
    const array<value_type, _Rank>& _Src,
    array_view<value_type, _Rank>& _Dest);

template <typename value_type, int _Rank>
void copy(
    const array_view<const value_type, _Rank>& _Src,
    array<value_type, _Rank>& _Dest);

template <typename value_type, int _Rank>
void copy(
    const array_view<value_type, _Rank>& _Src,
    array<value_type, _Rank>& _Dest);

template <typename value_type, int _Rank>
void copy(
    const array_view<const value_type, _Rank>& _Src,
    array_view<value_type, _Rank>& _Dest);

template <typename value_type, int _Rank>
void copy(
    const array_view<value_type, _Rank>& _Src,
    array_view<value_type, _Rank>& _Dest);

template <typename InputIterator, typename value_type, int _Rank>
void copy(
    InputIterator _SrcFirst,
    InputIterator _SrcLast,
    array_view<value_type, _Rank>& _Dest);

template <typename InputIterator, typename value_type, int _Rank>
void copy(
    InputIterator _SrcFirst,
    array_view<value_type, _Rank>& _Dest);

template <typename OutputIterator, typename value_type, int _Rank>
void copy(
    const array_view<value_type, _Rank>& _Src,
    OutputIterator _DestIter);

```

## Parameters

*\_Dest*

The object to copy to.

*\_DestIter*

An output iterator to the beginning position at destination.

*InputIterator*

The type of the input iterator.

*OutputIterator*

The type of the output iterator.

*\_Rank*

The rank of the object to copy from or the object to copy to.

*\_Src*

To object to copy.

*\_SrcFirst*

A beginning iterator into the source container.

*\_SrcLast*

An ending iterator into the source container.

*value\_type*

The data type of the elements that are copied.

## copy\_async

Copies a C++ AMP object and returns a [completion\\_future](#) object that can be waited on. You can't copy data when running code on an accelerator. The general form of this function is `copy(src, dest)`.

```

template <typename value_type, int _Rank>
concurrency::completion_future copy_async(
    const array<value_type, _Rank>& _Src,
    array<value_type, _Rank>& _Dest);

template <typename InputIterator, typename value_type, int _Rank>
concurrency::completion_future copy_async(InputIterator _SrcFirst, InputIterator _SrcLast,
    array<value_type, _Rank>& _Dest);

template <typename InputIterator, typename value_type, int _Rank>
concurrency::completion_future copy_async(InputIterator _SrcFirst,
    array<value_type, _Rank>& _Dest);

template <typename OutputIterator, typename value_type, int _Rank>
concurrency::completion_future copy_async(
    const array<value_type, _Rank>& _Src, OutputIterator _DestIter);

template <typename value_type, int _Rank>
concurrency::completion_future copy_async(
    const array<value_type, _Rank>& _Src,
    array_view<value_type, _Rank>& _Dest);

template <typename value_type, int _Rank>
concurrency::completion_future copy_async(
    const array_view<const value_type, _Rank>& _Src,
    array<value_type, _Rank>& _Dest);

template <typename value_type, int _Rank>
concurrency::completion_future copy_async(
    const array_view<value_type, _Rank>& _Src,
    array<value_type, _Rank>& _Dest);

template <typename value_type, int _Rank>
concurrency::completion_future copy_async(
    const array_view<const value_type, _Rank>& _Src,
    array_view<value_type, _Rank>& _Dest);

template <typename value_type, int _Rank>
concurrency::completion_future copy_async(
    const array_view<value_type, _Rank>& _Src,
    array_view<value_type, _Rank>& _Dest);

template <typename InputIterator, typename value_type, int _Rank>
concurrency::completion_future copy_async(InputIterator _SrcFirst, InputIterator _SrcLast,
    array_view<value_type, _Rank>& _Dest);

template <typename InputIterator, typename value_type, int _Rank>
concurrency::completion_future copy_async(InputIterator _SrcFirst,
    array_view<value_type, _Rank>& _Dest);

template <typename OutputIterator, typename value_type, int _Rank>
concurrency::completion_future copy_async(
    const array_view<value_type, _Rank>& _Src, OutputIterator _DestIter);

```

## Parameters

*\_Dest*

The object to copy to.

*\_DestIter*

An output iterator to the beginning position at destination.

*InputIterator*

The type of the input iterator.

*OutputIterator*

The type of the output iterator.

*\_Rank*

The rank of the object to copy from or the object to copy to.

*\_Src*

To object to copy.

*\_SrcFirst*

A beginning iterator into the source container.

*\_SrcLast*

An ending iterator into the source container.

*value\_type*

The data type of the elements that are copied.

### Return Value

A `future<void>` that can be waited on.

## direct3d\_abort

Aborts the execution of a function with the `restrict(amp)` restriction clause. When the AMP runtime detects the call, it raises a [runtime\\_exception](#) exception with the error message "Reference Rasterizer: Shader abort instruction hit".

```
void direct3d_abort() restrict(amp);
```

## direct3d\_errorf

Prints a formatted string to the Visual Studio output window. It is called from a function with the `restrict(amp)` restriction clause. When the AMP runtime detects the call, it raises a [runtime\\_exception](#) exception with the same formatting string.

```
void direct3d_errorf(  
    const char *,  
    ...) restrict(amp);
```

## direct3d\_printf

Prints a formatted string to the Visual Studio output window. It is called from a function with the `restrict(amp)` restriction clause.

```
void direct3d_printf(  
    const char *,  
    ...) restrict(amp);
```

## global\_memory\_fence

Blocks execution of all threads in a tile until all global memory accesses have been completed. This ensures that global memory accesses are visible to other threads in the thread tile, and are executed in program order.

```
inline void global_memory_fence(const tile_barrier& _Barrier) restrict(amp);
```

## Parameters

*\_Barrier*

A `tile_barrier` object

# parallel\_for\_each Function (C++ AMP)

Runs a function across the compute domain. For more information, see [C++ AMP Overview](#).

```
template <int _Rank, typename _Kernel_type>
void parallel_for_each(
    const extent<_Rank>& _Compute_domain,
    const _Kernel_type& _Kernel);

template <int _Dim0, int _Dim1, int _Dim2, typename _Kernel_type>
void parallel_for_each(
    const tiled_extent<_Dim0, _Dim1, _Dim2>& _Compute_domain,
    const _Kernel_type& _Kernel);

template <int _Dim0, int _Dim1, typename _Kernel_type>
void parallel_for_each(
    const tiled_extent<_Dim0, _Dim1>& _Compute_domain,
    const _Kernel_type& _Kernel);

template <int _Dim0, typename _Kernel_type>
void parallel_for_each(
    const tiled_extent<_Dim0>& _Compute_domain,
    const _Kernel_type& _Kernel);

template <int _Rank, typename _Kernel_type>
void parallel_for_each(
    const accelerator_view& _Accl_view,
    const extent<_Rank>& _Compute_domain,
    const _Kernel_type& _Kernel);

template <int _Dim0, int _Dim1, int _Dim2, typename _Kernel_type>
void parallel_for_each(
    const accelerator_view& _Accl_view,
    const tiled_extent<_Dim0, _Dim1, _Dim2>& _Compute_domain,
    const _Kernel_type& _Kernel);

template <int _Dim0, int _Dim1, typename _Kernel_type>
void parallel_for_each(
    const accelerator_view& _Accl_view,
    const tiled_extent<_Dim0, _Dim1>& _Compute_domain,
    const _Kernel_type& _Kernel);

template <int _Dim0, typename _Kernel_type>
void parallel_for_each(
    const accelerator_view& _Accl_view,
    const tiled_extent<_Dim0>& _Compute_domain,
    const _Kernel_type& _Kernel);
```

## Parameters

*\_Accl\_view*

The `accelerator_view` object to run the parallel computation on.

*\_Compute\_domain*

An `extent` object that contains the data for the computation.

*\_Dim0*

The dimension of the `tiled_extent` object.

*\_Dim1*

The dimension of the `tiled_extent` object.

*\_Dim2*

The dimension of the `tiled_extent` object.

*\_Kernel*

A lambda or function object that takes an argument of type "index<\_Rank>" and performs the parallel computation.

*\_Kernel\_type*

A lambda or functor.

*\_Rank*

The rank of the extent.

## tile\_static\_memory\_fence

Blocks execution of all threads in a tile until all outstanding `tile_static` memory accesses have been completed. This ensures that `tile_static` memory accesses are visible to other threads in the thread tile, and that accesses are executed in program order.

```
inline void tile_static_memory_fence(const tile_barrier& _Barrier) restrict(amp);
```

### Parameters

*\_Barrier*

A `tile_barrier` object.

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# Concurrency namespace enums (AMP)

3/4/2019 • 2 minutes to read • [Edit Online](#)

[access\\_type Enumeration](#)

[queuing\\_mode Enumeration](#)

## access\_type Enumeration

Enumeration type used to denote the various types of access to data.

```
enum access_type;
```

### Values

NAME	DESCRIPTION
<code>access_type_auto</code>	Automatically choose the best <code>access_type</code> for the accelerator.
<code>access_type_none</code>	Dedicated. The allocation is only accessible on the accelerator and not on the CPU.
<code>access_type_read</code>	Shared. The allocation is accessible on the accelerator and is readable on the CPU.
<code>access_type_read_write</code>	Shared. The allocation is accessible on the accelerator and is writable on the CPU.
<code>access_type_write</code>	Shared. The allocation is accessible on the accelerator and is both readable and writable on the CPU.

## queuing\_mode Enumeration

Specifies the queuing modes that are supported on the accelerator.

```
enum queuing_mode;
```

### Values

NAME	DESCRIPTION
<code>queuing_mode_immediate</code>	A queuing mode that specifies that any commands, for example, <a href="#">parallel_for_each Function (C++ AMP)</a> , are sent to the corresponding accelerator device as soon as they return to the caller.

NAME	DESCRIPTION
<code>queuing_mode_automatic</code>	A queuing mode that specifies that commands be queued up on a command queue that corresponds to the <a href="#">accelerator_view</a> object. Commands are sent to the device when <code>accelerator_view::flush</code> is called.

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# Concurrency namespace operators (AMP)

3/4/2019 • 2 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator%</code>	<code>operator*</code>
<code>operator+</code>	<code>operator-</code>	<code>operator/</code>
<code>operator==</code>		

## operator==

Determines whether the specified arguments are equal.

```
template <
    int _Rank,
    template <int> class _Tuple_type
>
bool operator== (
    const _Tuple_type<_Rank>& _Lhs,
    const _Tuple_type<_Rank>& _Rhs) restrict(amp);
```

### Parameters

*\_Rank*

The rank of the tuple arguments.

*\_Lhs*

One of the tuples to compare.

*\_Rhs*

One of the tuples to compare.

### Return Value

**true** if the tuples are equal; otherwise, **false**.

## operator!=

Determines whether the specified arguments are not equal.

```
template <
    int _Rank,
    template <int> class _Tuple_type
>
bool operator!= (
    const _Tuple_type<_Rank>& _Lhs,
    const _Tuple_type<_Rank>& _Rhs) restrict(amp);
```

### Parameters

*\_Rank*

The rank of the tuple arguments.

*\_Lhs*

One of the tuples to compare.

*\_Rhs*

One of the tuples to compare.

### Return Value

**true** if the tuples are not equal; otherwise, **false**.

## operator+

Computes the component-wise sum of the specified arguments.

```
template <
    int _Rank,
    template <int> class _Tuple_type
>
class _Tuple_type> _Tuple_type<_Rank> operator+(
    const _Tuple_type<_Rank>& _Lhs,
    const _Tuple_type<_Rank>& _Rhs) restrict(amp,cpu);

template <
    int _Rank,
    template <int> class _Tuple_type
>
class _Tuple_type> _Tuple_type<_Rank> operator+(
    const _Tuple_type<_Rank>& _Lhs,
    typename _Tuple_type<_Rank>::value_type _Rhs) restrict(amp,cpu);

template <
    int _Rank,
    template <int> class _Tuple_type
>
class _Tuple_type> _Tuple_type<_Rank> operator+(
    typename _Tuple_type<_Rank>::value_type _Lhs,
    const _Tuple_type<_Rank>& _Rhs) restrict(amp,cpu);
```

### Parameters

*\_Rank*

The rank of the tuple arguments.

*\_Lhs*

One of the arguments to add.

*\_Rhs*

One of the arguments to add.

### Return Value

The component-wise sum of the specified arguments.

## operator-

Computes the component-wise difference between the specified arguments.

```

template <
    int _Rank,
    template <int> class _Tuple_type
>
_Tuple_type<_Rank> operator-(
    const _Tuple_type<_Rank>& _Lhs,
    const _Tuple_type<_Rank>& _Rhs) restrict(amp,cpu);

template <
    int _Rank,
    template <int> class _Tuple_type
>
_Tuple_type<_Rank> operator-(
    const _Tuple_type<_Rank>& _Lhs,
    typename _Tuple_type<_Rank>::value_type _Rhs) restrict(amp,cpu);

template <
    int _Rank,
    template <int> class _Tuple_type
>
_Tuple_type<_Rank> operator-(
    typename _Tuple_type<_Rank>::value_type _Lhs,
    const _Tuple_type<_Rank>& _Rhs) restrict(amp,cpu);

```

### Parameters

*\_Rank*

The rank of the tuple arguments.

*\_Lhs*

The argument to be subtracted from.

*\_Rhs*

The argument to subtract.

### Return Value

The component-wise difference between the specified arguments.

## operator\*

Computes the component-wise product of the specified arguments.

```

template <
    int _Rank,
    template <int> class _Tuple_type
>
_Tuple_type<_Rank> operator*(
    const _Tuple_type<_Rank>& _Lhs,
    typename _Tuple_type<_Rank>::value_type _Rhs) restrict(amp,cpu);

template <
    int _Rank,
    template <int> class _Tuple_type
>
_Tuple_type<_Rank> operator*(
    typename _Tuple_type<_Rank>::value_type _Lhs,
    const _Tuple_type<_Rank>& _Rhs) restrict(amp,cpu);

```

### Parameters

*\_Rank*

The rank of the tuple arguments.

*\_Lhs*

One of the tuples to multiply.

*\_Rhs*

One of the tuples to multiply.

### **Return Value**

The component-wise product of the specified arguments.

## operator/

Computes the component-wise quotient of the specified arguments.

```
template <
    int _Rank,
    template <int> class _Tuple_type
>
_Tuple_type<_Rank> operator/(
    const _Tuple_type<_Rank>& _Lhs,
    typename _Tuple_type<_Rank>::value_type _Rhs) restrict(amp,cpu);

template <
    int _Rank,
    template <int> class _Tuple_type
>
_Tuple_type<_Rank> operator/(
    typename _Tuple_type<_Rank>::value_type _Lhs,
    const _Tuple_type<_Rank>& _Rhs) restrict(amp,cpu);
```

### **Parameters**

*\_Rank*

The rank of the tuple arguments.

*\_Lhs*

The tuple to be divided.

*\_Rhs*

The tuple to divide by.

### **Return Value**

The component-wise quotient of the specified arguments.

## operator%

Computes the modulus of the first specified argument by the second specified argument.

```
template <
    int _Rank,
    template <int> class _Tuple_type
>
_Tuple_type<_Rank> operator%(
    const _Tuple_type<_Rank>& _Lhs,
    typename _Tuple_type<_Rank>::value_type _Rhs) restrict(amp,cpu);

template <
    int _Rank,
    template <int> class _Tuple_type
>
_Tuple_type<_Rank> operator%(
    typename _Tuple_type<_Rank>::value_type _Lhs,
    const _Tuple_type<_Rank>& _Rhs) restrict(amp,cpu);
```

### Parameters

*\_Rank*

The rank of the tuple arguments.

*\_Lhs*

The tuple from which the modulo is calculated.

*\_Rhs*

The tuple to modulo by.

### Return Value

The result of the first specified argument modulus the second specified argument.

## See also

[Concurrency Namespace](#)

# Concurrency namespace constants (AMP)

3/4/2019 • 2 minutes to read • [Edit Online](#)

<a href="#">HLSL_MAX_NUM_BUFFERS</a>	<a href="#">MODULENAME_MAX_LENGTH</a>
--------------------------------------	---------------------------------------

## HLSL\_MAX\_NUM\_BUFFERS Constant

The maximum number of buffers allowed by DirectX.

```
static const UINT HLSL_MAX_NUM_BUFFERS = 64 + 128;
```

## MODULENAME\_MAX\_LENGTH Constant

Stores the maximum length of the module name. This value must be the same on the compiler and runtime.

```
static const UINT MODULENAME_MAX_LENGTH = 1024;
```

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# accelerator Class

3/4/2019 • 10 minutes to read • [Edit Online](#)

An accelerator is a hardware capability that is optimized for data-parallel computing. An accelerator may be a device attached to a PCIe bus (such as a GPU), or it might be an extended instruction set on the main CPU.

## Syntax

```
class accelerator;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">accelerator Constructor</a>	Initializes a new instance of the <code>accelerator</code> class.
<a href="#">~accelerator Destructor</a>	Destroys the <code>accelerator</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">create_view</a>	Creates and returns an <code>accelerator_view</code> object on this accelerator.
<a href="#">get_all</a>	Returns a vector of <code>accelerator</code> objects that represent all the available accelerators.
<a href="#">get_auto_selection_view</a>	Returns the auto-selection <code>accelerator_view</code> .
<a href="#">get_dedicated_memory</a>	Returns the dedicated memory for the <code>accelerator</code> , in kilobytes.
<a href="#">get_default_cpu_access_type</a>	Returns the default <code>access_type</code> for buffers created on this accelerator.
<a href="#">get_default_view</a>	Returns the default <code>accelerator_view</code> object that is associated with the <code>accelerator</code> .
<a href="#">get_description</a>	Returns a short description of the <code>accelerator</code> device.
<a href="#">get_device_path</a>	Returns the path of the device.
<a href="#">get_has_display</a>	Determines whether the <code>accelerator</code> is attached to a display.

NAME	DESCRIPTION
<a href="#">get_is_debug</a>	Determines whether the <code>accelerator</code> has the DEBUG layer enabled for extensive error reporting.
<a href="#">get_is_emulated</a>	Determines whether the <code>accelerator</code> is emulated.
<a href="#">get_supports_cpu_shared_memory</a>	Determines whether the <code>accelerator</code> supports shared memory
<a href="#">get_supports_double_precision</a>	Determines whether the <code>accelerator</code> is attached to a display.
<a href="#">get_supports_limited_double_precision</a>	Determines whether the <code>accelerator</code> has limited support for double-precision math.
<a href="#">get_version</a>	Returns the version of the <code>accelerator</code> .
<a href="#">set_default</a>	Returns the path of the default accelerator.
<a href="#">set_default_cpu_access_type</a>	Sets the default CPU <code>access_type</code> for arrays and implicit memory allocations made on this <code>accelerator</code> .

## Public Operators

NAME	DESCRIPTION
<a href="#">operator!=</a>	Compares this <code>accelerator</code> object with another and returns <b>false</b> if they are the same; otherwise, returns <b>true</b> .
<a href="#">operator=</a>	Copies the contents of the specified <code>accelerator</code> object to this one.
<a href="#">operator==</a>	Compares this <code>accelerator</code> object with another and returns <b>true</b> if they are the same; otherwise, returns <b>false</b> .

## Public Data Members

NAME	DESCRIPTION
<a href="#">cpu_accelerator</a>	Gets a string constant for the CPU <code>accelerator</code> .
<a href="#">dedicated_memory</a>	Gets the dedicated memory for the <code>accelerator</code> , in kilobytes.
<a href="#">default_accelerator</a>	Gets a string constant for the default <code>accelerator</code> .
<a href="#">default_cpu_access_type</a>	Gets or sets the default CPU <code>access_type</code> for arrays and implicit memory allocations made on this <code>accelerator</code> .
<a href="#">default_view</a>	Gets the default <code>accelerator_view</code> object that is associated with the <code>accelerator</code> .

NAME	DESCRIPTION
<a href="#">description</a>	Gets a short description of the <code>accelerator</code> device.
<a href="#">device_path</a>	Gets the path of the device.
<a href="#">direct3d_ref</a>	Gets a string constant for a Direct3D reference <code>accelerator</code> .
<a href="#">direct3d_warp</a>	Gets the string constant for an <code>accelerator</code> object that you can use for executing C++ AMP code on multi-core CPUs that use Streaming SIMD Extensions (SSE).
<a href="#">has_display</a>	Gets a Boolean value that indicates whether the <code>accelerator</code> is attached to a display.
<a href="#">is_debug</a>	Indicates whether the <code>accelerator</code> has the DEBUG layer enabled for extensive error reporting.
<a href="#">is_emulated</a>	Indicates whether the <code>accelerator</code> is emulated.
<a href="#">supports_cpu_shared_memory</a>	Indicates whether the <code>accelerator</code> supports shared memory.
<a href="#">supports_double_precision</a>	Indicates whether the accelerator supports double-precision math.
<a href="#">supports_limited_double_precision</a>	Indicates whether the accelerator has limited support for double-precision math.
<a href="#">version</a>	Gets the version of the <code>accelerator</code> .

## Inheritance Hierarchy

`accelerator`

## Remarks

An accelerator is a hardware capability that is optimized for data-parallel computing. An accelerator is often a discrete GPU, but it can also be a virtual host-side entity such as a DirectX REF device, a WARP (a CPU-side device that is accelerated by means of SSE instructions), or the CPU itself.

You can construct an `accelerator` object by enumerating the available devices, or by getting the default device, the reference device, or the WARP device.

## Requirements

**Header:** `amp.h`

**Namespace:** `Concurrency`

## ~accelerator

Destroys the `accelerator` object.

```
~accelerator();
```

### Return Value

## accelerator

Initializes a new instance of the [accelerator class](#).

```
accelerator();  
  
explicit accelerator(const std::wstring& _Device_path);  
  
accelerator(const accelerator& _Other);
```

### Parameters

*\_Device\_path*

The path of the physical device.

*\_Other*

The accelerator to copy.

## cpu\_accelerator

Gets a string constant for the CPU accelerator.

```
static const wchar_t cpu_accelerator[];
```

## create\_view

Creates and returns an `accelerator_view` object on this accelerator, using the specified queuing mode. When the queuing mode is not specified, the new `accelerator_view` uses the `queuing_mode::immediate` queuing mode.

```
accelerator_view create_view(queuing_mode qmode = queuing_mode_automatic);
```

### Parameters

*qmode*

The queuing mode.

### Return Value

A new `accelerator_view` object on this accelerator, using the specified queuing mode.

## dedicated\_memory

Gets the dedicated memory for the `accelerator`, in kilobytes.

```
__declspec(property(get= get_dedicated_memory)) size_t dedicated_memory;
```

## default\_accelerator

Gets a string constant for the default `accelerator`.

```
static const wchar_t default_accelerator[];
```

## default\_cpu\_access\_type

The default cpu [access\\_type](#) for arrays and implicit memory allocations made on this [accelerator](#).

```
__declspec(property(get= get_default_cpu_access_type)) access_type default_cpu_access_type;
```

## default\_view

Gets the default accelerator view that is associated with the [accelerator](#).

```
__declspec(property(get= get_default_view)) accelerator_view default_view;
```

## description

Gets a short description of the [accelerator](#) device.

```
__declspec(property(get= get_description)) std::wstring description;
```

## device\_path

Gets the path of the accelerator. The path is unique on the system.

```
__declspec(property(get= get_device_path)) std::wstring device_path;
```

## direct3d\_ref

Gets a string constant for a Direct3D reference accelerator.

```
static const wchar_t direct3d_ref[];
```

## direct3d\_warp

Gets the string constant for an [accelerator](#) object that you can use for executing your C++ AMP code on multi-core CPUs using Streaming SIMD Extensions (SSE).

```
static const wchar_t direct3d_warp[];
```

## get\_all

Returns a vector of [accelerator](#) objects that represent all the available accelerators.

```
static inline std::vector<accelerator> get_all();
```

### Return Value

The vector of available accelerators

## get\_auto\_selection\_view

Returns the auto selection `accelerator_view`, which when specified as the `parallel_for_each` target results in the target `accelerator_view` for executing the `parallel_for_each` kernel to be automatically selected by the runtime. For all other purposes, the `accelerator_view` returned by this method is the same as the default `accelerator_view` of the default accelerator

```
static accelerator_view __cdecl get_auto_selection_view();
```

### Return Value

The auto selection `accelerator_view`.

## get\_dedicated\_memory

Returns the dedicated memory for the `accelerator`, in kilobytes.

```
size_t get_dedicated_memory() const;
```

### Return Value

The dedicated memory for the `accelerator`, in kilobytes.

## get\_default\_cpu\_access\_type

Gets the default `cpu_access_type` for buffers created on this accelerator

```
access_type get_default_cpu_access_type() const;
```

### Return Value

The default `cpu_access_type` for buffers created on this accelerator.

## get\_default\_view

Returns the default `accelerator_view` object that is associated with the `accelerator`.

```
accelerator_view get_default_view() const;
```

### Return Value

The default `accelerator_view` object that is associated with the `accelerator`.

## get\_description

Returns a short description of the `accelerator` device.

```
std::wstring get_description() const;
```

### Return Value

A short description of the `accelerator` device.

## get\_device\_path

Returns the path of the accelerator. The path is unique on the system.

```
std::wstring get_device_path() const;
```

### Return Value

The system-wide unique device instance path.

## get\_has\_display

Returns a Boolean value that indicates whether the `accelerator` can output to a display.

```
bool get_has_display() const;
```

### Return Value

**true** if the `accelerator` can output to a display; otherwise, **false**.

## get\_is\_debug

Determines whether the `accelerator` has the DEBUG layer enabled for extensive error reporting.

```
bool get_is_debug() const;
```

### Return Value

**true** if the `accelerator` has the DEBUG layer enabled for extensive error reporting. Otherwise, **false**.

## get\_is\_emulated

Determines whether the `accelerator` is emulated.

```
bool get_is_emulated() const;
```

### Return Value

**true** if the `accelerator` is emulated. Otherwise, **false**.

## get\_supports\_cpu\_shared\_memory

Returns a boolean value indicating whether the accelerator supports memory accessible both by the accelerator and the CPU.

```
bool get_supports_cpu_shared_memory() const;
```

### Return Value

**true** if the accelerator supports CPU shared memory; otherwise, **false**.

## get\_supports\_double\_precision

Returns a Boolean value that indicates whether the accelerator supports double precision math, including fused

multiply add (FMA), division, reciprocal, and casting between **int** and **double**

```
bool get_supports_double_precision() const;
```

#### Return Value

**true** if the accelerator supports double precision math; otherwise, **false**.

## get\_supports\_limited\_double\_precision

Returns a Boolean value that indicates whether the accelerator has limited support for double precision math. If the accelerator has only limited support, then fused multiply add (FMA), division, reciprocal, and casting between **int** and **double** are not supported.

```
bool get_supports_limited_double_precision() const;
```

#### Return Value

**true** if the accelerator has limited support for double precision math; otherwise, **false**.

## get\_version

Returns the version of the `accelerator`.

```
unsigned int get_version() const;
```

#### Return Value

The version of the `accelerator`.

## has\_display

Gets a Boolean value that indicates whether the `accelerator` can output to a display.

```
__declspec(property(get= get_has_display)) bool has_display;
```

## is\_debug

Gets a Boolean value that indicates whether the `accelerator` has the DEBUG layer enabled for extensive error reporting.

```
__declspec(property(get= get_is_debug)) bool is_debug;
```

## is\_emulated

Gets a Boolean value that indicates whether the `accelerator` is emulated.

```
__declspec(property(get= get_is_emulated)) bool is_emulated;
```

## operator!=

Compares this `accelerator` object with another and returns **false** if they are the same; otherwise, returns **true**.

```
bool operator!= (const accelerator& _Other) const;
```

#### Parameters

*\_Other*

The `accelerator` object to compare with this one.

#### Return Value

**false** if the two `accelerator` objects are the same; otherwise, **true**.

## operator=

Copies the contents of the specified `accelerator` object to this one.

```
accelerator& operator= (const accelerator& _Other);
```

#### Parameters

*\_Other*

The `accelerator` object to copy from.

#### Return Value

A reference to this `accelerator` object.

## operator==

Compares this `accelerator` object with another and returns **true** if they are the same; otherwise, returns **false**.

```
bool operator== (const accelerator& _Other) const;
```

#### Parameters

*\_Other*

The `accelerator` object to compare with this one.

#### Return Value

**true** if the other `accelerator` object is same as this `accelerator` object; otherwise, **false**.

## set\_default

Sets the default accelerator to be used for any operation that implicitly uses the default accelerator. This method only succeeds if the runtime selected default accelerator has not already been used in an operation that implicitly uses the default accelerator

```
static inline bool set_default(std::wstring _Path);
```

#### Parameters

*\_Path*

The path to the accelerator.

#### Return Value

**true** if the call succeeds at setting the default accelerator. Otherwise, **false**.

## set\_default\_cpu\_access\_type

Set the default cpu access\_type for arrays created on this accelerator or for implicit memory allocations as part of array\_views accessed on this accelerator. This method only succeeds if the default\_cpu\_access\_type for the accelerator has not already been overridden by a previous call to this method and the runtime selected default\_cpu\_access\_type for this accelerator has not yet been used for allocating an array or for an implicit memory allocation backing an array\_view accessed on this accelerator.

```
bool set_default_cpu_access_type(access_type _Default_cpu_access_type);
```

### Parameters

*\_Default\_cpu\_access\_type*

The default cpu access\_type to be used for array/array\_view memory allocations on this accelerator.

### Return Value

A boolean value indicating if the default cpu access\_type for the accelerator was successfully set.

## supports\_cpu\_shared\_memory

Gets a Boolean value indicating whether the `accelerator` supports shared memory.

```
__declspec(property(get= get_supports_cpu_shared_memory)) bool supports_cpu_shared_memory;
```

## supports\_double\_precision

Gets a Boolean value that indicates whether the accelerator supports double precision math.

```
__declspec(property(get= get_supports_double_precision)) bool supports_double_precision;
```

## supports\_limited\_double\_precision

Gets a Boolean value that indicates whether the accelerator has limited support for double precision math. If the accelerator has only limited support, then fused multiply add (FMA), division, reciprocal, and casting between `int` and `double` are not supported.

```
__declspec(property(get= get_supports_limited_double_precision)) bool supports_limited_double_precision;
```

## version

Gets the version of the `accelerator`.

```
__declspec(property(get= get_version)) unsigned int version;
```

## ~accelerator\_view

Destroys the `accelerator_view` object.

```
~accelerator_view();
```

### Return Value

## accelerator

Gets the `accelerator` object for the `accelerator_view` object.

```
__declspec(property(get= get_accelerator)) Concurrency::accelerator accelerator;
```

## accelerator\_view

Initializes a new instance of the `accelerator_view` class by copying an existing `accelerator_view` object.

```
accelerator_view(const accelerator_view& _Other);
```

### Parameters

*\_Other*

The `accelerator_view` object to copy.

## create\_marker

Returns a future to track the completion of all commands submitted so far to this `accelerator_view` object.

```
concurrency::completion_future create_marker();
```

### Return Value

A future to track the completion of all commands submitted so far to this `accelerator_view` object.

## flush

Submits all pending commands queued to the `accelerator_view` object to the accelerator for execution.

```
void flush();
```

### Return Value

Returns `void`.

## get\_accelerator

Returns the `accelerator` object for the `accelerator_view` object.

```
accelerator get_accelerator() const;
```

### Return Value

The `accelerator` object for the `accelerator_view` object.

## get\_is\_auto\_selection

Returns a Boolean value that indicates whether the runtime will automatically select an appropriate accelerator when the `accelerator_view` is passed to a [parallel\\_for\\_each](#).

```
bool get_is_auto_selection() const;
```

#### Return Value

**true** if the runtime will automatically select an appropriate accelerator; otherwise, **false**.

## get\_is\_debug

Returns a Boolean value that indicates whether the `accelerator_view` object has the DEBUG layer enabled for extensive error reporting.

```
bool get_is_debug() const;
```

#### Return Value

A Boolean value that indicates whether the `accelerator_view` object has the DEBUG layer enabled for extensive error reporting.

## get\_queuing\_mode

Returns the queuing mode for the `accelerator_view` object.

```
queuing_mode get_queuing_mode() const;
```

#### Return Value

The queuing mode for the `accelerator_view` object.

## get\_version

Returns the version of the `accelerator_view`.

```
unsigned int get_version() const;
```

#### Return Value

The version of the `accelerator_view`.

## is\_auto\_selection

Gets a Boolean value that indicates whether the runtime will automatically select an appropriate accelerator when the `accelerator_view` is passed to a [parallel\\_for\\_each](#).

```
__declspec(property(get= get_is_auto_selection)) bool is_auto_selection;
```

## is\_debug

Gets a Boolean value that indicates whether the `accelerator_view` object has the DEBUG layer enabled for extensive error reporting.

```
__declspec(property(get= get_is_debug)) bool is_debug;
```

## operator!=

Compares this [accelerator\\_view](#) object with another and returns `false` if they are the same; otherwise, returns `true`.

```
bool operator!= (const accelerator_view& _Other) const;
```

### Parameters

*\_Other*

The `accelerator_view` object to compare with this one.

### Return Value

**false** if the two objects are the same; otherwise, **true**.

## operator=

Copies the contents of the specified [accelerator\\_view](#) object into this one.

```
accelerator_view& operator= (const accelerator_view& _Other);
```

### Parameters

*\_Other*

The `accelerator_view` object to copy from.

### Return Value

A reference to the modified `accelerator_view` object.

## operator==

Compares this [accelerator\\_view](#) object with another and returns **true** if they are the same; otherwise, returns **false**.

```
bool operator== (const accelerator_view& _Other) const;
```

### Parameters

*\_Other*

The `accelerator_view` object to compare with this one.

### Return Value

**true** if the two objects are the same; otherwise, **false**.

## queuing\_mode

Gets the queuing mode for the [accelerator\\_view](#) object.

```
__declspec(property(get= get_queuing_mode)) Concurrency::queuing_mode queuing_mode;
```

## version

Gets the version of the [accelerator\\_view](#).

```
__declspec(property(get= get_version)) unsigned int version;
```

## wait

Waits for all commands submitted to the [accelerator\\_view](#) object to finish.

```
void wait();
```

### Return Value

Returns `void`.

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# accelerator\_view Class

5/10/2019 • 4 minutes to read • [Edit Online](#)

Represents a virtual device abstraction on a C++ AMP data-parallel accelerator.

## Syntax

```
class accelerator_view;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">accelerator_view Constructor</a>	Initializes a new instance of the <code>accelerator_view</code> class.
<a href="#">~accelerator_view Destructor</a>	Destroys the <code>accelerator_view</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">create_marker</a>	Returns a future to track the completion of all commands submitted so far to this <code>accelerator_view</code> object.
<a href="#">flush</a>	Submits all pending commands queued to the <code>accelerator_view</code> object to the accelerator for execution.
<a href="#">get_accelerator</a>	Returns the <code>accelerator</code> object for the <code>accelerator_view</code> object.
<a href="#">get_is_auto_selection</a>	Returns a Boolean value that indicates whether the runtime will automatically select an appropriate accelerator when the <code>accelerator_view</code> object is passed to a <a href="#">parallel_for_each</a> .
<a href="#">get_is_debug</a>	Returns a Boolean value that indicates whether the <code>accelerator_view</code> object has the DEBUG layer enabled for extensive error reporting.
<a href="#">get_queuing_mode</a>	Returns the queuing mode for the <code>accelerator_view</code> object.
<a href="#">get_version</a>	Returns the version of the <code>accelerator_view</code> .
<a href="#">wait</a>	Waits for all commands submitted to the <code>accelerator_view</code> object to finish.

### Public Operators

NAME	DESCRIPTION
<code>operator!=</code>	Compares this <code>accelerator_view</code> object with another and returns <b>false</b> if they are the same; otherwise, returns <b>true</b> .
<code>operator=</code>	Copies the contents of the specified <code>accelerator_view</code> object into this one.
<code>operator==</code>	Compares this <code>accelerator_view</code> object with another and returns <b>true</b> if they are the same; otherwise, returns <b>false</b> .

### Public Data Members

NAME	DESCRIPTION
<code>accelerator</code>	Gets the <code>accelerator</code> object for the <code>accelerator_view</code> object.
<code>is_auto_selection</code>	Gets a Boolean value that indicates whether the runtime will automatically select an appropriate accelerator when the <code>accelerator_view</code> object is passed to a <code>parallel_for_each</code> .
<code>is_debug</code>	Gets a Boolean value that indicates whether the <code>accelerator_view</code> object has the DEBUG layer enabled for extensive error reporting.
<code>queuing_mode</code>	Gets the queuing mode for the <code>accelerator_view</code> object.
<code>version</code>	Gets the version of the accelerator.

## Inheritance Hierarchy

`accelerator_view`

### Remarks

An `accelerator_view` object represents a logical, isolated view of an accelerator. A single physical compute device can have many logical, isolated `accelerator_view` objects. Each accelerator has a default `accelerator_view` object. Additional `accelerator_view` objects can be created.

Physical devices can be shared among many client threads. Client threads can cooperatively use the same `accelerator_view` object of an accelerator, or each client can communicate with a compute device via an independent `accelerator_view` object for isolation from other client threads.

An `accelerator_view` object can have one of two `queuing_mode Enumeration` states. If the queuing mode is `immediate`, commands like `copy` and `parallel_for_each` are sent to the corresponding accelerator device as soon as they return to the caller. If the queuing mode is `deferred`, such commands are queued up on a command queue that corresponds to the `accelerator_view` object. Commands are not actually sent to the device until `flush()` is called.

## Requirements

**Header:** amprth

**Namespace:** Concurrency

## accelerator

Gets the accelerator object for the `accelerator_view` object.

### Syntax

```
__declspec(property(get= get_accelerator)) Concurrency::accelerator accelerator;
```

## accelerator\_view

Initializes a new instance of the `accelerator_view` class by copying an existing `accelerator_view` object.

### Syntax

```
accelerator_view( const accelerator_view & other );
```

### Parameters

*other*

The `accelerator_view` object to copy.

## create\_marker

Returns a future to track the completion of all commands submitted so far to this `accelerator_view` object.

### Syntax

```
concurrency::completion_future create_marker();
```

### Return Value

A future to track the completion of all commands submitted so far to this `accelerator_view` object.

## flush

Submits all pending commands queued to the `accelerator_view` object to the accelerator for execution.

### Syntax

```
void flush();
```

### Return Value

Returns `void`.

## get\_accelerator

Returns the accelerator object for the `accelerator_view` object.

### Syntax

```
accelerator get_accelerator() const;
```

### Return Value

The accelerator object for the `accelerator_view` object.

## get\_is\_auto\_selection

Returns a Boolean value that indicates whether the runtime will automatically select an appropriate accelerator when the `accelerator_view` is passed to a [parallel\\_for\\_each](#).

### Syntax

```
bool get_is_auto_selection() const;
```

### Return Value

**true** if the runtime will automatically select an appropriate accelerator; otherwise, **false**.

## get\_is\_debug

Returns a Boolean value that indicates whether the `accelerator_view` object has the DEBUG layer enabled for extensive error reporting.

### Syntax

```
bool get_is_debug() const;
```

### Return Value

A Boolean value that indicates whether the `accelerator_view` object has the DEBUG layer enabled for extensive error reporting.

## get\_queuing\_mode

Returns the queuing mode for the `accelerator_view` object.

### Syntax

```
queuing_mode get_queuing_mode() const;
```

### Return Value

The queuing mode for the `accelerator_view` object.

## get\_version

Returns the version of the `accelerator_view`.

### Syntax

```
unsigned int get_version() const;
```

### Return Value

The version of the `accelerator_view`.

## is\_auto\_selection

Gets a Boolean value that indicates whether the runtime will automatically select an appropriate accelerator when the `accelerator_view` is passed to a [parallel\\_for\\_each](#).

### Syntax

```
__declspec(property(get= get_is_auto_selection)) bool is_auto_selection;
```

## is\_debug

Gets a Boolean value that indicates whether the `accelerator_view` object has the DEBUG layer enabled for extensive error reporting.

### Syntax

```
__declspec(property(get= get_is_debug)) bool is_debug;
```

## operator!=

Compares this `accelerator_view` object with another and returns **false** if they are the same; otherwise, returns **true**.

### Syntax

```
bool operator!= ( const accelerator_view & other ) const;
```

### Parameters

*other*

The `accelerator_view` object to compare with this one.

### Return Value

**false** if the two objects are the same; otherwise, **true**.

## operator=

Copies the contents of the specified `accelerator_view` object into this one.

### Syntax

```
accelerator_view & operator= ( const accelerator_view & other );
```

### Parameters

*other*

The `accelerator_view` object to copy from.

### Return Value

A reference to the modified `accelerator_view` object.

## operator==

Compares this `accelerator_view` object with another and returns **true** if they are the same; otherwise, returns **false**.

## Syntax

```
bool operator== ( const accelerator_view & other ) const;
```

## Parameters

*other*

The `accelerator_view` object to compare with this one.

## Return Value

**true** if the two objects are the same; otherwise, **false**.

## queuing\_mode

Gets the queuing mode for the `accelerator_view` object.

## Syntax

```
__declspec(property(get= get_queuing_mode)) Concurrency::queuing_mode queuing_mode;
```

## version

Gets the version of the `accelerator_view`.

## Syntax

```
__declspec(property(get= get_version)) unsigned int version;
```

## wait

Waits for all commands submitted to the `accelerator_view` object to finish.

## Syntax

```
void wait();
```

## Return Value

Returns `void`.

## Remarks

If the `queuing_mode` is `immediate`, this method returns immediately without blocking.

## ~accelerator\_view

Destroys the `accelerator_view` object.

## Syntax

```
~accelerator_view();
```

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# accelerator\_view\_removed Class

5/10/2019 • 2 minutes to read • [Edit Online](#)

The exception that is thrown when an underlying DirectX call fails due to the Windows timeout detection and recovery mechanism.

## Syntax

```
class accelerator_view_removed : public runtime_exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">accelerator_view_removed Constructor</a>	Initializes a new instance of the <code>accelerator_view_removed</code> class.

### Public Methods

NAME	DESCRIPTION
<a href="#">get_view_removed_reason</a>	Returns an HRESULT error code indicating the cause of the <code>accelerator_view</code> object's removal.

## Inheritance Hierarchy

`exception`

`runtime_exception`

`out_of_memory`

## Requirements

**Header:** ampr.h

**Namespace:** Concurrency

## accelerator\_view\_removed

Initializes a new instance of the `accelerator_view_removed` class.

### Syntax

```
explicit accelerator_view_removed(
    const char * message,
    HRESULT view_removed_reason ) throw();

explicit accelerator_view_removed(
    HRESULT view_removed_reason ) throw();
```

### Parameters

*message*

A description of the error.

*view\_removed\_reason*

An HRESULT error code indicating the cause of removal of the `accelerator_view` object.

### Return Value

A new instance of the `accelerator_view_removed` class.

## get\_view\_removed\_reason

Returns an HRESULT error code indicating the cause of the `accelerator_view` object's removal.

### Syntax

```
HRESULT get_view_removed_reason() const throw();
```

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# array Class

3/4/2019 • 12 minutes to read • [Edit Online](#)

Represents a data container used to move data to an accelerator.

## Syntax

```
template <typename value_type, int _Rank>
friend class array;
```

### Parameters

*value\_type*

The element type of the data.

*\_Rank*

The rank of the array.

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">array Constructor</a>	Initializes a new instance of the <code>array</code> class.
<a href="#">~array Destructor</a>	Destroys the <code>array</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">copy_to</a>	Copies the contents of the array to another array.
<a href="#">data</a>	Returns a pointer to the raw data of the array.
<a href="#">get_accelerator_view</a>	Returns the <code>accelerator_view</code> object that represents the location where the array is allocated. This property can be accessed only on the CPU.
<a href="#">get_associated_accelerator_view</a>	Gets the second <code>accelerator_view</code> object that is passed as a parameter when a staging constructor is called to instantiate the <code>array</code> object.
<a href="#">get_cpu_access_type</a>	Returns the <code>access_type</code> of the array. This method can be accessed only on the CPU.
<a href="#">get_extent</a>	Returns the <code>extent</code> object of the array.
<a href="#">reinterpret_as</a>	Returns a one-dimensional array that contains all the elements in the <code>array</code> object.

NAME	DESCRIPTION
<a href="#">section</a>	Returns a subsection of the <code>array</code> object that is at the specified origin and, optionally, that has the specified extent.
<a href="#">view_as</a>	Returns an <code>array_view</code> object that is constructed from the <code>array</code> object.

## Public Operators

NAME	DESCRIPTION
<a href="#">operator std::vector&lt;value_type&gt;</a>	Uses <code>copy(*this, vector)</code> to implicitly convert the array to a <code>std::vector</code> object.
<a href="#">operator()</a>	Returns the element value that is specified by the parameters.
<a href="#">operator[]</a>	Returns the element that is at the specified index.
<a href="#">operator=</a>	Copies the contents of the specified <code>array</code> object into this one.

## Public Constants

NAME	DESCRIPTION
<a href="#">rank Constant</a>	Stores the rank of the array.

## Public Data Members

NAME	DESCRIPTION
<a href="#">accelerator_view</a>	Gets the <code>accelerator_view</code> object that represents the location where the array is allocated. This property can be accessed only on the CPU.
<a href="#">associated_accelerator_view</a>	Gets the second <code>accelerator_view</code> object that is passed as a parameter when a staging constructor is called to instantiate the <code>array</code> object.
<a href="#">cpu_access_type</a>	Gets the <code>access_type</code> that represents how the CPU can access the storage of the array.
<a href="#">extent</a>	Gets the extent that defines the shape of the array.

## Remarks

The type `array<T,N>` represents a dense and regular (not jagged)  $N$ -dimensional array that is located in a specific location, such as an accelerator or the CPU. The data type of the elements in the array is `T`, which must be of a type that is compatible with the target accelerator. Although the rank, `N`, (of the array is determined statically and is part of the type, the extent of the array is determined by the runtime and is expressed by using class `extent<N>`.

An array can have any number of dimensions, although some functionality is specialized for `array` objects with rank one, two, and three. If you omit the dimension argument, the default is 1.

Array data is laid out contiguously in memory. Elements that differ by one in the least significant dimension are adjacent in memory.

Arrays are logically considered to be value types, because when an array is copied to another array, a deep copy is performed. Two arrays never point to the same data.

The `array<T,N>` type is used in several scenarios:

- As a data container that can be used in computations on an accelerator.
- As a data container to hold memory on the host CPU (that can be used to copy to and from other arrays).
- As a staging object to act as a fast intermediary in host-to-device copies.

## Inheritance Hierarchy

`array`

## Requirements

**Header:** `amp.h`

**Namespace:** `Concurrency`

### `~array`

Destroys the `array` object.

```
~array() restrict(cpu);
```

## `accelerator_view`

Gets the `accelerator_view` object that represents the location where the array is allocated. This property can be accessed only on the CPU.

```
__declspec(property(get= get_accelerator_view)) Concurrency::accelerator_view accelerator_view;
```

## `array`

Initializes a new instance of the `array` class. There is no default constructor for `array<T,N>`. All constructors are run on the CPU only. They cannot be executed on a Direct3D target.

```
explicit array(
    const Concurrency::extent<_Rank>& _Extent) restrict(cpu);

explicit array(
    int _E0) restrict(cpu);

explicit array(
    int _E0,
    int _E1) restrict(cpu);

explicit array(
    int _E0,
    int _E1,
    int _E2) restrict(cpu);
```

```

array(
    const Concurrency::extent<_Rank>& _Extent,
    Concurrency::accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

array(
    int _E0,
    Concurrency::accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

array(
    int _E0,
    int _E1,
    Concurrency::accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

array(
    int _E0,
    int _E1,
    int _E2,
    Concurrency::accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

array(
    const Concurrency::extent<_Rank>& _Extent,
    Concurrency::accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

array(
    int _E0,
    accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

array(
    int _E0,
    int _E1,
    Concurrency::accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

array(
    int _E0,
    int _E1,
    int _E2,
    Concurrency::accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

template <typename _InputIterator>
array(
    const Concurrency::extent<_Rank>& _Extent,
    _InputIterator _Src_first,
    _InputIterator _Src_last) restrict(cpu);

template <typename _InputIterator>
array(
    const Concurrency::extent<_Rank>& _Extent,
    _InputIterator _Src_first) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    _InputIterator _Src_first,
    _InputIterator _Src_last) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    _InputIterator _Src_first) restrict(cpu);

template <typename _InputIterator>

```

```

array(
    int _E0,
    int _E1,
    _InputIterator _Src_first,
    _InputIterator _Src_last) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    int _E1,
    _InputIterator _Src_first) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    int _E1,
    int _E2,
    _InputIterator _Src_first,
    _InputIterator _Src_last) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    int _E1,
    int _E2,
    _InputIterator _Src_first) restrict(cpu);

template <typename _InputIterator>
array(
    const Concurrency::extent<_Rank>& _Extent,
    _InputIterator _Src_first,
    _InputIterator _Src_last,
    Concurrency::accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

template <typename _InputIterator>
array(
    const Concurrency::extent<_Rank>& _Extent,
    _InputIterator _Src_first,
    Concurrency::accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    _InputIterator _Src_first,
    _InputIterator _Src_last,
    Concurrency::accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    _InputIterator _Src_first,
    Concurrency::accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    int _E1,
    _InputIterator _Src_first,
    _InputIterator _Src_last,
    Concurrency::accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,

```

```

    int _E1,
    _InputIterator _Src_first,
    Concurrency::accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    int _E1,
    int _E2,
    _InputIterator _Src_first,
    _InputIterator _Src_last,
    Concurrency::accelerator_view _Av,
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    int _E1,
    int _E2,
    _InputIterator _Src_first,
    Concurrency::accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

template <typename _InputIterator>
array(
    const Concurrency::extent<_Rank>& _Extent,
    _InputIterator _Src_first,
    _InputIterator _Src_last,
    Concurrency::accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

template <typename _InputIterator>
array(
    const Concurrency::extent<_Rank>& _Extent,
    _InputIterator _Src_first,
    Concurrency::accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    _InputIterator _Src_first,
    _InputIterator _Src_last,
    Concurrency::accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0, _InputIterator _Src_first,
    Concurrency::accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    int _E1, _InputIterator _Src_first, _InputIterator _Src_last,
    Concurrency::accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    int _E1, _InputIterator _Src_first,
    Concurrency::accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

template <typename _InputIterator>
array(

```

```

    int _E0,
    int _E1,
    int _E2, _InputIterator _Src_first, _InputIterator _Src_last,
    Concurrency::accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

template <typename _InputIterator>
array(
    int _E0,
    int _E1,
    int _E2, _InputIterator _Src_first,
    Concurrency::accelerator_view _Av,
    Concurrency::accelerator_view _Associated_Av) restrict(cpu);

explicit array(
    const array_view<const value_type, _Rank>& _Src) restrict(cpu);

array(
    const array_view<const value_type, _Rank>& _Src,
    accelerator_view _Av
    access_type _Cpu_access_type = access_type_auto) restrict(cpu);

array(
    const array_view<const value_type, _Rank>& _Src,
    accelerator_view _Av,
    accelerator_view _Associated_Av) restrict(cpu);

array(const array& _Other) restrict(cpu);

array(array&& _Other) restrict(cpu);

```

## Parameters

*\_Associated\_Av*

An `accelerator_view` which specifies the preferred target location of the array.

*\_Av*

An [accelerator\\_view](#) object that specifies the location of the array.

*\_Cpu\_access\_type*

The desired [access\\_type](#) for the array on the CPU. This parameter has a default value of `access_type_auto` leaving the CPU `access_type` determination to the runtime. The actual CPU `access_type` for the array can be queried using the `get_cpu_access_type` method.

*\_Extent*

The extent in each dimension of the array.

*\_E0*

The most significant component of the extent of this section.

*\_E1*

The next-to-most-significant component of the extent of this section.

*\_E2*

The least significant component of the extent of this section.

*\_InputIterator*

The type of the input iterator.

*\_Src*

To object to copy.

*\_Src\_first*

A beginning iterator into the source container.

*\_Src\_last*

An ending iterator into the source container.

*\_Other*

Other data source.

*\_Rank*

The rank of the section.

*value\_type*

The data type of the elements that are copied.

## associated\_accelerator\_view

Gets the second [accelerator\\_view](#) object that is passed as a parameter when a staging constructor is called to instantiate the `array` object.

```
__declspec(property(get= get_associated_accelerator_view)) Concurrency::accelerator_view  
associated_accelerator_view;
```

## copy\_to

Copies the contents of the `array` to another `array`.

```
void copy_to(  
    array<value_type, _Rank>& _Dest) const ;  
  
void copy_to(  
    array_view<value_type, _Rank>& _Dest) const ;
```

### Parameters

*\_Dest*

The [array\\_view](#) object to copy to.

## cpu\_access\_type

Gets the CPU `access_type` allowed for this array.

```
__declspec(property(get= get_cpu_access_type)) access_type cpu_access_type;
```

## data

Returns a pointer to the raw data of the `array`.

```
value_type* data() restrict(amp, cpu);  
  
const value_type* data() const restrict(amp, cpu);
```

### Return Value

A pointer to the raw data of the array.

## extent

Gets the [extent](#) object that defines the shape of the `array`.

```
__declspec(property(get= get_extent)) Concurrency::extent<_Rank> extent;
```

## get\_accelerator\_view

Returns the [accelerator\\_view](#) object that represents the location where the `array` object is allocated. This property can be accessed only on the CPU.

```
Concurrency::accelerator_view get_accelerator_view() const;
```

### Return Value

The `accelerator_view` object that represents the location where the `array` object is allocated.

## get\_associated\_accelerator\_view

Gets the second [accelerator\\_view](#) object that is passed as a parameter when a staging constructor is called to instantiate the `array` object.

```
Concurrency::accelerator_view get_associated_accelerator_view() const ;
```

### Return Value

The second [accelerator\\_view](#) object passed to the staging constructor.

## get\_cpu\_access\_type

Returns the CPU `access_type` that's allowed for this array.

```
access_type get_cpu_access_type() const restrict(cpu);
```

### Return Value

## get\_extent

Returns the [extent](#) object of the `array`.

```
Concurrency::extent<_Rank> get_extent() const restrict(amp,cpu);
```

### Return Value

The `extent` object of the `array`.

## operator std::vector<value\_type>

Uses `copy(*this, vector)` to implicitly convert the array to a `std::vector` object.

```
operator std::vector<value_type>() const restrict(cpu);
```

## Parameters

*value\_type*

The data type of the elements of the vector.

## Return Value

An object of type `vector<T>` that contains a copy of the data that is contained in the array.

# operator()

Returns the element value that is specified by the parameters.

```
value_type& operator() (const index<_Rank>& _Index) restrict(amp,cpu);

const value_type& operator() (const index<_Rank>& _Index) const restrict(amp,cpu);

value_type& operator() (int _I0, int _I1) restrict(amp,cpu);

const value_type& operator() (int _I0, int _I1) const restrict(amp,cpu) ;

value_type& operator() (int _I0, int _I1, int _I2) restrict(amp,cpu);

const value_type& operator() (int _I0, int _I1, int _I2) const restrict(amp,cpu);

typename details::_Projection_result_type<value_type,_Rank>::_Result_type operator()(int _I)
restrict(amp,cpu);

typename details::_Projection_result_type<value_type,_Rank>::_Const_result_type operator()(int _I) const
restrict(amp,cpu);
```

## Parameters

*\_Index*

The location of the element.

*\_I0*

The most significant component of the origin of this section.

*\_I1*

The next-to-most-significant component of the origin of this section.

*\_I2*

The least significant component of the origin of this section.

*\_I*

The location of the element.

## Return Value

The element value specified by the parameters.

# operator[]

Returns the element that is at the specified index.

```

value_type& operator[](const index<_Rank>& _Index) restrict(amp,cpu);

const value_type& operator[]
    (const index<_Rank>& _Index) const restrict(amp,cpu);

typename details::_Projection_result_type<value_type,_Rank>::_Result_type operator[](int _i)
restrict(amp,cpu);

typename details::_Projection_result_type<value_type,_Rank>::_Const_result_type operator[](int _i) const
restrict(amp,cpu);

```

### Parameters

*\_Index*

The index.

*\_I*

The index.

### Return Value

The element that is at the specified index.

## operator=

Copies the contents of the specified `array` object.

```

array& operator= (const array& _Other) restrict(cpu);

array& operator= (array&& _Other) restrict(cpu);

array& operator= (
    const array_view<const value_type, _Rank>& _Src) restrict(cpu);

```

### Parameters

*\_Other*

The `array` object to copy from.

*\_Src*

The `array` object to copy from.

### Return Value

A reference to this `array` object.

## rank

Stores the rank of the `array`.

```

static const int rank = _Rank;

```

## reinterpret\_as

Reinterprets the array through a one-dimensional `array_view`, which optionally may have a different value type than the source array.

### Syntax

```
template <typename _Value_type2>
array_view<_Value_type2,1> reinterpret_as() restrict(amp,cpu);

template <typename _Value_type2>
array_view<const _Value_type2, 1> reinterpret_as() const restrict(amp,cpu);
```

### Parameters

*\_Value\_type2*

The data type of the returned data.

### Return Value

An `array_view` or `const array_view` object that is based on the array, with the element type reinterpreted from `T` to `ElementType` and the rank reduced from `N` to 1.

### Remarks

Sometimes it is convenient to view a multi-dimensional array as if it is a linear, one-dimensional array, possibly with a different value type than the source array. You can use this method to achieve this. **Caution** Reinterpreting an array object by using a different value type is a potentially unsafe operation. We recommend that you use this functionality carefully.

The following code provides an example.

```
struct RGB { float r; float g; float b; };

array<RGB,3> a = ...;
array_view<float,1> v = a.reinterpret_as<float>();

assert(v.extent == 3*a.extent);
```

## section

Returns a subsection of the `array` object that is at the specified origin and, optionally, that has the specified extent.

```

array_view<value_type,_Rank> section(
    const Concurrency::index<_Rank>& _Section_origin,
    const Concurrency::extent<_Rank>& _Section_extent) restrict(amp,cpu);

array_view<const value_type,_Rank> section(
    const Concurrency::index<_Rank>& _Section_origin,
    const Concurrency::extent<_Rank>& _Section_extent) const restrict(amp,cpu);

array_view<value_type,_Rank> section(
    const Concurrency::extent<_Rank>& _Ext) restrict(amp,cpu);

array_view<const value_type,_Rank> section(
    const Concurrency::extent<_Rank>& _Ext) const restrict(amp,cpu);

array_view<value_type,_Rank> section(
    const index<_Rank>& _Idx) restrict(amp,cpu);

array_view<const value_type,_Rank> section(
    const index<_Rank>& _Idx) const restrict(amp,cpu);

array_view<value_type,1> section(
    int _I0,
    int _E0) restrict(amp,cpu);

array_view<const value_type,1> section(
    int _I0,
    int _E0) const restrict(amp,cpu);

array_view<value_type,2> section(
    int _I0,
    int _I1,
    int _E0,
    int _E1) restrict(amp,cpu);

array_view<const value_type,2> section(
    int _I0,
    int _I1,
    int _E0,
    int _E1) const restrict(amp,cpu);

array_view<value_type,3> section(
    int _I0,
    int _I1,
    int _I2,
    int _E0,
    int _E1,
    int _E2) restrict(amp,cpu);

array_view<const value_type,3> section(
    int _I0,
    int _I1,
    int _I2,
    int _E0,
    int _E1,
    int _E2) const restrict(amp,cpu);

```

## Parameters

*\_E0*

The most significant component of the extent of this section.

*\_E1*

The next-to-most-significant component of the extent of this section.

*\_E2*

The least significant component of the extent of this section.

*\_Ext*

The [extent](#) object that specifies the extent of the section. The origin is 0.

*\_Idx*

The [index](#) object that specifies the location of the origin. The subsection is the rest of the extent.

*\_I0*

The most significant component of the origin of this section.

*\_I1*

The next-to-most-significant component of the origin of this section.

*\_I2*

The least significant component of the origin of this section.

*\_Rank*

The rank of the section.

*\_Section\_extent*

The [extent](#) object that specifies the extent of the section.

*\_Section\_origin*

The [index](#) object that specifies the location of the origin.

*value\_type*

The data type of the elements that are copied.

### Return Value

Returns a subsection of the [array](#) object that is at the specified origin and, optionally, that has the specified extent. When only the [index](#) object is specified, the subsection contains all elements in the associated grid that have indexes that are larger than the indexes of the elements in the [index](#) object.

## view\_as

Reinterprets this array as an [array\\_view](#) of a different rank.

```
template <int _New_rank>
array_view<value_type, _New_rank> view_as(
    const Concurrency::extent<_New_rank>& _View_extent) restrict(amp,cpu);

template <int _New_rank>
array_view<const value_type, _New_rank> view_as(
    const Concurrency::extent<_New_rank>& _View_extent) const restrict(amp,cpu);
```

### Parameters

*\_New\_rank*

The rank of the [extent](#) object passed as a parameter.

*\_View\_extent*

The extent that is used to construct the new [array\\_view](#) object.

*value\_type*

The data type of the elements in both the original [array](#) object and the returned [array\\_view](#) object.

### Return Value

The [array\\_view](#) object that is constructed.

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# array\_view Class

3/4/2019 • 12 minutes to read • [Edit Online](#)

Represents an N-dimensional view over the data held in another container.

## Syntax

```
template <
    typename value_type,
    int _Rank = 1
>
class array_view : public _Array_view_base<_Rank, sizeof(value_type)/sizeof(int)>;

template <
    typename value_type,
    int _Rank
>
class array_view<const value_type, _Rank> : public _Array_view_base<_Rank, sizeof(value_type)/sizeof(int)>;
```

### Parameters

*value\_type*

The data type of the elements in the `array_view` object.

*\_Rank*

The rank of the `array_view` object.

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">array_view Constructor</a>	Initializes a new instance of the <code>array_view</code> class. There is no default constructor for <code>array&lt;T,N&gt;</code> . All constructors are restricted to run on the CPU only and cannot be executed on a Direct3D target.
<a href="#">~array_view Destructor</a>	Destroys the <code>array_view</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">copy_to</a>	Copies the contents of the <code>array_view</code> object to the specified destination by calling <code>copy(*this, dest)</code> .
<a href="#">data</a>	Returns a pointer to the raw data of the <code>array_view</code> .
<a href="#">discard_data</a>	Discards the current data underlying this view.
<a href="#">get_extent</a>	Returns the extent object of the <code>array_view</code> object.

NAME	DESCRIPTION
<a href="#">get_ref</a>	Returns a reference to the indexed element.
<a href="#">get_source_accelerator_view</a>	Returns the <a href="#">accelerator_view</a> where the data source of the <code>array_view</code> is located.
<a href="#">refresh</a>	Notifies the <code>array_view</code> object that its bound memory has been modified outside the <code>array_view</code> interface. A call to this method renders all cached information stale.
<a href="#">reinterpret_as</a>	Returns a one-dimensional array that contains all the elements in the <code>array_view</code> object.
<a href="#">section</a>	Returns a subsection of the <code>array_view</code> object that's at the specified origin and, optionally, that has the specified extent.
<a href="#">synchronize</a>	Synchronizes any modifications made to the <code>array_view</code> object back to its source data.
<a href="#">synchronize_async</a>	Asynchronously synchronizes any modifications made to the <code>array_view</code> object back to its source data.
<a href="#">synchronize_to</a>	Synchronizes any modifications made to the <code>array_view</code> object to the specified <a href="#">accelerator_view</a> .
<a href="#">synchronize_to_async</a>	Asynchronously synchronizes any modifications made to the <code>array_view</code> object to the specified <a href="#">accelerator_view</a> .
<a href="#">view_as</a>	Produces an <code>array_view</code> object of a different rank using this <code>array_view</code> object's data.

## Public Operators

NAME	DESCRIPTION
<a href="#">operator()</a>	Returns the value of the element that is specified by the parameter or parameters.
<a href="#">operator[]</a>	Returns the element that is specified by the parameters.
<a href="#">operator=</a>	Copies the contents of the specified <code>array_view</code> object into this one.

## Public Constants

NAME	DESCRIPTION
<a href="#">rank Constant</a>	Stores the rank of the <code>array_view</code> object.

## Data Members

NAME	DESCRIPTION
<code>extent</code>	Gets the <code>extent</code> object that defines the shape of the <code>array_view</code> object.
<code>source_accelerator_view</code>	Gets the <code>accelerator_view</code> where the data source of the <code>array_view</code> is located
<code>value_type</code>	The value type of the <code>array_view</code> and the bound array.

## Remarks

The `array_view` class represents a view into the data that is contained in an `array` object or a subsection of an `array` object.

You can access the `array_view` object where the source data is located (locally) or on a different accelerator or a coherence domain (remotely). When you access the object remotely, views are copied and cached as necessary. Except for the effects of automatic caching, `array_view` objects have a performance profile similar to that of `array` objects. There is a small performance penalty when you access the data through views.

There are three remote usage scenarios:

- A view to a system memory pointer is passed by means of a `parallel_for_each` call to an accelerator and accessed on the accelerator.
- A view to an array located on an accelerator is passed by means of a `parallel_for_each` call to another accelerator and is accessed there.
- A view to an array located on an accelerator is accessed on the CPU.

In any one of these scenarios, the referenced views are copied by the runtime to the remote location and, if modified by the calls to the `array_view` object, are copied back to the local location. The runtime might optimize the process of copying changes back, might copy only changed elements, or might copy unchanged portions also. Overlapping `array_view` objects on one data source are not guaranteed to maintain referential integrity in a remote location.

You must synchronize any multithreaded access to the same data source.

The runtime makes the following guarantees regarding the caching of data in `array_view` objects:

- All well-synchronized accesses to an `array` object and an `array_view` object on it in program order obey a serial happens-before relationship.
- All well-synchronized accesses to overlapping `array_view` objects on the same accelerator on a single `array` object are aliased through the `array` object. They induce a total occurs-before relationship which obeys program order. There is no caching. If the `array_view` objects are executing on different accelerators, the order of access is undefined, creating a race condition.

When you create an `array_view` object using a pointer in system memory, you must change the view `array_view` object only through the `array_view` pointer. Alternatively, you must call `refresh()` on one of the `array_view` objects that are attached to the system pointer, if the underlying native memory is changed directly, instead of through the `array_view` object.

Either action notifies the `array_view` object that the underlying native memory is changed and that any copies that are located on an accelerator are outdated. If you follow these guidelines, the pointer-based views are identical to those provided to views of data-parallel arrays.

# Inheritance Hierarchy

`_Array_view_shape`

`_Array_view_base`

`array_view`

## Requirements

**Header:** `amp.h`

**Namespace:** `Concurrency`

### ~array\_view

Destroys the `array_view` object.

```
~array_view() restrict(amp,cpu);
```

### array\_view

Initializes a new instance of the `array_view` class.

```
array_view(
    array<value_type, _Rank>& _Src) restrict(amp,cpu);

array_view(
    const array_view& _Other) restrict(amp,cpu);

explicit array_view(
    const Concurrency::extent<_Rank>& _Extent) restrict(cpu);

template <
    typename _Container
>
array_view(
    const Concurrency::extent<_Rank>& _Extent,
    _Container& _Src) restrict(cpu);

array_view(
    const Concurrency::extent<_Rank>& _Extent,
    value_type* _Src) restrict(amp,cpu);

explicit array_view(
    int _E0) restrict(cpu);

template <
    typename _Container
>
explicit array_view(
    _Container& _Src,
    typename std::enable_if<details::_Is_container<_Container>::type::value, void **>::type = 0)
    restrict(cpu);

template <
    typename _Container
>
explicit array_view(
    int _E0,
    _Container& _Src) restrict(cpu);
```

```

explicit array_view(
    int _E0,
    int _E1) __CPU_ONLY;

template <
    typename _Container
>
explicit array_view(
    int _E0,
    int _E1,
    _Container& _Src) restrict(cpu);

explicit array_view(
    int _E0,
    int _E1,
    int _E2) __CPU_ONLY;

template <
    typename _Container
>
explicit array_view(
    int _E0,
    int _E1,
    int _E2,
    _Container& _Src);

explicit array_view(
    int _E0,
    _In_ value_type* _Src)restrict(amp,cpu);

template <
    typename _Arr_type,
    int _Size
>
explicit array_view(
    _In_ _Arr_type (& _Src) [_Size]) restrict(amp,cpu);

explicit array_view(
    int _E0,
    int _E1,
    _In_ value_type* _Src)restrict(amp,cpu);

explicit array_view(
    int _E0,
    int _E1,
    int _E2,
    _In_ value_type* _Src)restrict(amp,cpu);

array_view(
    const array<value_type, _Rank>& _Src)restrict(amp,cpu);

array_view(
    const array_view<value_type, _Rank>& _Src)restrict(amp,cpu);

array_view(
    const array_view<const value_type, _Rank>& _Src)restrict(amp,cpu);

template <
    typename _Container
>
array_view(
    const Concurrency::extent<_Rank>& _Extent,
    const _Container& _Src) restrict(cpu);

template <
    typename _Container
>
explicit array_view(
    const _Container& _Src,

```

```

    typename std::enable_if<details::_Is_container<_Container>::type::value, void **>::type = 0)
restrict(cpu);

array_view(
    const Concurrency::extent<_Rank>& _Extent,
    const value_type* _Src)restrict(amp,cpu);

template <
    typename _Arr_type,
    int _Size
>
explicit array_view(
    const _In_ _Arr_type (& _Src) [_Size]) restrict(amp,cpu);

template <
    typename _Container
>
array_view(
    int _E0,
    const _Container& _Src);

template <
    typename _Container
>
array_view(
    int _E0,
    int _E1,
    const _Container& _Src);

template <
    typename _Container
>
array_view(
    int _E0,
    int _E1,
    int _E2,
    const _Container& _Src);

array_view(
    int _E0,
    const value_type* _Src)restrict(amp,cpu);

array_view(
    int _E0,
    int _E1,
    const value_type* _Src) restrict(amp,cpu);

array_view(
    int _E0,
    int _E1,
    int _E2,
    const value_type* _Src) restrict(amp,cpu);

```

## Parameters

### *\_Arr\_type*

The element type of a C-style array from which data is supplied.

### *\_Container*

A template argument that must specify a linear container that supports `data()` and `size()` members.

### *\_E0*

The most significant component of the extent of this section.

### *\_E1*

The next-to-most-significant component of the extent of this section.

*\_E2*

The least significant component of the extent of this section.

*\_Extent*

The extent in each dimension of this `array_view`.

*\_Other*

An object of type `array_view<T,N>` from which to initialize the new `array_view`.

*\_Size*

The size of a C-style array from which data is supplied.

*\_Src*

A pointer to the source data that will be copied into the new array.

## copy\_to

Copies the contents of the `array_view` object to the specified destination object by calling `copy(*this, dest)`.

```
void copy_to(
    array<value_type, _Rank>& _Dest) const;

void copy_to(
    array_view<value_type, _Rank>& _Dest) const;
```

### Parameters

*\_Dest*

The object to copy to.

## data

Returns a pointer to the raw data of the `array_view`.

```
value_type* data() const restrict(amp,
    cpu);

const value_type* data() const restrict(amp,
    cpu);
```

### Return Value

A pointer to the raw data of the `array_view`.

## discard\_data

Discards the current data underlying this view. This is an optimization hint to the runtime used to avoid copying the current contents of the view to a target `accelerator_view` that it is accessed on, and its use is recommended if the existing content is not needed. This method is a no-op when used in a `restrict(amp)` context

```
void discard_data() const restrict(cpu);
```

## extent

Gets the `extent` object that defines the shape of the `array_view` object.

```
__declspec(property(get= get_extent)) Concurrency::extent<_Rank> extent;
```

## get\_extent

Returns the `extent` object of the `array_view` object.

```
Concurrency::extent<_Rank> get_extent() const restrict(cpu, amp);
```

### Return Value

The `extent` object of the `array_view` object

## get\_ref

Get a reference to the element indexed by `_Index`. Unlike the other indexing operators for accessing the `array_view` on the CPU, this method does not implicitly synchronize this `array_view`'s contents to the CPU. After accessing the `array_view` on a remote location or performing a copy operation involving this `array_view` users are responsible to explicitly synchronize the `array_view` to the CPU before calling this method. Failure to do so results in undefined behavior.

```
value_type& get_ref(  
    const index<_Rank>& _Index) const restrict(amp, cpu);
```

### Parameters

`_Index`

The index.

### Return Value

Reference to the element indexed by `_Index`

## get\_source\_accelerator\_view

Returns the `accelerator_view` where the data source of the `array_view` is located. If the `array_view` does not have a data source, this API throws a `runtime_exception`

```
accelerator_view get_source_accelerator_view() const;
```

### Return Value

## operator()

Returns the value of the element that is specified by the parameter or parameters.

```

value_type& operator() (
    const index<_Rank>& _Index) const restrict(amp,cpu);

typename details::_Projection_result_type<value_type,_Rank>::_Result_type operator() (
    int _I) const restrict(amp,cpu);

value_type& operator() (
    int _I0,
    int _I1) const restrict(amp,cpu);

value_type& operator() (
    int _I0,
    int _I1,
    int _I2) const restrict(amp,cpu);

typename details::_Projection_result_type<value_type,_Rank>::_Const_result_type operator() (
    int _I) const restrict(amp,cpu);

```

### Parameters

*\_Index*

The location of the element.

*\_I0*

The index in the first dimension.

*\_I1*

The index in the second dimension.

*\_I2*

The index in the third dimension.

*\_I*

The location of the element.

### Return Value

The value of the element that is specified by the parameter or parameters.

## operator[]

Returns the element that is specified by the parameters.

```

typename details::_Projection_result_type<value_type,_Rank>::_Const_result_type operator[] (
    int _I) const restrict(amp,cpu);

value_type& operator[] (
    const index<_Rank>& _Index) const restrict(amp,cpu);

```

### Parameters

*\_Index*

The index.

*\_I*

The index.

### Return Value

The value of the element at the index, or an `array_view` projected on the most-significant dimension.

## operator=

Copies the contents of the specified `array_view` object to this one.

```
array_view& operator= (
    const array_view& _Other) restrict(amp,cpu);

array_view& operator= (
    const array_view<value_type, _Rank>& _Other) restrict(amp,cpu);
```

### Parameters

*\_Other*

The `array_view` object to copy from.

### Return Value

A reference to this `array_view` object.

## rank

Stores the rank of the `array_view` object.

```
static const int rank = _Rank;
```

## refresh

Notifies the `array_view` object that its bound memory has been modified outside the `array_view` interface. A call to this method renders all cached information stale.

```
void refresh() const restrict(cpu);
```

## reinterpret\_as

Reinterprets the `array_view` through a one-dimensional `array_view`, which as an option can have a different value type than the source `array_view`.

### Syntax

```
template <
    typename _Value_type2
>
array_view< _Value_type2, _Rank> reinterpret_as() const restrict(amp,cpu);

template <
    typename _Value_type2
>
array_view<const _Value_type2, _Rank> reinterpret_as() const restrict(amp,cpu);
```

### Parameters

*\_Value\_type2*

The data type of the new `array_view` object.

### Return Value

An `array_view` object or a const `array_view` object that is based on this `array_view`, with the element type

converted from `T` to `_Value_type2`, and the rank reduced from  $N$  to 1.

### Remarks

Sometimes it is convenient to view a multi-dimensional array as a linear, one-dimensional array, which may have a different value type than the source array. You can achieve this on an `array_view` by using this method.

**Warning** Reinterpreting an `array_view` object by using a different value type is a potentially unsafe operation. This functionality should be used with care.

Here's an example:

```
struct RGB { float r; float g; float b; };

array<RGB,3> a = ...;
array_view<float,1> v = a.reinterpret_as<float>();

assert(v.extent == 3*a.extent);
```

## section

Returns a subsection of the `array_view` object that's at the specified origin and, optionally, that has the specified extent.

```
array_view section(
    const Concurrency::index<_Rank>& _Section_origin,
    const Concurrency::extent<_Rank>& _Section_extent) const restrict(amp,cpu);

array_view section(
    const Concurrency::index<_Rank>& _Idx) const restrict(amp,cpu);

array_view section(
    const Concurrency::extent<_Rank>& _Ext) const restrict(amp,cpu);

array_view section(
    int _I0,
    int _E0) const restrict(amp,cpu);

array_view section(
    int _I0,
    int _I1,
    int _E0,
    int _E1) const restrict(amp,cpu);

array_view section(
    int _I0,
    int _I1,
    int _I2,
    int _E0,
    int _E1,
    int _E2) const restrict(amp,cpu);
```

### Parameters

`_E0`

The most significant component of the extent of this section.

`_E1`

The next-to-most-significant component of the extent of this section.

`_E2`

The least significant component of the extent of this section.

*\_Ext*

The [extent](#) object that specifies the extent of the section. The origin is 0.

*\_Idx*

The [index](#) object that specifies the location of the origin. The subsection is the rest of the extent.

*\_I0*

The most significant component of the origin of this section.

*\_I1*

The next-to-most-significant component of the origin of this section.

*\_I2*

The least significant component of the origin of this section.

*\_Rank*

The rank of the section.

*\_Section\_extent*

The [extent](#) object that specifies the extent of the section.

*\_Section\_origin*

The [index](#) object that specifies the location of the origin.

### Return Value

A subsection of the `array_view` object that's at the specified origin and, optionally, that has the specified extent. When only the `index` object is specified, the subsection contains all elements in the associated extent that have indexes that are larger than the indexes of the elements in the `index` object.

## source\_accelerator\_view

Gets the source `accelerator_view` that this `array_view` is associated with.

```
__declspec(property(get= get_source_accelerator_view)) accelerator_view source_accelerator_view;
```

## synchronize

Synchronizes any modifications made to the `array_view` object back to its source data.

```
void synchronize(access_type _Access_type = access_type_read) const restrict(cpu);  
void synchronize() const restrict(cpu);
```

### Parameters

*\_Access\_type*

The intended [access\\_type](#) on the target [accelerator\\_view](#). This parameter has a default value of

```
access_type_read .
```

## synchronize\_async

Asynchronously synchronizes any modifications made to the `array_view` object back to its source data.

```
concurrency::completion_future synchronize_async(access_type _Access_type = access_type_read) const
restrict(cpu);

concurrency::completion_future synchronize_async() const restrict(cpu);
```

### Parameters

*\_Access\_type*

The intended [access\\_type](#) on the target [accelerator\\_view](#). This parameter has a default value of

```
access_type_read .
```

### Return Value

A future upon which to wait for the operation to complete.

## synchronize\_to

Synchronizes any modifications made to this [array\\_view](#) to the specified [accelerator\\_view](#).

```
void synchronize_to(
    const accelerator_view& _Accl_view,
    access_type _Access_type = access_type_read) const restrict(cpu);

void synchronize_to(
    const accelerator_view& _Accl_view) const restrict(cpu);
```

### Parameters

*\_Accl\_view*

The target [accelerator\\_view](#) to synchronize to.

*\_Access\_type*

The desired [access\\_type](#) on the target [accelerator\\_view](#). This parameter has a default value of [access\\_type\\_read](#).

## synchronize\_to\_async

Asynchronously synchronizes any modifications made to this [array\\_view](#) to the specified [accelerator\\_view](#).

```
concurrency::completion_future synchronize_to_async(
    const accelerator_view& _Accl_view,
    access_type _Access_type = access_type_read) const restrict(cpu);

concurrency::completion_future synchronize_to_async(
    const accelerator_view& _Accl_view) const restrict(cpu);
```

### Parameters

*\_Accl\_view*

The target [accelerator\\_view](#) to synchronize to.

*\_Access\_type*

The desired [access\\_type](#) on the target [accelerator\\_view](#). This parameter has a default value of [access\\_type\\_read](#).

### Return Value

A future upon which to wait for the operation to complete.

## value\_type

The value type of the `array_view` and the bound array.

```
typedef typenamevalue_type value_type;
```

## view\_as

Reinterprets this `array_view` as an `array_view` of a different rank.

```
template <
    int _New_rank
>
array_view<value_type,_New_rank> view_as(
    const Concurrency::extent<_New_rank>& _View_extent) const restrict(amp,cpu);

template <
    int _New_rank
>
array_view<const value_type,_New_rank> view_as(
    const Concurrency::extent<_New_rank> _View_extent) const restrict(amp,cpu);
```

### Parameters

*\_New\_rank*

The rank of the new `array_view` object.

*\_View\_extent*

The reshaping `extent`.

*value\_type*

The data type of the elements in both the original `array` object and the returned `array_view` object.

### Return Value

The `array_view` object that is constructed.

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# completion\_future Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

Represents a future corresponding to a C++ AMP asynchronous operation.

## Syntax

```
class completion_future;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">completion_future Constructor</a>	Initializes a new instance of the <code>completion_future</code> class.
<a href="#">~completion_future Destructor</a>	Destroys the <code>completion_future</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">get</a>	Waits until the associated asynchronous operation completes.
<a href="#">then</a>	Chains a callback function object to the <code>completion_future</code> object to be executed when the associated asynchronous operation finishes execution.
<a href="#">to_task</a>	Returns a <code>task</code> object corresponding to the associated asynchronous operation.
<a href="#">valid</a>	Gets a Boolean value that indicates whether the object is associated with an asynchronous operation.
<a href="#">wait</a>	Blocks until the associated asynchronous operation completes.
<a href="#">wait_for</a>	Blocks until the associated asynchronous operation completes or the time specified by <code>_Rel_time</code> has elapsed.
<a href="#">wait_until</a>	Blocks until the associated asynchronous operation completes or until the current time exceeds the value specified by <code>_Abs_time</code> .

### Public Operators

NAME	DESCRIPTION
------	-------------

NAME	DESCRIPTION
<code>operator std::shared_future&lt;void&gt;</code>	Implicitly converts the <code>completion_future</code> object to an <code>std::shared_future</code> object.
<code>operator=</code>	Copies the contents of the specified <code>completion_future</code> object into this one.

## Inheritance Hierarchy

`completion_future`

## Requirements

**Header:** `ampprt.h`

**Namespace:** `concurrency`

## `completion_future`

Initializes a new instance of the `completion_future` class.

### Syntax

```
completion_future();

completion_future(
    const completion_future& _Other );

completion_future(
    completion_future&& _Other );
```

### Parameters

*\_Other*

The `completion_future` object to copy or move.

### Overloads List

NAME	DESCRIPTION
<code>completion_future();</code>	Initializes a new instance of the <code>completion_future</code> Class
<code>completion_future(const completion_future&amp; _Other);</code>	Initializes a new instance of the <code>completion_future</code> class by copying a constructor.
<code>completion_future(completion_future&amp;&amp; _Other);</code>	Initializes a new instance of the <code>completion_future</code> class by moving a constructor.

## `get`

Waits until the associated asynchronous operation completes. Throws the stored exception if one was encountered during the asynchronous operation.

### Syntax

```
void get() const;
```

## operator std::shared\_future

Implicitly converts the `completion_future` object to an `std::shared_future` object.

### Syntax

```
operator std::shared_future<void>() const;
```

### Return Value

An `std::shared_future` object.

## operator=

Copies the contents of the specified `completion_future` object into this one.

### Syntax

```
completion_future& operator= (const completion_future& _Other );  
completion_future& operator= (completion_future&& _Other );
```

### Parameters

*\_Other*

The object to copy from.

### Return Value

A reference to this `completion_future` object.

## Overloads List

NAME	DESCRIPTION
<code>completion_future&amp; operator=(const completion_future&amp; _Other);</code>	Copies the contents of the specified <code>completion_future</code> object into this one, using a deep copy.
<code>completion_future&amp; operator=(completion_future&amp;&amp; _Other);</code>	Copies the contents of the specified <code>completion_future</code> object into this one, using a move assignment.

## then

Chains a callback function object to the `completion_future` object to be executed when the associated asynchronous operation finishes execution.

### Syntax

```
template <typename _Funcor>  
void then(const _Funcor & _Func ) const;
```

### Parameters

*\_Funcor*

The callback functor.

*\_Func*

The callback function object.

## to\_task

Returns a `task` object corresponding to the associated asynchronous operation.

### Syntax

```
concurrency::task<void> to_task() const;
```

### Return Value

A `task` object corresponding to the associated asynchronous operation.

## valid

Gets a Boolean value that indicates whether the object is associated with an asynchronous operation.

### Syntax

```
bool valid() const;
```

### Return Value

**true** if the object is associated with an asynchronous operation; otherwise, **false**.

## wait

Blocks until the associated asynchronous operation completes.

### Syntax

```
void wait() const;
```

## wait\_for

Blocks until the associated asynchronous operation completes or the time that's specified by `_Rel_time` has elapsed.

### Syntax

```
template <
    class _Rep,
    class _Period
>
std::future_status::future_status wait_for(
    const std::chrono::duration< _Rep, _Period>& _Rel_time ) const;
```

### Parameters

*\_Rep*

An arithmetic type that represents the number of ticks.

*\_Period*

A `std::ratio` that represents the number of seconds that elapse per tick.

`_Rel_time`

The maximum amount of time to wait for the operation to complete.

### Return Value

Returns:

- `std::future_status::deferred` if the associated asynchronous operation is not running.
- `std::future_status::ready` if the associated asynchronous operation is finished.
- `std::future_status::timeout` if the specified time period has elapsed.

## wait\_until

Blocks until the associated asynchronous operation completes or until the current time exceeds the value specified by `_Abs_time`.

### Syntax

```
template <
    class _Clock,
    class _Duration
>
std::future_status::future_status wait_until(
    const std::chrono::time_point< _Clock, _Duration>& _Abs_time ) const;
```

### Parameters

`_Clock`

The clock on which this time point is measured.

`_Duration`

The time interval since the start of `_Clock`'s epoch, after which the function will time out.

`_Abs_time`

The point in time after which the function will time out.

### Return Value

Returns:

1. `std::future_status::deferred` if the associated asynchronous operation is not running.
2. `std::future_status::ready` if the associated asynchronous operation is finished.
3. `std::future_status::timeout` if the time period specified has elapsed.

## ~completion\_future

Destroys the `completion_future` object.

### Syntax

```
~completion_future();
```

## See also

[Concurrency Namespace \(C++ AMP\)](#)



# extent Class (C++ AMP)

3/28/2019 • 4 minutes to read • [Edit Online](#)

Represents a vector of  $N$  integer values that specify the bounds of an  $N$ -dimensional space that has an origin of 0. The values in the vector are ordered from most significant to least significant.

## Syntax

```
template <int _Rank>
class extent;
```

## Parameters

*\_Rank*

The rank of the `extent` object.

## Requirements

**Header:** amp.h

**Namespace:** Concurrency

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">extent Constructor</a>	Initializes a new instance of the <code>extent</code> class.

### Public Methods

NAME	DESCRIPTION
<a href="#">contains</a>	Verifies that the specified <code>extent</code> object has the specified rank.
<a href="#">size</a>	Returns the total linear size of the extent (in units of elements).
<a href="#">tile</a>	Produces a <code>tiled_extent</code> object with the tile extents given by specified dimensions.

### Public Operators

NAME	DESCRIPTION
<a href="#">operator-</a>	Returns a new <code>extent</code> object that's created by subtracting the <code>index</code> elements from the corresponding <code>extent</code> elements.
<a href="#">operator--</a>	Decrements each element of the <code>extent</code> object.

NAME	DESCRIPTION
<code>operator%=</code>	Calculates the modulus (remainder) of each element in the <code>extent</code> object when that element is divided by a number.
<code>operator* =</code>	Multiplies each element of the <code>extent</code> object by a number.
<code>operator/=</code>	Divides each element of the <code>extent</code> object by a number.
<code>extent::operator[]</code>	Returns the element that's at the specified index.
<code>operator+</code>	Returns a new <code>extent</code> object that's created by adding the corresponding <code>index</code> and <code>extent</code> elements.
<code>operator++</code>	Increments each element of the <code>extent</code> object.
<code>operator+=</code>	Adds the specified number to each element of the <code>extent</code> object.
<code>operator=</code>	Copies the contents of another <code>extent</code> object into this one.
<code>operator-=</code>	Subtracts the specified number from each element of the <code>extent</code> object.

### Public Constants

NAME	DESCRIPTION
<code>rank Constant</code>	Gets the rank of the <code>extent</code> object.

## Inheritance Hierarchy

`extent`

## contains

Indicates whether the specified `index` value is contained within the `extent` object.

### Syntax

```
bool contains(const index<rank>& _Index) const restrict(amp,cpu);
```

### Parameters

`_Index`

The `index` value to test.

### Return Value

**true** if the specified `index` value is contained in the `extent` object; otherwise, **false**.

# extent

Initializes a new instance of the `extent` class.

## Syntax

```
extent() restrict(amp,cpu);
extent(const extent<_Rank>& _Other) restrict(amp,cpu);
explicit extent(int _I) restrict(amp,cpu);
extent(int _I0, int _I1) restrict(amp,cpu);
extent(int _I0, int _I1, int _I2) restrict(amp,cpu);
explicit extent(const int _Array[_Rank])restrict(amp,cpu);
```

## Parameters

### *\_Array*

An array of `_Rank` integers that is used to create the new `extent` object.

### *\_I*

The length of the extent.

### *\_I0*

The length of the most significant dimension.

### *\_I1*

The length of the next-to-most-significant dimension.

### *\_I2*

The length of the least significant dimension.

### *\_Other*

An `extent` object on which the new `extent` object is based.

## Remarks

The parameterless constructor initializes an `extent` object that has a rank of three.

If an array is used to construct an `extent` object, the length of the array must match the rank of the `extent` object.

## operator%=

Calculates the modulus (remainder) of each element in the `extent` when that element is divided by a number.

## Syntax

```
extent<_Rank>& operator%=(int _Rhs) restrict(cpu, direct3d);
```

## Parameters

### *\_Rhs*

The number to find the modulus of.

## Return Value

The `extent` object.

## operator\*=

Multiplies each element in the `extent` object by the specified number.

### Syntax

```
extent<_Rank>& operator*=(int _Rhs) restrict(amp,cpu);
```

### Parameters

*\_Rhs*

The number to multiply.

### Return Value

The `extent` object.

## operator+

Returns a new `extent` object created by adding the corresponding `index` and `extent` elements.

### Syntax

```
extent<_Rank> operator+(const index<_Rank>& _Rhs) restrict(amp,cpu);
```

### Parameters

*\_Rhs*

The `index` object that contains the elements to add.

### Return Value

The new `extent` object.

## operator++

Increments each element of the `extent` object.

### Syntax

```
extent<_Rank>& operator++() restrict(amp,cpu);  
extent<_Rank> operator++(int)restrict(amp,cpu);
```

### Return Value

For the prefix operator, the `extent` object (`*this`). For the suffix operator, a new `extent` object.

## operator+=

Adds the specified number to each element of the `extent` object.

### Syntax

```
extent<_Rank>& operator+=(const extent<_Rank>& _Rhs) restrict(amp,cpu);  
extent<_Rank>& operator+=(const index<_Rank>& _Rhs) restrict(amp,cpu);  
extent<_Rank>& operator+=(int _Rhs) restrict(amp,cpu);
```

### Parameters

*\_Rhs*

The number, index, or extent to add.

### Return Value

The resulting `extent` object.

## operator-

Creates a new `extent` object by subtracting each element in the specified `index` object from the corresponding element in this `extent` object.

### Syntax

```
extent<_Rank> operator-(const index<_Rank>& _Rhs) restrict(amp,cpu);
```

### Parameters

*\_Rhs*

The `index` object that contains the elements to subtract.

### Return Value

The new `extent` object.

## operator--

Decrements each element in the `extent` object.

### Syntax

```
extent<_Rank>& operator--() restrict(amp,cpu);  
extent<_Rank> operator--(int)restrict(amp,cpu);
```

### Return Value

For the prefix operator, the `extent` object (`*this`). For the suffix operator, a new `extent` object.

## operator/=

Divides each element in the `extent` object by the specified number.

### Syntax

```
extent<_Rank>& operator/=(int _Rhs) restrict(amp,cpu);
```

### Parameters

*\_Rhs*

The number to divide by.

### Return Value

The `extent` object.

## operator-=

Subtracts the specified number from each element of the `extent` object.

### Syntax

```
extent<_Rank>& operator-=(const extent<_Rank>& _Rhs) restrict(amp,cpu);
extent<_Rank>& operator-=(const index<_Rank>& _Rhs) restrict(amp,cpu);
extent<_Rank>& operator-=(int _Rhs) restrict(amp,cpu);
```

### Parameters

*\_Rhs*

The number to subtract.

### Return Value

The resulting `extent` object.

## operator=

Copies the contents of another `extent' object into this one.

### Syntax

```
extent<_Rank>& operator=(const extent<_Rank>& _Other) restrict(amp,cpu);
```

### Parameters

*\_Other*

The `extent` object to copy from.

### Return Value

A reference to this `extent` object.

## extent::operator []

Returns the element that's at the specified index.

### Syntax

```
int operator[](unsigned int _Index) const restrict(amp,cpu);
int& operator[](unsigned int _Index) restrict(amp,cpu);
```

### Parameters

*\_Index*

An integer from 0 through the rank minus 1.

### Return Value

The element that's at the specified index.

## rank

Stores the rank of the `extent' object.

### Syntax

```
static const int rank = _Rank;
```

## size

Returns the total linear size of the `extent` object (in units of elements).

## Syntax

```
unsigned int size() const restrict(amp,cpu);
```

## tile

Produces a `tiled_extent` object with the specified tile dimensions.

```
template <int _Dim0>
tiled_extent<_Dim0> tile() const ;

template <int _Dim0, int _Dim1>
tiled_extent<_Dim0, _Dim1> tile() const ;

template <int _Dim0, int _Dim1, int _Dim2>
tiled_extent<_Dim0, _Dim1, _Dim2> tile() const ;
```

## Parameters

*\_Dim0*

The most significant component of the tiled extent. *\_Dim1*

The next-to-most-significant component of the tiled extent. *\_Dim2*

The least significant component of the tiled extent.

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# index Class

3/28/2019 • 3 minutes to read • [Edit Online](#)

Defines an  $N$ -dimensional index `pographics-cpp-amp.md`.

## Syntax

```
template <int _Rank>
class index;
```

### Parameters

`_Rank`

The rank, or number of dimensions.

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">index Constructor</a>	Initializes a new instance of the <code>index</code> class.

### Public Operators

NAME	DESCRIPTION
<a href="#">operator--</a>	Decrements each element of the <code>index</code> object.
<a href="#">operator%="&gt;</a>	Calculates the modulus (remainder) of each element in the <code>index</code> object when that element is divided by a number.
<a href="#">operator*="&gt;</a>	Multiplies each element of the <code>index</code> object by a number.
<a href="#">operator/="&gt;</a>	Divides each element of the <code>index</code> object by a number.
<a href="#">index::operator[]</a>	Returns the element that's at the specified index.
<a href="#">operator++</a>	Increments each element of the <code>index</code> object.
<a href="#">operator+=</a>	Adds the specified number to each element of the <code>index</code> object.
<a href="#">operator="&gt;</a>	Copies the contents of the specified <code>index</code> object into this one.
<a href="#">operator-="&gt;</a>	Subtracts the specified number from each element of the <code>index</code> object.

### Public Constants

NAME	DESCRIPTION
<a href="#">rank Constant</a>	Stores the rank of the <code>index</code> object.

## Inheritance Hierarchy

`index`

## Remarks

The `index` structure represents a coordinate vector of  $N$  integers that specifies a unique position in an  $N$ -dimensional space. The values in the vector are ordered from most significant to least significant. You can retrieve the values of the components using `operator=`.

## Requirements

**Header:** `amp.h`

**Namespace:** `Concurrency`

## `index` Constructor

Initializes a new instance of the `index` class.

```

index() restrict(amp,cpu);

index(
    const index<_Rank>& _Other
) restrict(amp,cpu);

explicit index(
    int _I
) restrict(amp,cpu);

index(
    int _I0,
    int _I1
) restrict(amp,cpu);

index(
    int _I0,
    int _I1,
    int _I2
) restrict(amp,cpu);

explicit index(
    const int _Array[_Rank]
) restrict(amp,cpu);

```

### Parameters

`_Array`

A one-dimensional array with the rank values.

`_I`

The index location in a one-dimensional index.

`_I0`

The length of the most significant dimension.

*\_I1*

The length of the next-to-most-significant dimension.

*\_I2*

The length of the least significant dimension.

*\_Other*

An index object on which the new index object is based.

## operator--

Decrements each element of the index object.

```
index<_Rank>& operator--() restrict(amp,cpu);

index operator--(
    int
) restrict(amp,cpu);
```

### Return values

For the prefix operator, the index object (\*this). For the suffix operator, a new index object.

## operator%=

Calculates the modulus (remainder) of each element in the index object when that element is divided by the specified number.

```
index<_Rank>& operator%=(
    int _Rhs
) restrict(cpu, amp);
```

### Parameters

*\_Rhs*

The number to divide by to find the modulus.

## Return Value

The index object.

## operator\*=

Multiplies each element in the index object by the specified number.

```
index<_Rank>& operator*=(
    int _Rhs
) restrict(amp,cpu);
```

### Parameters

*\_Rhs*

The number to multiply.

## operator/=

Divides each element in the index object by the specified number.

```
index<_Rank>& operator/=(
    int _Rhs
) restrict(amp,cpu);
```

### Parameters

*\_Rhs*

The number to divide by.

## operator[]

Returns the component of the index at the specified location.

```
int operator[] (
    unsigned _Index
) const restrict(amp,cpu);

int& operator[] (
    unsigned _Index
) restrict(amp,cpu);
```

### Parameters

*\_Index*

An integer from 0 through the rank minus 1.

### Return Value

The element that's at the specified index.

### Remarks

This following example displays the component values of the index.

```
// Prints 1 2 3.
concurrency::index<3> idx(1, 2, 3);
std::cout << idx[0] << "\n";
std::cout << idx[1] << "\n";
std::cout << idx[2] << "\n";
```

## operator++

Increments each element of the index object.

```
index<_Rank>& operator++() restrict(amp,cpu);

index<_Rank> operator++(
    int
) restrict(amp,cpu);
```

### Return Value

For the prefix operator, the index object (\*this). For the suffix operator, a new index object.

## operator+=

Adds the specified number to each element of the index object.

```
index<_Rank>& operator+=(  
    const index<_Rank>& _Rhs  
) restrict(amp,cpu);  
  
index<_Rank>& operator+=(  
    int _Rhs  
) restrict(amp,cpu);
```

### Parameters

*\_Rhs*

The number to add.

### Return Value

The index object.

## operator=

Copies the contents of the specified index object into this one.

```
index<_Rank>& operator=(  
    const index<_Rank>& _Other  
) restrict(amp,cpu);
```

### Parameters

*\_Other*

The index object to copy from.

### Return Value

A reference to this index object.

## operator-=

Subtracts the specified number from each element of the index object.

```
index<_Rank>& operator-=(  
    const index<_Rank>& _Rhs  
) restrict(amp,cpu);  
  
index<_Rank>& operator-=(  
    int _Rhs  
) restrict(amp,cpu);
```

### Parameters

*\_Rhs*

The number to subtract.

### Return Value

The index object.

## Rank

Gets the rank of the index object.

```
static const int rank = _Rank;
```

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# invalid\_compute\_domain Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The exception that's thrown when the runtime can't start a kernel by using the compute domain specified at the [parallel\\_for\\_each](#) call site.

## Syntax

```
class invalid_compute_domain : public runtime_exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">invalid_compute_domain Constructor</a>	Initializes a new instance of the <code>invalid_compute_domain</code> class.

## Inheritance Hierarchy

`exception`

`runtime_exception`

`invalid_compute_domain`

## Requirements

**Header:** `amprth`

**Namespace:** `Concurrency`

## invalid\_compute\_domain

Initializes a new instance of the class.

## Syntax

```
explicit invalid_compute_domain(  
    const char * _Message ) throw();  
  
invalid_compute_domain() throw();
```

### Parameters

*\_Message*

A description of the error.

### Return Value

An instance of the `invalid_compute_domain` class

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# out\_of\_memory Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The exception that is thrown when a method fails because of a lack of system or device memory.

## Syntax

```
class out_of_memory : public runtime_exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">out_of_memory Constructor</a>	Initializes a new instance of the <code>out_of_memory</code> class.

## Inheritance Hierarchy

`exception`

`runtime_exception`

`out_of_memory`

## Requirements

**Header:** amprth

**Namespace:** Concurrency

## out\_of\_memory

Initializes a new instance of the class.

### Syntax

```
explicit out_of_memory(  
    const char * _Message ) throw();  
  
out_of_memory () throw();
```

### Parameters

*\_Message*

A description of the error.

### Return Value

A new instance of the `out_of_memory` class.

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# runtime\_exception Class

5/21/2019 • 2 minutes to read • [Edit Online](#)

The base type for exceptions in the C++ Accelerated Massive Parallelism (AMP) library.

## Syntax

```
class runtime_exception : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">runtime_exception Constructor</a>	Initializes a new instance of the <code>runtime_exception</code> class.
<a href="#">~runtime_exception Destructor</a>	Destroys the <code>runtime_exception</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">get_error_code</a>	Returns the error code that caused the exception.

### Public Operators

NAME	DESCRIPTION
<a href="#">operator=</a>	Copies the contents of the specified <code>runtime_exception</code> object into this one.

## Inheritance Hierarchy

`exception`

`runtime_exception`

## Requirements

**Header:** `ampr.h`

**Namespace:** `Concurrency`

## runtime\_exception Constructor

Initializes a new instance of the class.

## Syntax

```
runtime_exception(  
    const char * _Message,  
    HRESULT _Hresult ) throw();  
  
explicit runtime_exception(  
    HRESULT _Hresult ) throw();  
  
runtime_exception(  
    const runtime_exception & _Other ) throw();
```

### Parameters

#### *\_Message*

A description of the error that caused the exception.

#### *\_Hresult*

The HRESULT of error that caused the exception.

#### *\_Other*

The `runtime_exception` object to copy.

### Return Value

The `runtime_exception` object.

## ~runtime\_exception Destructor

Destroys the object.

### Syntax

```
virtual ~runtime_exception() throw();
```

## get\_error\_code

Returns the error code that caused the exception.

### Syntax

```
HRESULT get_error_code() const throw();
```

### Return Value

The HRESULT of error that caused the exception.

## operator=

Copies the contents of the specified `runtime_exception` object into this one.

### Syntax

```
runtime_exception & operator= (    const runtime_exception & _Other ) throw();
```

### Parameters

#### *\_Other*

The `runtime_exception` object to copy.

### Return Value

A reference to this `runtime_exception` object.

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# tile\_barrier Class

5/21/2019 • 2 minutes to read • [Edit Online](#)

Synchronizes the execution of threads that are running in the thread group (the tile) by using `wait` methods. Only the runtime can instantiate this class.

## Syntax

```
class tile_barrier;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">tile_barrier Constructor</a>	Initializes a new instance of the <code>tile_barrier</code> class.

### Public Methods

NAME	DESCRIPTION
<a href="#">wait</a>	Instructs all threads in the thread group (tile) to stop executing until all threads in the tile have finished waiting.
<a href="#">wait_with_all_memory_fence</a>	Blocks execution of all threads in a tile until all memory accesses have been completed and all threads in the tile have reached this call.
<a href="#">wait_with_global_memory_fence</a>	Blocks execution of all threads in a tile until all global memory accesses have been completed and all threads in the tile have reached this call.
<a href="#">wait_with_tile_static_memory_fence</a>	Blocks execution of all threads in a tile until all <code>tile_static</code> memory accesses have been completed and all threads in the tile have reached this call.

## Inheritance Hierarchy

`tile_barrier`

## Requirements

**Header:** amp.h

**Namespace:** Concurrency

## tile\_barrier Constructor

Initializes a new instance of the class by copying an existing one.

## Syntax

```
tile_barrier(  
    const tile_barrier& _Other ) restrict(amp,cpu);
```

## Parameters

*\_Other*

The `tile_barrier` object to copy.

## wait

Instructs all threads in the thread group (tile) to stop execution until all threads in the tile have finished waiting.

## Syntax

```
void wait() const restrict(amp);
```

## wait\_with\_all\_memory\_fence

Blocks execution of all threads in a tile until all threads in a tile have reached this call. This ensures that all memory accesses are visible to other threads in the thread tile, and have been executed in program order.

## Syntax

```
void wait_with_all_memory_fence() const restrict(amp);
```

## wait\_with\_global\_memory\_fence

Blocks execution of all threads in a tile until all threads in a tile have reached this call. This ensures that all global memory accesses are visible to other threads in the thread tile, and have been executed in program order.

## Syntax

```
void wait_with_global_memory_fence() const restrict(amp);
```

## wait\_with\_tile\_static\_memory\_fence

Blocks execution of all threads in a tile until all threads in a tile have reached this call. This ensures that `tile_static` memory accesses are visible to other threads in the thread tile, and have been executed in program order.

## Syntax

```
void wait_with_tile_static_memory_fence() const restrict(amp);
```

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# tiled\_extent Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

A `tiled_extent` object is an `extent` object of one to three dimensions that subdivides the extent space into one-, two-, or three-dimensional tiles.

## Syntax

```
template <
    int _Dim0,
    int _Dim1,
    int _Dim2
>
class tiled_extent : public Concurrency::extent<3>;

template <
    int _Dim0,
    int _Dim1
>
class tiled_extent<_Dim0, _Dim1, 0> : public Concurrency::extent<2>;

template <
    int _Dim0
>
class tiled_extent<_Dim0, 0, 0> : public Concurrency::extent<1>;
```

## Parameters

*\_Dim0*

The length of the most significant dimension.

*\_Dim1*

The length of the next-to-most significant dimension.

*\_Dim2*

The length of the least significant dimension.

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">tiled_extent Constructor</a>	Initializes a new instance of the <code>tiled_extent</code> class.

### Public Methods

NAME	DESCRIPTION
<a href="#">get_tile_extent</a>	Returns an <code>extent</code> object that captures the values of the <code>tiled_extent</code> template arguments <code>_Dim0</code> , <code>_Dim1</code> , and <code>_Dim2</code> .
<a href="#">pad</a>	Returns a new <code>tiled_extent</code> object with extents adjusted up to be evenly divisible by the tile dimensions.

NAME	DESCRIPTION
<a href="#">truncate</a>	Returns a new <code>tiled_extent</code> object with extents adjusted down to be evenly divisible by the tile dimensions.

### Public Operators

NAME	DESCRIPTION
<code>operator=</code>	Copies the contents of the specified <code>tiled_index</code> object into this one.

### Public Constants

NAME	DESCRIPTION
<a href="#">tile_dim0 Constant</a>	Stores the length of the most significant dimension.
<a href="#">tile_dim1 Constant</a>	Stores the length of the next-to-most significant dimension.
<a href="#">tile_dim2 Constant</a>	Stores the length of the least significant dimension.

### Public Data Members

NAME	DESCRIPTION
<a href="#">tile_extent</a>	Gets an <code>extent</code> object that captures the values of the <code>tiled_extent</code> template arguments <code>_Dim0</code> , <code>_Dim1</code> , and <code>_Dim2</code> .

## Inheritance Hierarchy

`extent`

`tiled_extent`

## Requirements

**Header:** `amp.h`

**Namespace:** `Concurrency`

## `tiled_extent` Constructor

Initializes a new instance of the `tiled_extent` class.

### Syntax

```
tiled_extent();

tiled_extent(
    const Concurrency::extent<rank>& _Other );

tiled_extent(
    const tiled_extent& _Other );
```

## Parameters

*\_Other*

The `extent` or `tiled_extent` object to copy.

## get\_tile\_extent

Returns an `extent` object that captures the values of the `tiled_extent` template arguments `_Dim0`, `_Dim1`, and `_Dim2`.

## Syntax

```
Concurrency::extent<rank> get_tile_extent() const restrict(amp,cpu);
```

## Return Value

An `extent` object that captures the dimensions of this `tiled_extent` instance.

## pad

Returns a new `tiled_extent` object with extents adjusted up to be evenly divisible by the tile dimensions.

## Syntax

```
tiled_extent pad() const;
```

## Return Value

The new `tiled_extent` object, by value.

## truncate

Returns a new `tiled_extent` object with extents adjusted down to be evenly divisible by the tile dimensions.

## Syntax

```
tiled_extent truncate() const;
```

## Return Value

Returns a new `tiled_extent` object with extents adjusted down to be evenly divisible by the tile dimensions.

## operator=

Copies the contents of the specified `tiled_index` object into this one.

## Syntax

```
tiled_extent& operator= (  
    const tiled_index& _Other ) restrict (amp, cpu);
```

## Parameters

*\_Other*

The `tiled_index` object to copy from.

## Return Value

A reference to this `tiled_index` instance.

## tile\_dim0

Stores the length of the most significant dimension.

### Syntax

```
static const int tile_dim0 = _Dim0;
```

## tile\_dim1

Stores the length of the next-to-most significant dimension.

### Syntax

```
static const int tile_dim1 = _Dim1;
```

## tile\_dim2

Stores the length of the least significant dimension.

### Syntax

```
static const int tile_dim2 = _Dim2;
```

## tile\_extent

Gets an `extent` object that captures the values of the `tiled_extent` template arguments `_Dim0`, `_Dim1`, and `_Dim2`.

### Syntax

```
__declspec(property(get= get_tile_extent)) Concurrency::extent<rank> tile_extent;
```

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# tiled\_index Class

3/28/2019 • 3 minutes to read • [Edit Online](#)

Provides an index into a [tiled\\_extent](#) object. This class has properties to access elements relative to the local tile origin and relative to the global origin. For more information about tiled spaces, see [Using Tiles](#).

## Syntax

```
template <
    int _Dim0,
    int _Dim1 = 0,
    int _Dim2 = 0
>
class tiled_index : public _Tiled_index_base<3>;

template <
    int _Dim0,
    int _Dim1
>
class tiled_index<_Dim0, _Dim1, 0> : public _Tiled_index_base<2>;

template <
    int _Dim0
>
class tiled_index<_Dim0, 0, 0> : public _Tiled_index_base<1>;
```

### Parameters

*\_Dim0*

The length of the most significant dimension.

*\_Dim1*

The length of the next-to-most significant dimension.

*\_Dim2*

The length of the least significant dimension.

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">tiled_index Constructor</a>	Initializes a new instance of the <code>tiled_index</code> class.

### Public Methods

NAME	DESCRIPTION
<a href="#">get_tile_extent</a>	Returns an <a href="#">extent</a> object that has the values of the <code>tiled_index</code> template arguments <code>_Dim0</code> , <code>_Dim1</code> , and <code>_Dim2</code> .

### Public Constants

NAME	DESCRIPTION
<a href="#">barrier Constant</a>	Stores a <a href="#">tile_barrier</a> object that represents a barrier in the current tile of threads.
<a href="#">global Constant</a>	Stores an <a href="#">index</a> object of rank 1, 2, or 3 that represents the global index in a grid object.
<a href="#">local Constant</a>	Stores an <a href="#">index</a> object of rank 1, 2, or 3 that represents the relative index in the current tile of a <a href="#">tiled_extent</a> object.
<a href="#">rank Constant</a>	Stores the rank of the <a href="#">tiled_index</a> object.
<a href="#">tile Constant</a>	Stores an <a href="#">index</a> object of rank 1, 2, or 3 that represents the coordinates of the current tile of a <a href="#">tiled_extent</a> object.
<a href="#">tile_dim0 Constant</a>	Stores the length of the most significant dimension.
<a href="#">tile_dim1 Constant</a>	Stores the length of the next-to-most significant dimension.
<a href="#">tile_dim2 Constant</a>	Stores the length of the least significant dimension.
<a href="#">tile_origin Constant</a>	Stores an <a href="#">index</a> object of rank 1, 2, or 3 that represents the global coordinates of the origin of the current tile in a <a href="#">tiled_extent</a> object.

## Public Data Members

NAME	DESCRIPTION
<a href="#">tile_extent</a>	Gets an <a href="#">extent</a> object that has the values of the <a href="#">tiled_index</a> template arguments <a href="#">tiled_index</a> template arguments <code>_Dim0</code> , <code>_Dim1</code> , and <code>_Dim2</code> .

## Inheritance Hierarchy

`_Tiled_index_base`

`tiled_index`

## Requirements

**Header:** `amp.h`

**Namespace:** `Concurrency`

## `tiled_index` Constructor

Initializes a new instance of the `tiled_index` class.

## Syntax

```

tiled_index(
    const index<rank>& _Global,
    const index<rank>& _Local,
    const index<rank>& _Tile,
    const index<rank>& _Tile_origin,
    const tile_barrier& _Barrier ) restrict(amp,cpu);

tiled_index(
    const tiled_index& _Other ) restrict(amp,cpu);

```

### Parameters

#### *\_Global*

The global [index](#) of the constructed `tiled_index`.

#### *\_Local*

The local [index](#) of the constructed `tiled_index`.

#### *\_Tile*

The tile [index](#) of the constructed `tiled_index`.

#### *\_Tile\_origin*

The tile origin [index](#) of the constructed `tiled_index`.

#### *\_Barrier*

The [tile\\_barrier](#) object of the constructed `tiled_index`.

#### *\_Other*

The `tiled_index` object to be copied to the constructed `tiled_index`.

## Overloads

Name	Description
<pre>tiled_index(const index&lt;rank&gt;&amp; _Global, const index&lt;rank&gt;&amp; _Local, const index&lt;rank&gt;&amp; _Tile, const index&lt;rank&gt;&amp; _Tile_origin, const tile_barrier&amp; _Barrier restrict(amp,cpu);</pre>	<p>Initializes a new instance of the <code>tiled_index</code> class from the index of the tile in global coordinates and the relative position in the tile in local coordinates. The <code>_Global</code> and <code>_Tile_origin</code> parameters are computed.</p>
<pre>tiled_index( const tiled_index&amp; _Other) restrict(amp,cpu);</pre>	<p>Initializes a new instance of the <code>tiled_index</code> class by copying the specified <code>tiled_index</code> object.</p>

## get\_tile\_extent

Returns an [extent](#) object that has the values of the `tiled_index` template arguments `_Dim0`, `_Dim1`, and `_Dim2`.

## Syntax

```
extent<rank> get_tile_extent()restrict(amp,cpu);
```

## Return Value

An `extent` object that has the values of the `tiled_index` template arguments `_Dim0`, `_Dim1`, and `_Dim2`.

## barrier

Stores a [tile\\_barrier](#) object that represents a barrier in the current tile of threads.

## Syntax

```
const tile_barrier barrier;
```

## global

Stores an [index](#) object of rank 1, 2, or 3 that represents the global index of an object.

## Syntax

```
const index<rank> global;
```

## local

Stores an [index](#) object of rank 1, 2, or 3 that represents the relative index in the current tile of a [tiled\\_extent](#) object.

## Syntax

```
const index<rank> local;
```

## rank

Stores the rank of the [tiled\\_index](#) object.

## Syntax

```
static const int rank = _Rank;
```

## tile

Stores an [index](#) object of rank 1, 2, or 3 that represents the coordinates of the current tile of a [tiled\\_extent](#) object.

## Syntax

```
const index<rank> tile;
```

## tile\_dim0

Stores the length of the most significant dimension.

## Syntax

```
static const int tile_dim0 = _Dim0;
```

## tile\_dim1

Stores the length of the next-to-most significant dimension.

## Syntax

```
static const int tile_dim1 = _Dim1;
```

## tile\_dim2

Stores the length of the least significant dimension.

## Syntax

```
static const int tile_dim2 = _Dim2;
```

## tile\_origin

Stores an [index](#) object of rank 1, 2, or 3 that represents the global coordinates of the origin of the current tile within a [tiled\\_extent](#) object.

## Syntax

```
const index<rank> tile_origin
```

## tile\_extent

Gets an [extent](#) object that has the values of the `tiled_index` template arguments `tiled_index` template arguments `_Dim0`, `_Dim1`, and `_Dim2`.

## Syntax

```
__declspec(property(get= get_tile_extent)) extent<rank> tile_extent;
```

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# uninitialized\_object Class

5/21/2019 • 2 minutes to read • [Edit Online](#)

The exception that is thrown when an uninitialized object is used.

## Syntax

```
class uninitialized_object : public runtime_exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">uninitialized_object Constructor</a>	Initializes a new instance of the <code>uninitialized_object</code> class.

## Inheritance Hierarchy

`exception`

`runtime_exception`

`uninitialized_object`

## Requirements

**Header:** `amprth`

**Namespace:** `Concurrency`

## uninitialized\_object

Constructs a new instance of the `uninitialized_object` exception.

### Syntax

```
explicit uninitialized_object(  
    const char * _Message ) throw();  
  
uninitialized_object() throw();
```

### Parameters

*\_Message*

A description of the error.

### Return Value

The `uninitialized_object` exception object.

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# unsupported\_feature Class

5/10/2019 • 2 minutes to read • [Edit Online](#)

The exception that is thrown when an unsupported feature is used.

## Syntax

```
class unsupported_feature : public runtime_exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">unsupported_feature Constructor</a>	Constructs a new instance of the <code>unsupported_feature</code> exception.

## Inheritance Hierarchy

`exception`

`runtime_exception`

`unsupported_feature`

## unsupported\_feature

Constructs a new instance of the `unsupported_feature` exception.

### Syntax

```
explicit unsupported_feature(  
    const char * _Message ) throw();  
  
unsupported_feature() throw();
```

### Parameters

*\_Message*

A description of the error.

### Return Value

The `unsupported_feature` object.

## Requirements

**Header:** `ampprt.h`

**Namespace:** `Concurrency`

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# Concurrency::direct3d Namespace

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `direct3d` namespace provides functions that support D3D interoperability. It enables seamless use of D3D resources for compute in AMP code as well as allow use of resources created in AMP in D3D code, without creating redundant intermediate copies. You can incrementally accelerate the compute intensive sections of your DirectX applications by using C++ AMP and use the D3D API on data produced from AMP computations.

## Syntax

```
namespace direct3d;
```

## Members

### Classes

NAME	DESCRIPTION
<a href="#">scoped_d3d_access_lock Class</a>	An RAll wrapper for a D3D access lock on an <code>accelerator_view</code> object.

### Structures

NAME	DESCRIPTION
<a href="#">adopt_d3d_access_lock_t Structure</a>	Tag type to indicate the D3D access lock should be adopted rather than acquired.

### Functions

NAME	DESCRIPTION
<a href="#">abs</a>	Returns the absolute value of the argument
<a href="#">clamp</a>	Overloaded. Clamps <code>_X</code> to the specified <code>_Min</code> and <code>_Max</code> range
<a href="#">countbits</a>	Counts the number of set bits in <code>_X</code>
<a href="#">create_accelerator_view</a>	Creates an <code>accelerator_view Class</code> from a pointer to a Direct3D device interface
<a href="#">d3d_access_lock</a>	Acquires a lock on an <code>accelerator_view</code> for the purpose of safely performing D3D operations on resources shared with the <code>accelerator_view</code>
<a href="#">d3d_access_try_lock</a>	Attempt to acquire the D3D access lock on an <code>accelerator_view</code> without blocking.
<a href="#">d3d_access_unlock</a>	Release the D3D access lock on the given <code>accelerator_view</code> .

NAME	DESCRIPTION
<a href="#">firstbithigh</a>	Gets the location of the first set bit in <code>_X</code> , starting from the highest order bit and working downward
<a href="#">firstbitlow</a>	Gets the location of the first set bit in <code>_X</code> , starting from the lowest order bit and working upward
<a href="#">get_buffer</a>	Get the D3D buffer interface underlying an array.
<a href="#">imax</a>	Compares two values, returning the value which is greater.
<a href="#">imin</a>	Compares two values, returning the value which is smaller.
<a href="#">is_timeout_disabled</a>	Returns a boolean flag indicating if timeout is disabled for the specified <code>accelerator_view</code> .
<a href="#">mad</a>	Overloaded. Performs an arithmetic multiply/add operation on three arguments: <code>_X * _Y + _Z</code>
<a href="#">make_array</a>	Create an array from a D3D buffer interface pointer.
<a href="#">noise</a>	Generates a random value by using the Perlin noise algorithm
<a href="#">radians</a>	Converts <code>_X</code> from degrees to radians
<a href="#">rcp</a>	Calculates a fast, approximate reciprocal of the argument
<a href="#">reversebits</a>	Reverses the order of the bits in <code>_X</code>
<a href="#">saturate</a>	Clamps <code>_X</code> within the range of 0 to 1
<a href="#">sign</a>	Overloaded. Returns the sign of the argument
<a href="#">smoothstep</a>	Returns a smooth Hermite interpolation between 0 and 1, if <code>_X</code> is in the range <code>[_Min, _Max]</code> .
<a href="#">step</a>	Compares two values, returning 0 or 1 based on which value is greater
<a href="#">umax</a>	Compares two unsigned values, returning the value which is greater.
<a href="#">umin</a>	Compares two unsigned values, returning the value which is smaller.

## Requirements

**Header:** `amp.h`

**Namespace:** `Concurrency`

## See also



# Concurrency::direct3d namespace functions (AMP)

3/4/2019 • 7 minutes to read • [Edit Online](#)

<a href="#">abs</a>	<a href="#">clamp</a>	<a href="#">countbits</a>
<a href="#">create_accelerator_view</a>	<a href="#">d3d_access_lock</a>	
<a href="#">d3d_access_try_lock</a>	<a href="#">d3d_access_unlock</a>	<a href="#">firstbithigh</a>
<a href="#">firstbitlow</a>	<a href="#">get_buffer</a>	<a href="#">get_device</a>
<a href="#">imax</a>	<a href="#">imin</a>	<a href="#">is_timeout_disabled</a>
<a href="#">mad</a>	<a href="#">make_array</a>	<a href="#">noise</a>
<a href="#">radians</a>	<a href="#">rcp</a>	<a href="#">reversebits</a>
<a href="#">saturate</a>	<a href="#">sign</a>	<a href="#">smoothstep</a>
<a href="#">step</a>	<a href="#">umax</a>	<a href="#">umin</a>

## Requirements

**Header:** amp.h **Namespace:** Concurrency

### abs

Returns the absolute value of the argument

```
inline int abs(int _X) restrict(amp);
```

#### Parameters

`_X`

Integer value

#### Return Value

Returns the absolute value of the argument.

### clamp

Computes the value of the first specified argument clamped to a range defined by the second and third specified arguments.

```
inline float clamp(  
    float _X,  
    float _Min,  
    float _Max) restrict(amp);  
  
inline int clamp(  
    int _X,  
    int _Min,  
    int _Max) restrict(amp);
```

### Parameters

*\_X*

The value to be clamped

*\_Min*

The lower bound of the clamping range.

*\_Max*

The upper bound of the clamping range.

### Return Value

The clamped value of `_X`.

## countbits

Counts the number of set bits in *\_X*

```
inline unsigned int countbits(unsigned int _X) restrict(amp);
```

### Parameters

*\_X*

Unsigned integer value

### Return Value

Returns the number of set bits in *\_X*

## create\_accelerator\_view

Creates an [accelerator\\_view](#) object from a pointer to a Direct3D device interface.

## Syntax

```
accelerator_view create_accelerator_view(  
    IUnknown * _D3D_device  
    queuing_mode _Qmode = queuing_mode_automatic);  
  
accelerator_view create_accelerator_view(  
    accelerator& _Accelerator,  
    bool _Disable_timeout  
    queuing_mode _Qmode = queuing_mode_automatic);
```

### Parameters

*\_Accelerator*

The accelerator on which the new `accelerator_view` is to be created.

`_D3D_device`

The pointer to the Direct3D device interface.

`_Disable_timeout`

A Boolean parameter that specifies whether timeout should be disabled for the newly created `accelerator_view`. This corresponds to the `D3D11_CREATE_DEVICE_DISABLE_GPU_TIMEOUT` flag for Direct3D device creation and is used to indicate if the operating system should allow workloads that take more than 2 seconds to execute without resetting the device per the Windows timeout detection and recovery mechanism. Use of this flag is recommended if you need to perform time consuming tasks on the `accelerator_view`.

`_Qmode`

The `queuing_mode` to be used for the newly created `accelerator_view`. This parameter has a default value of `queuing_mode_automatic`.

## Return Value

The `accelerator_view` object created from the passed Direct3D device interface.

## Remarks

This function creates a new `accelerator_view` object from an existing pointer to a Direct3D device interface. If the function call succeeds, the reference count of the parameter is incremented by means of an `AddRef` call to the interface. You can safely release the object when it is no longer required in your DirectX code. If the method call fails, a `runtime_exception` is thrown.

The `accelerator_view` object that you create by using this function is thread safe. You must synchronize concurrent use of the `accelerator_view` object. Unsynchronized concurrent usage of the `accelerator_view` object and the raw `ID3D11Device` interface causes undefined behavior.

The C++ AMP runtime provides detailed error information in debug mode by using the D3D Debug layer if you use the `D3D11_CREATE_DEVICE_DEBUG` flag.

## d3d\_access\_lock

Acquire a lock on an `accelerator_view` for the purpose of safely performing D3D operations on resources shared with the `accelerator_view`. The `accelerator_view` and all C++ AMP resources associated with this `accelerator_view` internally take this lock when performing operations and will block while another thread holds the D3D access lock. This lock is non-recursive: It is undefined behavior to call this function from a thread that already holds the lock. It is undefined behavior to perform operations on the `accelerator_view` or any data container associated with the `accelerator_view` from the thread that holds the D3D access lock. See also `scoped_d3d_access_lock`, a RAII-style class for a scope-based D3D access lock.

```
void __cdecl d3d_access_lock(accelerator_view& _Av);
```

### Parameters

`_Av`

The `accelerator_view` to lock.

## d3d\_access\_try\_lock

Attempt to acquire the D3D access lock on an `accelerator_view` without blocking.

```
bool __cdecl d3d_access_try_lock(accelerator_view& _Av);
```

## Parameters

*\_Av*

The `accelerator_view` to lock.

## Return Value

true if the lock was acquired, or false if it is currently held by another thread.

# d3d\_access\_unlock

Release the D3D access lock on the given `accelerator_view`. If the calling thread does not hold the lock on the `accelerator_view` the results are undefined.

```
void __cdecl d3d_access_unlock(accelerator_view& _Av);
```

## Parameters

*\_Av*

The `accelerator_view` for which the lock is to be released.

# firstbithigh

Gets the location of the first set bit in `_X`, beginning with the highest-order bit and moving towards the lowest-order bit.

```
inline int firstbithigh(int _X) restrict(amp);
```

## Parameters

*\_X*

Integer value

## Return Value

The location of the first set bit

# firstbitlow

Gets the location of the first set bit in `_X`, beginning with the lowest-order bit and working toward the highest-order bit.

```
inline int firstbitlow(int _X) restrict(amp);
```

## Parameters

*\_X*

Integer value

## Return Value

Returns The location of the first set bit

# get\_buffer

Get the Direct3D buffer interface underlying the specified array.

```
template<
    typename value_type,
    int _Rank
>
IUnknown *get_buffer(
    const array<value_type, _Rank>& _Array) ;
```

### Parameters

*value\_type*

The type of elements in the array.

*\_Rank*

The rank of the array.

*\_Array*

An array on a Direct3D accelerator\_view for which the underlying Direct3D buffer interface is returned.

### Return Value

The IUnknown interface pointer corresponding to the Direct3D buffer underlying the array.

## get\_device

Get the D3D device interface underlying a accelerator\_view.

```
IUnknown* get_device(const accelerator_view Av);
```

### Parameters

*Av*

The D3D accelerator\_view for which the underlying D3D device interface is returned.

### Return value

The `IUnknown` interface pointer of the D3D device underlying the accelerator\_view.

## imax

Determine the maximum numeric value of the arguments

```
inline int imax(
    int _X,
    int _Y) restrict(amp);
```

### Parameters

*\_X*

Integer value

*\_Y*

Integer value

### Return Value

Return the maximum numeric value of the arguments

## imin

Determine the minimum numeric value of the arguments

```
inline int imin(  
    int _X,  
    int _Y) restrict(amp);
```

### Parameters

*\_X*

Integer value

*\_Y*

Integer value

### Return Value

Return the minimum numeric value of the arguments

## is\_timeout\_disabled

Returns a boolean flag indicating if timeout is disabled for the specified *accelerator\_view*. This corresponds to the `D3D11_CREATE_DEVICE_DISABLE_GPU_TIMEOUT` flag for Direct3D device creation.

```
bool __cdecl is_timeout_disabled(const accelerator_view& _Accelerator_view);
```

### Parameters

*\_Accelerator\_view*

The *accelerator\_view* for which the timeout disabled setting is to be queried.

### Return Value

A boolean flag indicating if timeout is disabled for the specified *accelerator\_view*.

## mad

Computes the product of the first and second specified argument, then adds the third specified argument.

```
inline float mad(  
    float _X,  
    float _Y,  
    float _Z) restrict(amp);  
  
inline double mad(  
    double _X,  
    double _Y,  
    double _Z) restrict(amp);  
  
inline int mad(  
    int _X,  
    int _Y,  
    int _Z) restrict(amp);  
  
inline unsigned int mad(  
    unsigned int _X,  
    unsigned int _Y,  
    unsigned int _Z) restrict(amp);
```

### Parameters

*\_X*

The first specified argument.

*\_Y*

The second specified argument.

*\_Z*

The third specified argument.

### Return Value

The result of  $\_X * \_Y + \_Z$ .

## make\_array

Create an array from a Direct3D buffer interface pointer.

```
template<
    typename value_type,
    int _Rank
>
array<value_type, _Rank> make_array(
    const extent<_Rank>& _Extent,
    const Concurrency::accelerator_view& _Rv,
    IUnknown* _D3D_buffer) ;
```

### Parameters

*value\_type*

The element type of the array to be created.

*\_Rank*

The rank of the array to be created.

*\_Extent*

An extent that describes the shape of the array aggregate.

*\_Rv*

A D3D accelerator view on which the array is to be created.

*\_D3D\_buffer*

IUnknown interface pointer of the D3D buffer to create the array from.

### Return Value

An array created using the provided Direct3D buffer.

## noise

Generates a random value using the Perlin noise algorithm

```
inline float noise(float _X) restrict(amp);
```

### Parameters

*\_X*

Floating-point value from which to generate Perlin noise

### Return Value

Returns The Perlin noise value within a range between -1 and 1

## radians

Converts `_X` from degrees to radians

```
inline float radians(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns `_X` converted from degrees to radians

## rcp

Computes the reciprocal of the specified argument by using a fast approximation.

```
inline float rcp(float _X) restrict(amp);  
  
inline double rcp(double _X) restrict(amp);
```

#### Parameters

`_X`

The value for which to compute the reciprocal.

#### Return Value

The reciprocal of the specified argument.

## reversebits

Reverses the order of the bits in `_X`

```
inline unsigned int reversebits(unsigned int _X) restrict(amp);
```

#### Parameters

`_X`

Unsigned integer value

#### Return Value

Returns the value with the bit order reversed in `_X`

## saturate

Clamps `_X` within the range of 0 to 1

```
inline float saturate(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns `_X` clamped within the range of 0 to 1

# sign

Determines the sign of the specified argument.

```
inline int sign(int _X) restrict(amp);
```

## Parameters

*\_X*

Integer value

## Return Value

The sign of the argument.

# smoothstep

Returns a smooth Hermite interpolation between 0 and 1, if *\_X* is in the range [*\_Min*, *\_Max*].

```
inline float smoothstep(  
    float _Min,  
    float _Max,  
    float _X) restrict(amp);
```

## Parameters

*\_Min*

Floating-point value

*\_Max*

Floating-point value

*\_X*

Floating-point value

## Return Value

Returns 0 if *\_X* is less than *\_Min*; 1 if *\_X* is greater than *\_Max*; otherwise, a value between 0 and 1 if *\_X* is in the range [*\_Min*, *\_Max*]

# step

Compares two values, returning 0 or 1 based on which value is greater

```
inline float step(  
    float _Y,  
    float _X) restrict(amp);
```

## Parameters

*\_Y*

Floating-point value

*\_X*

Floating-point value

## Return Value

Returns 1 if the *\_X* is greater than or equal to *\_Y*; otherwise, 0

# umax

Determine the maximum numeric value of the arguments

```
inline unsigned int umax(  
    unsigned int _X,  
    unsigned int _Y) restrict(amp);
```

## Parameters

`_X`

Integer value

`_Y`

Integer value

## Return Value

Return the maximum numeric value of the arguments

# umin

Determine the minimum numeric value of the arguments

```
inline unsigned int umin(  
    unsigned int _X,  
    unsigned int _Y) restrict(amp);
```

## Parameters

`_X`

Integer value

`_Y`

Integer value

## Return Value

Return the minimum numeric value of the arguments

## See also

[Concurrency::direct3d Namespace](#)

# adopt\_d3d\_access\_lock\_t Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

Tag type to indicate the D3D access lock should be adopted rather than acquired.

## Syntax

```
struct adopt_d3d_access_lock_t;
```

## Members

## Inheritance Hierarchy

```
adopt_d3d_access_lock_t
```

## Requirements

**Header:** amprt.h

**Namespace:** concurrency::direct3d

## See also

[Concurrency::direct3d Namespace](#)

# scoped\_d3d\_access\_lock Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

RAll wrapper for a D3D access lock on an `accelerator_view` object.

## Syntax

```
class scoped_d3d_access_lock;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">scoped_d3d_access_lock Constructor</a>	Overloaded. Constructs a <code>scoped_d3d_access_lock</code> object. The lock is released when this object goes out of scope.
<a href="#">~scoped_d3d_access_lock Destructor</a>	Releases the D3D access lock on the associated <code>accelerator_view</code> object.

### Public Operators

NAME	DESCRIPTION
<a href="#">operator=</a>	Takes ownership of a lock from another <code>scoped_d3d_access_lock</code> .

## Inheritance Hierarchy

```
scoped_d3d_access_lock
```

## Requirements

**Header:** `amprt.h`

**Namespace:** `concurrency::direct3d`

## scoped\_d3d\_access\_lock

Constructs a `scoped_d3d_access_lock` object. The lock is released when this object goes out of scope.

```
explicit scoped_d3d_access_lock(// [1] constructor
    accelerator_view& _Av);

explicit scoped_d3d_access_lock(// [2] constructor
    accelerator_view& _Av,
    adopt_d3d_access_lock_t _T);

scoped_d3d_access_lock(// [3] move constructor
    scoped_d3d_access_lock&& _Other);
```

## Parameters

*\_Av*

The `accelerator_view` for the lock to adopt.

*\_T*

The `adopt_d3d_access_lock_t` object.

*\_Other*

The `scoped_d3d_access_lock` object from which to move an existing lock.

## Construction

[1] Constructor Acquires a D3D access lock on the given `accelerator_view` object. Construction blocks until the lock is acquired.

[2] Constructor Adopt a D3D access lock from the given `accelerator_view` object.

[3] Move Constructor Takes an existing D3D access lock from another `scoped_d3d_access_lock` object. Construction does not block.

## ~scoped\_d3d\_access\_lock

Releases the D3D access lock on the associated `accelerator_view` object.

```
~scoped_d3d_access_lock();
```

## operator=

Takes ownership of a D3D access lock from another `scoped_d3d_access_lock` object, releasing the previous lock.

```
scoped_d3d_access_lock& operator= (scoped_d3d_access_lock&& _Other);
```

## Parameters

*\_Other*

The `accelerator_view` from which to move the D3D access lock.

## Return Value

A reference to this `scoped_accelerator_view_lock`.

## See also

[Concurrency::direct3d Namespace](#)

# Concurrency::fast\_math Namespace

3/4/2019 • 2 minutes to read • [Edit Online](#)

Functions in the `fast_math` namespace have lower accuracy, support only single-precision (`float`), and call the DirectX intrinsics. There are two versions of each function, for example `cos` and `cosf`. Both versions take and return a `float`, but each calls the same DirectX intrinsic.

## Syntax

```
namespace fast_math;
```

## Members

### Functions

NAME	DESCRIPTION
<a href="#">cos</a>	Calculates the arccosine of the argument
<a href="#">cosf</a>	Calculates the arccosine of the argument
<a href="#">asin</a>	Calculates the arcsine of the argument
<a href="#">asinf</a>	Calculates the arcsine of the argument
<a href="#">atan</a>	Calculates the arctangent of the argument
<a href="#">atan2</a>	Calculates the arctangent of $_Y/_X$
<a href="#">atan2f</a>	Calculates the arctangent of $_Y/_X$
<a href="#">atanf</a>	Calculates the arctangent of the argument
<a href="#">ceil</a>	Calculates the ceiling of the argument
<a href="#">ceilf</a>	Calculates the ceiling of the argument
<a href="#">cos</a>	Calculates the cosine of the argument
<a href="#">cosf</a>	Calculates the cosine of the argument
<a href="#">cosh</a>	Calculates the hyperbolic cosine value of the argument
<a href="#">coshf</a>	Calculates the hyperbolic cosine value of the argument
<a href="#">exp</a>	Calculates the base-e exponential of the argument
<a href="#">exp2</a>	Calculates the base-2 exponential of the argument

NAME	DESCRIPTION
<a href="#">exp2f</a>	Calculates the base-2 exponential of the argument
<a href="#">expf</a>	Calculates the base-e exponential of the argument
<a href="#">fabs</a>	Returns the absolute value of the argument
<a href="#">fabsf</a>	Returns the absolute value of the argument
<a href="#">floor</a>	Calculates the floor of the argument
<a href="#">floorf</a>	Calculates the floor of the argument
<a href="#">fmax</a>	Determine the maximum numeric value of the arguments
<a href="#">fmaxf</a>	Determine the maximum numeric value of the arguments
<a href="#">fmin</a>	Determine the minimum numeric value of the arguments
<a href="#">fminf</a>	Determine the minimum numeric value of the arguments
<a href="#">fmod</a>	Calculates the floating-point remainder of <code>_X/_Y</code>
<a href="#">fmodf</a>	Calculates the floating-point remainder of <code>_X/_Y</code>
<a href="#">frexp</a>	Gets the mantissa and exponent of <code>_X</code>
<a href="#">frexpf</a>	Gets the mantissa and exponent of <code>_X</code>
<a href="#">isfinite</a>	Determines whether the argument has a finite value
<a href="#">isinf</a>	Determines whether the argument is an infinity
<a href="#">isnan</a>	Determines whether the argument is a NaN
<a href="#">ldexp</a>	Computes a real number from the mantissa and exponent
<a href="#">ldexpf</a>	Computes a real number from the mantissa and exponent
<a href="#">log</a>	Calculates the base-e logarithm of the argument
<a href="#">log10</a>	Calculates the base-10 logarithm of the argument
<a href="#">log10f</a>	Calculates the base-10 logarithm of the argument
<a href="#">log2</a>	Calculates the base-2 logarithm of the argument
<a href="#">log2f</a>	Calculates the base-2 logarithm of the argument
<a href="#">logf</a>	Calculates the base-e logarithm of the argument

NAME	DESCRIPTION
<code>modf</code>	Splits <code>_X</code> into fractional and integer parts.
<code>modff</code>	Splits <code>_X</code> into fractional and integer parts.
<code>pow</code>	Calculates <code>_X</code> raised to the power of <code>_Y</code>
<code>powf</code>	Calculates <code>_X</code> raised to the power of <code>_Y</code>
<code>round</code>	Rounds <code>_X</code> to the nearest integer
<code>roundf</code>	Rounds <code>_X</code> to the nearest integer
<code>rsqrt</code>	Returns the reciprocal of the square root of the argument
<code>rsqrtf</code>	Returns the reciprocal of the square root of the argument
<code>signbit</code>	Returns the sign of the argument
<code>signbitf</code>	Returns the sign of the argument
<code>sin</code>	Calculates the sine value of the argument
<code>sincos</code>	Calculates sine and cosine value of <code>_X</code>
<code>sincosf</code>	Calculates sine and cosine value of <code>_X</code>
<code>sinf</code>	Calculates the sine value of the argument
<code>sinh</code>	Calculates the hyperbolic sine value of the argument
<code>sinhf</code>	Calculates the hyperbolic sine value of the argument
<code>sqrt</code>	Calculates the square root of the argument
<code>sqrtf</code>	Calculates the square root of the argument
<code>tan</code>	Calculates the tangent value of the argument
<code>tanf</code>	Calculates the tangent value of the argument
<code>tanh</code>	Calculates the hyperbolic tangent value of the argument
<code>tanhf</code>	Calculates the hyperbolic tangent value of the argument
<code>trunc</code>	Truncates the argument to the integer component
<code>truncf</code>	Truncates the argument to the integer component

## Requirements

**Header:** amp\_math.h

**Namespace:** Concurrency::fast\_math

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# Concurrency::fast\_math namespace functions

3/4/2019 • 9 minutes to read • [Edit Online](#)

acos	acosf	asin
asinf	atan	atan2
atan2f	atanf	ceil
ceilf	cos	cosf
cosh	coshf	exp
exp2	exp2f	expf
fabs	fabsf	floor
floorf	fmax	fmaxf
fmin	fminf	fmod
fmodf	frexp	frexpf
isfinite	isinf	isnan
ldexp	ldexpf	log
log10	log10f	log2
log2f	logf	modf
modff	pow	powf
round	roundf	rsqrt
rsqrtf	signbit	signbitf
sin	sincos	sincosf
sinf	sinh	sinhf
sqrt	sqrtf	tan
tanf	tanh	tanhf
trunc	truncf	

## acos

Calculates the arccosine of the argument

```
inline float acos(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the arccosine value of the argument

## acosf

Calculates the arccosine of the argument

```
inline float acosf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the arccosine value of the argument

## asin

Calculates the arcsine of the argument

```
inline float asin(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the arcsine value of the argument

## asinf

Calculates the arcsine of the argument

```
inline float asinf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the arcsine value of the argument

# atan

Calculates the arctangent of the argument

```
inline float atan(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the arctangent value of the argument

# atan2

Calculates the arctangent of  $_Y/_X$

```
inline float atan2(  
    float _Y,  
    float _X) restrict(amp);
```

## Parameters

`_Y`

Floating-point value

`_X`

Floating-point value

## Return Value

Returns the arctangent value of  $_Y/_X$

# atan2f

Calculates the arctangent of  $_Y/_X$

```
inline float atan2f(  
    float _Y,  
    float _X) restrict(amp);
```

## Parameters

`_Y`

Floating-point value

`_X`

Floating-point value

## Return Value

Returns the arctangent value of  $_Y/_X$

# atanf

Calculates the arctangent of the argument

```
inline float atanf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the arctangent value of the argument

## ceil

Calculates the ceiling of the argument

```
inline float ceil(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the ceiling of the argument

## ceilf

Calculates the ceiling of the argument

```
inline float ceilf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the ceiling of the argument

## cosf

Calculates the cosine of the argument

```
inline float cosf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the cosine value of the argument

## coshf

Calculates the hyperbolic cosine value of the argument

```
inline float coshf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the hyperbolic cosine value of the argument

## COS

Calculates the cosine of the argument

```
inline float cos(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the cosine value of the argument

## cosh

Calculates the hyperbolic cosine value of the argument

```
inline float cosh(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the hyperbolic cosine value of the argument

## exp

Calculates the base-e exponential of the argument

```
inline float exp(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the base-e exponential of the argument

## exp2

Calculates the base-2 exponential of the argument

```
inline float exp2(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the base-2 exponential of the argument

## exp2f

Calculates the base-2 exponential of the argument

```
inline float exp2f(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the base-2 exponential of the argument

## expf

Calculates the base-e exponential of the argument

```
inline float expf(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the base-e exponential of the argument

## fabs

Returns the absolute value of the argument

```
inline float fabs(float _X) restrict(amp);
```

#### Parameters

`_X`

Integer value

#### Return Value

Returns the absolute value of the argument

# fabsf

Returns the absolute value of the argument

```
inline float fabsf(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the absolute value of the argument

# floor

Calculates the floor of the argument

```
inline float floor(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the floor of the argument

# floorf

Calculates the floor of the argument

```
inline float floorf(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the floor of the argument

# fmax

Determine the maximum numeric value of the arguments

```
inline float max(  
    float _X,  
    float _Y) restrict(amp);
```

## Parameters

`_X`

Integer value

`_Y`

Integer value

### **Return Value**

Return the maximum numeric value of the arguments

## fmaxf

Determine the maximum numeric value of the arguments

```
inline float fmaxf(  
    float _X,  
    float _Y) restrict(amp);
```

### **Parameters**

\_X

Floating-point value

\_Y

Floating-point value

### **Return Value**

Return the maximum numeric value of the arguments

## fmin

Determine the minimum numeric value of the arguments

```
inline float min(  
    float _X,  
    float _Y) restrict(amp);
```

### **Parameters**

\_X

Integer value

\_Y

Integer value

### **Return Value**

Return the minimum numeric value of the arguments

## fminf

Determine the minimum numeric value of the arguments

```
inline float fminf(  
    float _X,  
    float _Y) restrict(amp);
```

### **Parameters**

\_X

Floating-point value

\_Y

Floating-point value

### Return Value

Return the minimum numeric value of the arguments

## fmod

Calculates the floating-point remainder of  $_X/_Y$

```
inline float fmod(  
    float _X,  
    float _Y) restrict(amp);
```

### Parameters

$_X$

Floating-point value

$_Y$

Floating-point value

### Return Value

Returns the floating-point remainder of  $_X/_Y$

## fmodf

Calculates the floating-point remainder of  $_X/_Y$ .

```
inline float fmodf(  
    float _X,  
    float _Y) restrict(amp);
```

### Parameters

$_X$

Floating-point value

$_Y$

Floating-point value

### Return Value

Returns the floating-point remainder of  $_X/_Y$

## frexp

Gets the mantissa and exponent of  $_X$

```
inline float frexp(  
    float _X,  
    _Out_ int* _Exp) restrict(amp);
```

### Parameters

$_X$

Floating-point value

$_Exp$

Returns the integer exponent of `_X` in floating-point value

### Return Value

Returns the mantissa `_X`

## frexpf

Gets the mantissa and exponent of `_X`

```
inline float frexpf(  
    float _X,  
    _Out_ int* _Exp) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Exp`

Returns the integer exponent of `_X` in floating-point value

### Return Value

Returns the mantissa `_X`

## isfinite

Determines whether the argument has a finite value

```
inline int isfinite(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns a nonzero value if and only if the argument has a finite value

## isinf

Determines whether the argument is an infinity

```
inline int isinf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns a nonzero value if and only if the argument has an infinite value

## isnan

Determines whether the argument is a NaN

```
inline int isnan(float _X) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

### Return Value

Returns a nonzero value if and only if the argument has a NaN value

## ldexp

Computes a real number from the mantissa and exponent

```
inline float ldexp(  
    float _X,  
    int _Exp) restrict(amp);
```

### Parameters

*\_X*

Floating-point value, mantissa

*\_Exp*

Integer exponent

### Return Value

Returns  $_X * 2^{\_Exp}$

## ldexpf

Computes a real number from the mantissa and exponent

```
inline float ldexpf(  
    float _X,  
    int _Exp) restrict(amp);
```

### Parameters

*\_X*

Floating-point value, mantissa

*\_Exp*

Integer exponent

### Return Value

Returns  $_X * 2^{\_Exp}$

## log

Calculates the base-e logarithm of the argument

```
inline float log(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### **Return Value**

Returns the base-e logarithm of the argument

## log10

Calculates the base-10 logarithm of the argument

```
inline float log10(float _X) restrict(amp);
```

### **Parameters**

`_X`

Floating-point value

### **Return Value**

Returns the base-10 logarithm of the argument

## log10f

Calculates the base-10 logarithm of the argument

```
inline float log10f(float _X) restrict(amp);
```

### **Parameters**

`_X`

Floating-point value

### **Return Value**

Returns the base-10 logarithm of the argument

## log2

Calculates the base-2 logarithm of the argument

```
inline float log2(float _X) restrict(amp);
```

### **Parameters**

`_X`

Floating-point value

### **Return Value**

Returns the base-2 logarithm of the argument

## log2f

Calculates the base-2 logarithm of the argument

```
inline float log2f(float _X) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

### Return Value

Returns the base-10 logarithm of the argument

## logf

Calculates the base-e logarithm of the argument

```
inline float logf(float _X) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

### Return Value

Returns the base-e logarithm of the argument

## modf

Splits *\_X* into fractional and integer parts.

```
inline float modf(  
    float _X,  
    float* _Ip) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

*\_Ip*

Receives integer part of the value

### Return Value

Returns the signed fractional portion of *\_X*

## modff

Splits *\_X* into fractional and integer parts.

```
inline float modff(  
    float _X,  
    float* _Ip) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

*\_Ip*

Receives integer part of the value

### Return Value

Returns the signed fractional portion of `_X`

## pow

Calculates `_X` raised to the power of `_Y`

```
inline float pow(  
    float _X,  
    float _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value, base

`_Y`

Floating-point value, exponent

### Return Value

Returns the value of `_X` raised to the power of `_Y`

## powf

Calculates `_X` raised to the power of `_Y`

```
inline float powf(  
    float _X,  
    float _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value, base

`_Y`

Floating-point value, exponent

### Return Value

## round

Rounds `_X` to the nearest integer

```
inline float round(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the nearest integer of `_X`

## roundf

Rounds `_X` to the nearest integer

```
inline float roundf(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the nearest integer of `_X`

## rsqrt

Returns the reciprocal of the square root of the argument

```
inline float rsqrt(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the reciprocal of the square root of the argument

## rsqrtf

Returns the reciprocal of the square root of the argument

```
inline float rsqrtf(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the reciprocal of the square root of the argument

## signbit

Determines whether the sign of `_X` is negative

```
inline int signbit(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns a nonzero value if and only if the sign of `_X` is negative

# signbitf

Determines whether the sign of `_X` is negative

```
inline int signbitf(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns a nonzero value if and only if the sign of `_X` is negative

# sin

Calculates the sine value of the argument

```
inline float sin(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the sine value of the argument

# sinf

Calculates the sine value of the argument

```
inline float sinf(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the sine value of the argument

# sincos

Calculates sine and cosine value of `_X`

```
inline void sincos(  
    float _X,  
    float* _S,  
    float* _C) restrict(amp);
```

## Parameters

`_X`

Floating-point value

`_S`

Returns the sine value of `_X`

`_C`

Returns the cosine value of `_X`

## sincosf

Calculates sine and cosine value of `_X`

```
inline void sincosf(  
    float _X,  
    float* _S,  
    float* _C) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_S`

Returns the sine value of `_X`

`_C`

Returns the cosine value of `_X`

## sinh

Calculates the hyperbolic sine value of the argument

```
inline float sinh(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the hyperbolic sine value of the argument

## sinhf

Calculates the hyperbolic sine value of the argument

```
inline float sinhf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the hyperbolic sine value of the argument

## sqrt

Calculates the square root of the argument

```
inline float sqrt(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the square root of the argument

## sqrtf

Calculates the square root of the argument

```
inline float sqrtf(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the square root of the argument

## tan

Calculates the tangent value of the argument

```
inline float tan(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the tangent value of the argument

## tanf

Calculates the tangent value of the argument

```
inline float tanf(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the tangent value of the argument

# tanh

Calculates the hyperbolic tangent value of the argument

```
inline float tanh(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the hyperbolic tangent value of the argument

# tanhf

Calculates the hyperbolic tangent value of the argument

```
inline float tanhf(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the hyperbolic tangent value of the argument

# trunc

Truncates the argument to the integer component

```
inline float trunc(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the integer component of the argument

# truncf

Truncates the argument to the integer component

```
inline float truncf(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the integer component of the argument

# Requirements

**Header:** amp\_math.h **Namespace:** Concurrency::fast\_math

## See also

[Concurrency::fast\\_math Namespace](#)

# Concurrency::graphics Namespace

3/4/2019 • 2 minutes to read • [Edit Online](#)

The graphics namespace provides types and functions that are designed for graphics programming.

## Syntax

```
namespace graphics;
```

## Members

### Namespaces

NAME	DESCRIPTION
<a href="#">Concurrency::graphics::direct3d Namespace</a>	Provides functions for Direct3D interop.

### Typedefs

NAME	DESCRIPTION
<code>uint</code>	The element type for <a href="#">uint_2 Class</a> , <a href="#">uint_3 Class</a> , and <a href="#">uint_4 Class</a> . Defined as <code>typedef unsigned int uint;</code> .

### Enumerations

NAME	DESCRIPTION
<a href="#">address_mode Enumeration</a> .	Specifies address modes supported for texture sampling.
<a href="#">filter_mode Enumeration</a>	Specifies filter modes supported for texture sampling.

### Classes

NAME	DESCRIPTION
<a href="#">texture Class</a>	A texture is a data aggregate on an <code>accelerator_view</code> in the extent domain. It is a collection of variables, one for each element in an extent domain. Each variable holds a value corresponding to C++ primitive type (unsigned int, int, float, double), or scalar type norm, or unorm (defined in <code>concurrency::graphics</code> ), or eligible short vector types defined in <code>concurrency::graphics</code> .
<a href="#">writeonly_texture_view Class</a>	A <code>writeonly_texture_view</code> provides writeonly access to a texture.
<a href="#">double_2 Class</a>	Represents a short vector of 2 <code>double</code> values.
<a href="#">double_3 Class</a>	Represents a short vector of 3 <code>double</code> values.

NAME	DESCRIPTION
<a href="#">double_4 Class</a>	Represents a short vector of 4 <code>double</code> values.
<a href="#">float_2 Class</a>	Represents a short vector of 2 <code>float</code> values.
<a href="#">float_3 Class</a>	Represents a short vector of 3 <code>float</code> values.
<a href="#">float_4 Class</a>	Represents a short vector of 4 <code>float</code> values.
<a href="#">int_2 Class</a>	Represents a short vector of 2 <code>int</code> values.
<a href="#">int_3 Class</a>	Represents a short vector of 3 <code>int</code> values.
<a href="#">int_4 Class</a>	Represents a short vector of 4 <code>int</code> values.
<a href="#">norm_2 Class</a>	Represents a short vector of 2 <code>norm</code> values.
<a href="#">norm_3 Class</a>	Represents a short vector of 3 <code>norm</code> values.
<a href="#">norm_4 Class</a>	Represents a short vector of 4 <code>norm</code> values.
<a href="#">uint_2 Class</a>	Represents a short vector of 2 <code>uint</code> values.
<a href="#">uint_3 Class</a>	Represents a short vector of 3 <code>uint</code> values.
<a href="#">uint_4 Class</a>	Represents a short vector of 4 <code>uint</code> values.
<a href="#">unorm_2 Class</a>	Represents a short vector of 2 <code>unorm</code> values.
<a href="#">unorm_3 Class</a>	Represents a short vector of 3 <code>unorm</code> values.
<a href="#">unorm_4 Class</a>	Represents a short vector of 4 <code>unorm</code> values.
<a href="#">sampler Class</a>	Represents the sampler configuration used for texture sampling.
<a href="#">short_vector Structure</a>	Provides a basic implementation of a short vector of values.
<a href="#">short_vector_traits Structure</a>	Provides for retrieval of the length and type of a short vector.
<a href="#">texture_view Class</a>	Provides read access and write access to a texture.

## Functions

NAME	DESCRIPTION
<a href="#">copy</a>	Overloaded. Copies the contents of the source texture into the destination host buffer.

NAME	DESCRIPTION
<a href="#">copy_async</a>	Overloaded. Asynchronously copies the contents of the source texture into the destination host buffer.

## Requirements

**Header:** amp\_graphics.h

**Namespace:** Concurrency

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# Concurrency::graphics::direct3d Namespace

3/4/2019 • 2 minutes to read • [Edit Online](#)

Provides the [get\\_texture](#) and [make\\_texture](#) methods.

## Syntax

```
namespace direct3d;
```

## Members

### Functions

NAME	DESCRIPTION
<a href="#">get_sampler</a>	Get the Direct3D sampler state interface on the given accelerator view that represents the specified sampler object.
<a href="#">get_texture</a>	Gets the Direct3D texture interface underlying the specified <a href="#">texture</a> object.
<a href="#">make_sampler</a>	Create a sampler from a Direct3D sampler state interface pointer.
<a href="#">make_texture</a>	Creates a <a href="#">texture</a> object by using the specified parameters.
<a href="#">msad4</a>	Compares a 4-byte reference value and an 8-byte source value and accumulates a vector of 4 sums.

## Requirements

**Header:** amp\_graphics.h

**Namespace:** Concurrency::graphics

## See also

[Concurrency::graphics Namespace](#)

# Concurrency::graphics::direct3d namespace functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

<a href="#">get_sampler</a>	<a href="#">get_texture</a>	<a href="#">make_sampler</a>
<a href="#">make_texture</a>	<a href="#">msad4</a>	

## get\_sampler

Get the D3D sampler state interface on the given accelerator view that represents the specified sampler object.

```
IUnknown* get_sampler(  
    const Concurrency::accelerator_view& _Av,  
    const sampler& _Sampler) restrict(cpu);
```

### Parameters

*\_Av*

A D3D accelerator view on which the D3D sampler state is to be created.

*\_Sampler*

A sampler object for which the underlying D3D sampler state interface is created.

### Return Value

The IUnknown interface pointer corresponding to the D3D sampler state that represents the given sampler.

## get\_texture

Gets the Direct3D texture interface underlying the specified [texture](#) object.

```
template<  
    typename value_type,  
    int _Rank  
>  
_Ret_ IUnknown *get_texture(  
    const texture<value_type, _Rank>& _Texture) restrict(cpu);  
  
template<  
    typename value_type,  
    int _Rank  
>  
_Ret_ IUnknown *get_texture(  
    const writeonly_texture_view<value_type, _Rank>& _Texture) restrict(cpu);  
  
template<  
    typename value_type,  
    int _Rank  
>  
_Ret_ IUnknown *get_texture(  
    const texture_view<value_type, _Rank>& _Texture) restrict(cpu);
```

### Parameters

*value\_type*

The element type of the texture.

*\_Rank*

The rank of the texture.

*\_Texture*

A texture or texture view associated with the `accelerator_view` for which the underlying Direct3D texture interface is returned.

### Return Value

The IUnknown interface pointer corresponding to the Direct3D texture underlying the texture.

## make\_sampler

Create a sampler from a D3D sampler state interface pointer.

```
sampler make_sampler(_In_ IUnknown* _D3D_sampler) restrict(amp);
```

### Parameters

*\_D3D\_sampler*

IUnknown interface pointer of the D3D sampler state to create the sampler from.

### Return Value

A sampler represents the provided D3D sampler state.

## make\_texture

Creates a [texture](#) object by using the specified parameters.

```
template<
    typename value_type,
    int _Rank
>
texture<value_type, _Rank> make_texture(
    const Concurrency::accelerator_view& _Av,
    _In_ IUnknown* _D3D_texture,
    DXGI_FORMAT _View_format = DXGI_FORMAT_UNKNOWN) restrict(cpu);
```

### Parameters

*value\_type*

The type of the elements in the texture.

*\_Rank*

The rank of the texture.

*\_Av*

A D3D accelerator view on which the texture is to be created.

*\_D3D\_texture*

IUnknown interface pointer of the D3D texture to create the texture from.

*\_View\_format*

The DXGI format to use for views created from this texture. Pass `DXGI_FORMAT_UNKNOWN` (the default) to derive the format from the underlying format of `_D3D_texture` and the `value_type` of this template. The provided format must be compatible with the underlying format of `_D3D_texture`.

## Return Value

A texture using the provided D3D texture.

## msad4

Compares a 4-byte reference value and an 8-byte source value and accumulates a vector of 4 sums. Each sum corresponds to the masked sum of absolute differences of different byte alignments between the reference value and the source value.

```
inline uint4 msad4(  
    uint _Reference,  
    uint2 _Source,  
    uint4 _Accum) restrict(amp);
```

## Parameters

### *\_Reference*

The reference array of 4 bytes in one uint value

### *\_Source*

The source array of 8 bytes in a vector of two uint values.

### *\_Accum*

A vector of 4 values to be added to the masked sum of absolute differences of the different byte alignments between the reference value and the source value.

## Return Value

Returns a vector of 4 sums. Each sum corresponds to the masked sum of absolute differences of different byte alignments between the reference value and the source value.

## Requirements

**Header:** amp\_graphics.h

**Namespace:** Concurrency::graphics::direct3d

## See also

[Concurrency::graphics::direct3d Namespace](#)

# Concurrency::graphics namespace functions

3/4/2019 • 3 minutes to read • [Edit Online](#)

[copy](#)

[copy\\_async](#)

## copy Function (Concurrency::graphics Namespace)

Copies a source texture into a destination buffer, or copies a source buffer into a destination buffer. The general form of this function is `copy(src, dest)`.

```
template <
    typename _Src_type,
    typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture, void>::type
>
void copy (
    const _Src_type& _Src,
    _Out_ void* _Dst,
    unsigned int _Dst_byte_size);

template <
    typename _Src_type,
    typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture, void>::type
>
void copy(
    const _Src_type& _Src,
    const index<_Src_type::rank>& _Src_offset,
    const extent<_Src_type::rank>& _Copy_extent,
    _Out_ void* _Dst,
    unsigned int _Dst_byte_size);

template <
    typename _Dst_type,
    typename = typename std::enable_if<details::texture_traits<_Dst_type>::is_texture, void>::type
>
void copy(
    const void* _Src,
    unsigned int _Src_byte_size, _Dst_type& _Dst);

template <
    typename _Dst_type,
    typename = typename std::enable_if<details::texture_traits<_Dst_type>::is_texture, void>::type
>
void copy(
    const void* _Src,
    unsigned int _Src_byte_size,
    _Dst_type& _Dst,
    const index<_Dst_type::rank>& _Dst_offset,
    const extent<_Dst_type::rank>& _Copy_extent);

template <
    typename InputIterator,
    typename _Dst_type,
    typename = typename std::enable_if<details::texture_traits<_Dst_type>::is_texture, void>::type
>
void copy(InputIterator first, InputIterator last, _Dst_type& _Dst);

template <
    typename InputIterator,
```

```

typename _Dst_type,
typename = typename std::enable_if<details::texture_traits<_Dst_type>::is_texture, void>::type
>void copy(InputIterator first, InputIterator last, _Dst_type& _Dst,
const index<_Dst_type::rank>& _Dst_offset,
const extent<_Dst_type::rank>& _Copy_extent);

template <
typename _Src_type,
typename OutputIterator,
typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture&&
!details::texture_traits<OutputIterator>::is_texture, void>::type
>
void copy(
const _Src_type& _Src, OutputIterator _Dst);

template <
typename _Src_type,
typename OutputIterator,
typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture&&
!details::texture_traits<OutputIterator>::is_texture, void>::type
>
void copy (
const _Src_type& _Src,
const index<_Src_type::rank>& _Src_offset,
const extent<_Src_type::rank>& _Copy_extent, OutputIterator _Dst);

template <
typename _Src_type,
typename _Dst_type,
typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture&&
details::texture_traits<_Dst_type>::is_texture, void>::type
>
void copy (
const _Src_type& _Src, _Dst_type& _Dst);

template <
typename _Src_type,
typename _Dst_type,
typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture&&
details::texture_traits<_Dst_type>::is_texture,
void>::type
>
void copy (
const _Src_type& _Src,
const index<_Src_type::rank>& _Src_offset, _Dst_type& _Dst,
const index<_Dst_type::rank>& _Dst_offset,
const extent<_Src_type::rank>& _Copy_extent);

```

## Parameters

### *\_Copy\_extent*

The extent of the texture section to be copied.

### *\_Dst*

The object to copy to.

### *\_Dst\_byte\_size*

The number of bytes in the destination.

### *\_Dst\_type*

The type of the destination object.

### *\_Dst\_offset*

The offset into the destination at which to begin copying.

### *InputIterator*

The type of the input iterator.

### *OutputIterator*

The type of the output iterator.

### *\_Src*

To object to copy.

### *\_Src\_byte\_size*

The number of bytes in the source.

### *\_Src\_type*

The type of the source object.

### *\_Src\_offset*

The offset into the source from which to begin copying.

### *first*

A beginning iterator into the source container.

### *last*

An ending iterator into the source container.

## copy\_async Function (Concurrency::graphics Namespace)

Asynchronously copies a source texture into a destination buffer, or copies a source buffer into a destination buffer, and then returns a [completion\\_future](#) object that can be waited on. Data can't be copied when code is running on an accelerator. The general form of this function is `copy(src, dest)`.

```
template<
    typename _Src_type,
    typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture, void>::type
>
concurrency::completion_future copy_async(
    const _Src_type& _Src,
    _Out_ void* _Dst,
    unsigned int _Dst_byte_size);

template<
    typename _Src_type,
    typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture, void>::type
>
concurrency::completion_future copy_async(
    const _Src_type& _Src,
    const index<_Src_type::rank>& _Src_offset,
    const extent<_Src_type::rank>& _Copy_extent,
    _Out_ void* _Dst,
    unsigned int _Dst_byte_size);

template <
    typename _Dst_type,
    typename = typename std::enable_if<details::texture_traits<_Dst_type>::is_texture, void>::type
>
concurrency::completion_future copy_async(
    const void* _Src,
    unsigned int _Src_byte_size, _Dst_type& _Dst);

template <
    typename _Dst_type,
    typename = typename std::enable_if<details::texture_traits<_Dst_type>::is_texture, void>::type
>
concurrency::completion_future copy_async(
    const void* _Src,
    unsigned int _Src_byte_size, _Dst_type& _Dst
```

```

    unsigned int _Src_byte_size, _Dst_type& _Dst,
    const index<_Dst_type::rank>& _Dst_offset,
    const extent<_Dst_type::rank>& _Copy_extent);

template <
    typename InputIterator,
    typename _Dst_type,
    typename = typename std::enable_if<details::texture_traits<_Dst_type>::is_texture, void>::type
>
concurrency::completion_future copy_async(InputIterator first, InputIterator last, _Dst_type& _Dst);

template <
    typename InputIterator,
    typename _Dst_type,
    typename = typename std::enable_if<details::texture_traits<_Dst_type>::is_texture, void>::type
>
concurrency::completion_future copy_async(InputIterator first, InputIterator last, _Dst_type& _Dst,
    const index<_Dst_type::rank>& _Dst_offset,
    const extent<_Dst_type::rank>& _Copy_extent);

template <
    typename _Src_type,
    typename OutputIterator,
    typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture&&
!details::texture_traits<OutputIterator>::is_texture, void>::type
>
concurrency::completion_future copy_async(_Src_type& _Src, OutputIterator _Dst);

template <
    typename _Src_type,
    typename OutputIterator,
    typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture&&
!details::texture_traits<OutputIterator>::is_texture, void>::type
>
concurrency::completion_future copy_async(_Src_type& _Src,
    const index<_Src_type::rank>& _Src_offset,
    const extent<_Src_type::rank>& _Copy_extent,
    OutputIterator _Dst);

template <
    typename _Src_type,
    typename _Dst_type,
    typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture&&
details::texture_traits<_Dst_type>::is_texture, void>::type
>
concurrency::completion_future copy_async(_Src_type& _Src, _Dst_type& _Dst);

template <
    typename _Src_type,
    typename _Dst_type,
    typename = typename std::enable_if<details::texture_traits<_Src_type>::is_texture&&
details::texture_traits<_Dst_type>::is_texture, void>::type
>
concurrency::completion_future copy_async(_Src_type& _Src,
    const index<_Src_type::rank>& _Src_offset, _Dst_type &_Dst,
    const index<_Dst_type::rank>& _Dst_offset,
    const extent<_Src_type::rank>& _Copy_extent);

```

## Parameters

*\_Copy\_extent*

The extent of the texture section to be copied.

*\_Dst*

The object to copy to.

*\_Dst\_byte\_size*

The number of bytes in the destination.

*\_Dst\_type*

The type of the destination object.

*\_Dst\_offset*

The offset into the destination at which to begin copying.

*InputIterator*

The type of the input iterator.

*OutputIterator*

The type of the output iterator.

*\_Src*

The object to copy.

*\_Src\_byte\_size*

The number of bytes in the source.

*\_Src\_type*

The type of the source object.

*\_Src\_offset*

The offset into the source from which to begin copying.

*first*

A beginning iterator into the source container.

*last*

An ending iterator into the source container.

## Requirements

**Header:** `amp_graphics.h`

**Namespace:** `Concurrency::graphics`

## See also

[Concurrency::graphics Namespace](#)

# Concurrency::graphics namespace enums

3/4/2019 • 2 minutes to read • [Edit Online](#)

<a href="#">_mode Enumeration</a>	<a href="#">filter_mode Enumeration</a>
-----------------------------------	---

## address\_mode Enumeration

Enumeration type use to denote address modes supported for texture sampling.

```
enum address_mode;
```

## filter\_mode Enumeration

Enumeration type use to denote filter modes supported for texture sampling.

```
enum filter_mode;
```

## Requirements

**Header:** amp\_graphics.h **Namespace:** Concurrency::graphics

## See also

[Concurrency::graphics Namespace](#)

# double\_2 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represent a short vector of 2 double's.

## Syntax

```
class double_2;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">double_2 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>double_2::get_x</code>	
<code>double_2::get_xy</code>	
<code>double_2::get_y</code>	
<code>double_2::get_yx</code>	
<code>double_2::ref_g</code>	
<code>double_2::ref_r</code>	
<code>double_2::ref_x</code>	
<code>double_2::ref_y</code>	
<code>double_2::set_x</code>	
<code>double_2::set_xy</code>	
<code>double_2::set_y</code>	

NAME	DESCRIPTION
double_2::set_yx	

### Public Operators

NAME	DESCRIPTION
double_2::operator-	
double_2::operator--	
double_2::operator* =	
double_2::operator/=	
double_2::operator++	
double_2::operator+ =	
double_2::operator=	
double_2::operator- =	

### Public Constants

NAME	DESCRIPTION
double_2::size Constant	

### Public Data Members

NAME	DESCRIPTION
double_2::g	
double_2::gr	
double_2::r	
double_2::rg	
double_2::x	
double_2::xy	
double_2::y	
double_2::yx	

## Inheritance Hierarchy

double\_2

# Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## double\_2

Default constructor, initializes all elements with 0.

```
double_2() restrict(amp,
    cpu);

double_2(
    double _V0,
    double _V1) restrict(amp,
    cpu);

double_2(
    double _V) restrict(amp,
    cpu);

double_2(
    const double_2& _Other) restrict(amp,
    cpu);

explicit inline double_2(
    const uint_2& _Other) restrict(amp,
    cpu);

explicit inline double_2(
    const int_2& _Other) restrict(amp,
    cpu);

explicit inline double_2(
    const float_2& _Other) restrict(amp,
    cpu);

explicit inline double_2(
    const unorm_2& _Other) restrict(amp,
    cpu);

explicit inline double_2(
    const norm_2& _Other) restrict(amp,
    cpu);
```

### Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 2;
```

## See also

[Concurrency::graphics Namespace](#)

# double\_3 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of three doubles.

## Syntax

```
class double_3;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
value_type	

### Public Constructors

NAME	DESCRIPTION
<a href="#">double_3 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
double_3::get_x	
double_3::get_xy	
double_3::get_xyz	
double_3::get_xz	
double_3::get_xzy	
double_3::get_y	
double_3::get_yx	
double_3::get_yxz	
double_3::get_yz	
double_3::get_yzx	
double_3::get_z	

NAME	DESCRIPTION
double_3::get_zx	
double_3::get_zxy	
double_3::get_zy	
double_3::get_zyx	
double_3::ref_b	
double_3::ref_g	
double_3::ref_r	
double_3::ref_x	
double_3::ref_y	
double_3::ref_z	
double_3::set_x	
double_3::set_xy	
double_3::set_xyz	
double_3::set_xz	
double_3::set_xzy	
double_3::set_y	
double_3::set_yx	
double_3::set_yxz	
double_3::set_yz	
double_3::set_yzx	
double_3::set_z	
double_3::set_zx	
double_3::set_zxy	
double_3::set_zy	
double_3::set_zyx	

**Public Operators**

NAME	DESCRIPTION
double_3::operator-	
double_3::operator--	
double_3::operator* =	
double_3::operator/=	
double_3::operator++	
double_3::operator+ =	
double_3::operator=	
double_3::operator-=	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
double_3::b	
double_3::bg	
double_3::bgr	
double_3::br	
double_3::brg	
double_3::g	
double_3::gb	
double_3::gbr	
double_3::gr	
double_3::grb	
double_3::r	
double_3::rb	
double_3::rbg	

NAME	DESCRIPTION
double_3::rg	
double_3::rgb	
double_3::x	
double_3::xy	
double_3::xyz	
double_3::xz	
double_3::xzy	
double_3::y	
double_3::yx	
double_3::yz	
double_3::yz	
double_3::yzx	
double_3::z	
double_3::zx	
double_3::zxy	
double_3::zy	
double_3::zyx	

## Inheritance Hierarchy

`double_3`

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## double\_3

Default constructor, initializes all elements with 0.

```

double_3() restrict(amp,
    cpu);

double_3(
    double _V0,
    double _V1,
    double _V2) restrict(amp,
    cpu);

double_3(
    double _V) restrict(amp,
    cpu);

double_3(
    const double_3& _Other) restrict(amp,
    cpu);

explicit inline double_3(
    const uint_3& _Other) restrict(amp,
    cpu);

explicit inline double_3(
    const int_3& _Other) restrict(amp,
    cpu);

explicit inline double_3(
    const float_3& _Other) restrict(amp,
    cpu);

explicit inline double_3(
    const unorm_3& _Other) restrict(amp,
    cpu);

explicit inline double_3(
    const norm_3& _Other) restrict(amp,
    cpu);

```

## Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V2*

The value to initialize element 2.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 3;
```

## See also

[Concurrency::graphics Namespace](#)

# double\_4 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of four doubles.

## Syntax

```
class double_4;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">double_4 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>double_4::get_w</code>	
<code>double_4::get_wx</code>	
<code>double_4::get_wxy</code>	
<code>double_4::get_wxyz</code>	
<code>double_4::get_wxz</code>	
<code>double_4::get_wxzy</code>	
<code>double_4::get_wy</code>	
<code>double_4::get_wyx</code>	
<code>double_4::get_wyz</code>	
<code>double_4::get_wyz</code>	
<code>double_4::get_wyzx</code>	

NAME	DESCRIPTION
double_4::get_wz	
double_4::get_wzx	
double_4::get_wzxy	
double_4::get_wzy	
double_4::get_wzyx	
double_4::get_x	
double_4::get_xw	
double_4::get_xwy	
double_4::get_xwyz	
double_4::get_xwz	
double_4::get_xwzy	
double_4::get_xy	
double_4::get_xyw	
double_4::get_xywz	
double_4::get_xyz	
double_4::get_xyzw	
double_4::get_xz	
double_4::get_xzw	
double_4::get_xzwy	
double_4::get_xzy	
double_4::get_xzyw	
double_4::get_y	
double_4::get_yw	
double_4::get_ywx	
double_4::get_ywxz	

NAME	DESCRIPTION
double_4::get_ywz	
double_4::get_ywzx	
double_4::get_yx	
double_4::get_yxw	
double_4::get_yxwz	
double_4::get_yxz	
double_4::get_yxzw	
double_4::get_yz	
double_4::get_yzw	
double_4::get_yzwx	
double_4::get_yzx	
double_4::get_yzxw	
double_4::get_z	
double_4::get_zw	
double_4::get_zwx	
double_4::get_zwxy	
double_4::get_zwy	
double_4::get_zwyx	
double_4::get_zx	
double_4::get_zxw	
double_4::get_zxwy	
double_4::get_zxy	
double_4::get_zxyw	
double_4::get_zy	
double_4::get_zyw	

NAME	DESCRIPTION
double_4::get_zywx	
double_4::get_zyx	
double_4::get_zyxw	
double_4::ref_a	
double_4::ref_b	
double_4::ref_g	
double_4::ref_r	
double_4::ref_w	
double_4::ref_x	
double_4::ref_y	
double_4::ref_z	
double_4::set_w	
double_4::set_wx	
double_4::set_wxy	
double_4::set_wxyz	
double_4::set_wxz	
double_4::set_wxzy	
double_4::set_wy	
double_4::set_wyx	
double_4::set_wyxz	
double_4::set_wyz	
double_4::set_wyzx	
double_4::set_wz	
double_4::set_wzx	
double_4::set_wzxy	

NAME	DESCRIPTION
double_4::set_wzy	
double_4::set_wzyx	
double_4::set_x	
double_4::set_xw	
double_4::set_xwy	
double_4::set_xwyz	
double_4::set_xwz	
double_4::set_xwzy	
double_4::set_xy	
double_4::set_xyw	
double_4::set_xywz	
double_4::set_xyz	
double_4::set_xyzw	
double_4::set_xz	
double_4::set_xzw	
double_4::set_xzwy	
double_4::set_xzy	
double_4::set_xzyw	
double_4::set_y	
double_4::set_yw	
double_4::set_ywx	
double_4::set_ywxz	
double_4::set_ywz	
double_4::set_ywzx	
double_4::set_yx	

NAME	DESCRIPTION
double_4::set_yxw	
double_4::set_yxwz	
double_4::set_yxz	
double_4::set_yxzw	
double_4::set_yz	
double_4::set_yzw	
double_4::set_yzwx	
double_4::set_yzx	
double_4::set_yzwx	
double_4::set_z	
double_4::set_zw	
double_4::set_zwx	
double_4::set_zwxy	
double_4::set_zwy	
double_4::set_zwyx	
double_4::set_zx	
double_4::set_zxw	
double_4::set_zxwy	
double_4::set_zxy	
double_4::set_zxyw	
double_4::set_zy	
double_4::set_zyw	
double_4::set_zywx	
double_4::set_zyx	
double_4::set_zyxw	

**Public Operators**

NAME	DESCRIPTION
double_4::operator-	
double_4::operator--	
double_4::operator* =	
double_4::operator/=	
double_4::operator++	
double_4::operator+ =	
double_4::operator=	
double_4::operator-=	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
double_4::a	
double_4::ab	
double_4::abg	
double_4::abgr	
double_4::abr	
double_4::abrg	
double_4::ag	
double_4::agb	
double_4::agbr	
double_4::agr	
double_4::agrb	
double_4::ar	
double_4::arb	

NAME	DESCRIPTION
double_4::arbg	
double_4::arg	
double_4::argb	
double_4::b	
double_4::ba	
double_4::bag	
double_4::bagr	
double_4::bar	
double_4::barg	
double_4::bg	
double_4::bga	
double_4::bgar	
double_4::bgr	
double_4::bgra	
double_4::br	
double_4::bra	
double_4::brag	
double_4::brg	
double_4::brga	
double_4::g	
double_4::ga	
double_4::gab	
double_4::gabr	
double_4::gar	
double_4::garb	

NAME	DESCRIPTION
double_4::gb	
double_4::gba	
double_4::gbar	
double_4::gbr	
double_4::gbra	
double_4::gr	
double_4::gra	
double_4::grab	
double_4::grb	
double_4::grba	
double_4::r	
double_4::ra	
double_4::rab	
double_4::rabg	
double_4::rag	
double_4::ragb	
double_4::rb	
double_4::rba	
double_4::rbag	
double_4::rbg	
double_4::rbga	
double_4::rg	
double_4::rga	
double_4::rgab	
double_4::rgb	

NAME	DESCRIPTION
double_4::rgba	
double_4::w	
double_4::wx	
double_4::wxy	
double_4::wxyz	
double_4::wxz	
double_4::wxzy	
double_4::wy	
double_4::wyx	
double_4::wyz	
double_4::wyzx	
double_4::wz	
double_4::wzx	
double_4::wzxy	
double_4::wzy	
double_4::wzyx	
double_4::x	
double_4::xw	
double_4::xwy	
double_4::xwyz	
double_4::xwz	
double_4::xwzy	
double_4::xy	
double_4::xyw	

NAME	DESCRIPTION
double_4::xyzw	
double_4::xyz	
double_4::xyzw	
double_4::xz	
double_4::xzw	
double_4::xzwy	
double_4::xzy	
double_4::xzyw	
double_4::y	
double_4::yw	
double_4::ywx	
double_4::ywxz	
double_4::yzwz	
double_4::yzwx	
double_4::yx	
double_4::yxw	
double_4::yxwz	
double_4::yxz	
double_4::yxzw	
double_4::yz	
double_4::yzw	
double_4::yzwx	
double_4::yzx	
double_4::yzxw	
double_4::z	

NAME	DESCRIPTION
double_4::zw	
double_4::zwx	
double_4::zwxxy	
double_4::zwy	
double_4::zwyx	
double_4::zx	
double_4::zxw	
double_4::zxwy	
double_4::zxy	
double_4::zxyw	
double_4::zy	
double_4::zyw	
double_4::zywx	
double_4::zyx	
double_4::zyxw	

## Inheritance Hierarchy

`double_4`

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## double\_4

Default constructor, initializes all elements with 0.

```

double_4() restrict(amp,
    cpu);

double_4(
    double _V0,
    double _V1,
    double _V2,
    double _V3) restrict(amp,
    cpu);

double_4(
    double _V) restrict(amp,
    cpu);

double_4(
    const double_4& _Other) restrict(amp,
    cpu);

explicit inline double_4(
    const uint_4& _Other) restrict(amp,
    cpu);

explicit inline double_4(
    const int_4& _Other) restrict(amp,
    cpu);

explicit inline double_4(
    const float_4& _Other) restrict(amp,
    cpu);

explicit inline double_4(
    const unorm_4& _Other) restrict(amp,
    cpu);

explicit inline double_4(
    const norm_4& _Other) restrict(amp,
    cpu);

```

## Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V2*

The value to initialize element 2.

*\_V3*

The value to initialize element 3.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 4;
```

## See also

[Concurrency::graphics Namespace](#)

# float\_2 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of two floats.

## Syntax

```
class float_2;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">float_2 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>float_2::get_x</code>	
<code>float_2::get_xy</code>	
<code>float_2::get_y</code>	
<code>float_2::get_yx</code>	
<code>float_2::ref_g</code>	
<code>float_2::ref_r</code>	
<code>float_2::ref_x</code>	
<code>float_2::ref_y</code>	
<code>float_2::set_x</code>	
<code>float_2::set_xy</code>	
<code>float_2::set_y</code>	

NAME	DESCRIPTION
float_2::set_yx	

### Public Operators

NAME	DESCRIPTION
float_2::operator-	
float_2::operator--	
float_2::operator*= float_2::operator*=	
float_2::operator/=	
float_2::operator++	
float_2::operator+=	
float_2::operator=	
float_2::operator-=	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
float_2::g	
float_2::gr	
float_2::r	
float_2::rg	
float_2::x	
float_2::xy	
float_2::y	
float_2::yx	

## Inheritance Hierarchy

float\_2

# Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## float\_2

Default constructor, initializes all elements with 0.

```
float_2() restrict(amp,
    cpu);

float_2(
    float _V0,
    float _V1) restrict(amp,
    cpu);

float_2(
    float _V) restrict(amp,
    cpu);

float_2(
    const float_2& _Other) restrict(amp,
    cpu);

explicit inline float_2(
    const uint_2& _Other) restrict(amp,
    cpu);

explicit inline float_2(
    const int_2& _Other) restrict(amp,
    cpu);

explicit inline float_2(
    const unorm_2& _Other) restrict(amp,
    cpu);

explicit inline float_2(
    const norm_2& _Other) restrict(amp,
    cpu);

explicit inline float_2(
    const double_2& _Other) restrict(amp,
    cpu);
```

### Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 2;
```

## See also

[Concurrency::graphics Namespace](#)

# float\_3 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of three floats.

## Syntax

```
class float_3;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">float_3 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>float_3::get_x</code>	
<code>float_3::get_xy</code>	
<code>float_3::get_xyz</code>	
<code>float_3::get_xz</code>	
<code>float_3::get_xzy</code>	
<code>float_3::get_y</code>	
<code>float_3::get_yx</code>	
<code>float_3::get_yxz</code>	
<code>float_3::get_yz</code>	
<code>float_3::get_yzx</code>	
<code>float_3::get_z</code>	

NAME	DESCRIPTION
float_3::get_zx	
float_3::get_zxy	
float_3::get_zy	
float_3::get_zyx	
float_3::ref_b	
float_3::ref_g	
float_3::ref_r	
float_3::ref_x	
float_3::ref_y	
float_3::ref_z	
float_3::set_x	
float_3::set_xy	
float_3::set_xyz	
float_3::set_xz	
float_3::set_xzy	
float_3::set_y	
float_3::set_yx	
float_3::set_yxz	
float_3::set_yz	
float_3::set_yzx	
float_3::set_z	
float_3::set_zx	
float_3::set_zxy	
float_3::set_zy	
float_3::set_zyx	

## Public Operators

NAME	DESCRIPTION
float_3::operator-	
float_3::operator--	
float_3::operator*= float_3::operator/= float_3::operator+ + float_3::operator+ = float_3::operator= float_3::operator-=	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
float_3::b	
float_3::bg	
float_3::bgr	
float_3::br	
float_3::brg	
float_3::g	
float_3::gb	
float_3::gbr	
float_3::gr	
float_3::grb	
float_3::r	
float_3::rb	
float_3::rbg	

NAME	DESCRIPTION
float_3::rg	
float_3::rgb	
float_3::x	
float_3::xy	
float_3::xyz	
float_3::xz	
float_3::xzy	
float_3::y	
float_3::yx	
float_3::yz	
float_3::yz	
float_3::yzx	
float_3::z	
float_3::zx	
float_3::zxy	
float_3::zy	
float_3::zyx	

## Inheritance Hierarchy

float\_3

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## float\_3

Default constructor, initializes all elements with 0.

```

float_3() restrict(amp,
    cpu);

float_3(
    float _V0,
    float _V1,
    float _V2) restrict(amp,
    cpu);

float_3(
    float _V) restrict(amp,
    cpu);

float_3(
    const float_3& _Other) restrict(amp,
    cpu);

explicit inline float_3(
    const uint_3& _Other) restrict(amp,
    cpu);

explicit inline float_3(
    const int_3& _Other) restrict(amp,
    cpu);

explicit inline float_3(
    const unorm_3& _Other) restrict(amp,
    cpu);

explicit inline float_3(
    const norm_3& _Other) restrict(amp,
    cpu);

explicit inline float_3(
    const double_3& _Other) restrict(amp,
    cpu);

```

## Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V2*

The value to initialize element 2.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 3;
```

## See also

[Concurrency::graphics Namespace](#)

# float\_4 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of four floats.

## Syntax

```
class float_4;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">float_4 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>float_4::get_w</code>	
<code>float_4::get_wx</code>	
<code>float_4::get_wxy</code>	
<code>float_4::get_wxyz</code>	
<code>float_4::get_wxz</code>	
<code>float_4::get_wxzy</code>	
<code>float_4::get_wy</code>	
<code>float_4::get_wyx</code>	
<code>float_4::get_wyxz</code>	
<code>float_4::get_wyz</code>	
<code>float_4::get_wyzx</code>	

NAME	DESCRIPTION
float_4::get_wz	
float_4::get_wzx	
float_4::get_wzxy	
float_4::get_wzy	
float_4::get_wzyx	
float_4::get_x	
float_4::get_xw	
float_4::get_xwy	
float_4::get_xwyz	
float_4::get_xwz	
float_4::get_xwzy	
float_4::get_xy	
float_4::get_xyw	
float_4::get_xywz	
float_4::get_xyz	
float_4::get_xyzw	
float_4::get_xz	
float_4::get_xzw	
float_4::get_xzwy	
float_4::get_xzy	
float_4::get_xzyw	
float_4::get_y	
float_4::get_yw	
float_4::get_ywx	
float_4::get_ywzx	

NAME	DESCRIPTION
float_4::get_ywz	
float_4::get_ywzx	
float_4::get_yx	
float_4::get_yxw	
float_4::get_yxwz	
float_4::get_yxz	
float_4::get_yxzw	
float_4::get_yz	
float_4::get_yzw	
float_4::get_yzwx	
float_4::get_yzx	
float_4::get_yzxw	
float_4::get_z	
float_4::get_zw	
float_4::get_zwx	
float_4::get_zwxy	
float_4::get_zwy	
float_4::get_zwyx	
float_4::get_zx	
float_4::get_zxw	
float_4::get_zxwy	
float_4::get_zxy	
float_4::get_zxyw	
float_4::get_zy	
float_4::get_zyw	

NAME	DESCRIPTION
float_4::get_zywx	
float_4::get_zyx	
float_4::get_zyxw	
float_4::ref_a	
float_4::ref_b	
float_4::ref_g	
float_4::ref_r	
float_4::ref_w	
float_4::ref_x	
float_4::ref_y	
float_4::ref_z	
float_4::set_w	
float_4::set_wx	
float_4::set_wxy	
float_4::set_wxyz	
float_4::set_wxz	
float_4::set_wxzy	
float_4::set_wy	
float_4::set_wyx	
float_4::set_wyxz	
float_4::set_wyz	
float_4::set_wyzx	
float_4::set_wz	
float_4::set_wzx	
float_4::set_wzxy	

NAME	DESCRIPTION
float_4::set_wzy	
float_4::set_wzyx	
float_4::set_x	
float_4::set_xw	
float_4::set_xwy	
float_4::set_xwyz	
float_4::set_xwz	
float_4::set_xwzy	
float_4::set_xy	
float_4::set_xyw	
float_4::set_xywz	
float_4::set_xyz	
float_4::set_xyzw	
float_4::set_xz	
float_4::set_xzw	
float_4::set_xzwy	
float_4::set_xzy	
float_4::set_xzyw	
float_4::set_y	
float_4::set_yw	
float_4::set_ywx	
float_4::set_ywzx	
float_4::set_ywz	
float_4::set_ywzx	
float_4::set_yx	

NAME	DESCRIPTION
float_4::set_yxw	
float_4::set_yxwz	
float_4::set_yxz	
float_4::set_yxzw	
float_4::set_yz	
float_4::set_yzw	
float_4::set_yzwx	
float_4::set_yzx	
float_4::set_yzxw	
float_4::set_z	
float_4::set_zw	
float_4::set_zwx	
float_4::set_zwxy	
float_4::set_zwy	
float_4::set_zwyx	
float_4::set_zx	
float_4::set_zxw	
float_4::set_zxwy	
float_4::set_zxy	
float_4::set_zxyw	
float_4::set_zy	
float_4::set_zyw	
float_4::set_zywx	
float_4::set_zyx	
float_4::set_zyxw	

## Public Operators

NAME	DESCRIPTION
float_4::operator-	
float_4::operator--	
float_4::operator*= float_4::operator/=	
float_4::operator++	
float_4::operator+=	
float_4::operator=	
float_4::operator-=	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
float_4::a	
float_4::ab	
float_4::abg	
float_4::abgr	
float_4::abr	
float_4::abrg	
float_4::ag	
float_4::agb	
float_4::agbr	
float_4::agr	
float_4::agrb	
float_4::ar	
float_4::arb	

NAME	DESCRIPTION
float_4::arbg	
float_4::arg	
float_4::argb	
float_4::b	
float_4::ba	
float_4::bag	
float_4::bagr	
float_4::bar	
float_4::barg	
float_4::bg	
float_4::bga	
float_4::bgar	
float_4::bgr	
float_4::bgra	
float_4::br	
float_4::bra	
float_4::brag	
float_4::brg	
float_4::brga	
float_4::g	
float_4::ga	
float_4::gab	
float_4::gabr	
float_4::gar	
float_4::garb	

NAME	DESCRIPTION
float_4::gb	
float_4::gba	
float_4::gbar	
float_4::gbr	
float_4::gbra	
float_4::gr	
float_4::gra	
float_4::grab	
float_4::grb	
float_4::grba	
float_4::r	
float_4::ra	
float_4::rab	
float_4::rabg	
float_4::rag	
float_4::ragb	
float_4::rb	
float_4::rba	
float_4::rbag	
float_4::rbg	
float_4::rbga	
float_4::rg	
float_4::rga	
float_4::rgab	
float_4::rgb	

NAME	DESCRIPTION
float_4::rgba	
float_4::w	
float_4::wx	
float_4::wxy	
float_4::wxyz	
float_4::wxz	
float_4::wxzy	
float_4::wy	
float_4::wyx	
float_4::wyz	
float_4::wyzx	
float_4::wz	
float_4::wzx	
float_4::wzxy	
float_4::wzy	
float_4::wzyx	
float_4::x	
float_4::xw	
float_4::xwy	
float_4::xwyz	
float_4::xwz	
float_4::xwzy	
float_4::xy	
float_4::xyw	

NAME	DESCRIPTION
float_4::xywz	
float_4::xyz	
float_4::xyzw	
float_4::xz	
float_4::xzw	
float_4::xzwy	
float_4::xzy	
float_4::xzyw	
float_4::y	
float_4::yw	
float_4::ywx	
float_4::ywxz	
float_4::yz	
float_4::yzx	
float_4::yx	
float_4::yxw	
float_4::yxwz	
float_4::yxz	
float_4::yxzw	
float_4::yz	
float_4::yzw	
float_4::yzwx	
float_4::yzx	
float_4::yzxw	
float_4::z	

NAME	DESCRIPTION
float_4::zw	
float_4::zwx	
float_4::zwxxy	
float_4::zwy	
float_4::zwyx	
float_4::zx	
float_4::zxw	
float_4::zxy	
float_4::zxyw	
float_4::zy	
float_4::zyw	
float_4::zywx	
float_4::zyx	
float_4::zyxw	

## Inheritance Hierarchy

float\_4

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## float\_4

Default constructor, initializes all elements with 0.

```

float_4() restrict(amp,
    cpu);

float_4(
    float _V0,
    float _V1,
    float _V2,
    float _V3) restrict(amp,
    cpu);

float_4(
    float _V) restrict(amp,
    cpu);

float_4(
    const float_4& _Other) restrict(amp,
    cpu);

explicit inline float_4(
    const uint_4& _Other) restrict(amp,
    cpu);

explicit inline float_4(
    const int_4& _Other) restrict(amp,
    cpu);

explicit inline float_4(
    const unorm_4& _Other) restrict(amp,
    cpu);

explicit inline float_4(
    const norm_4& _Other) restrict(amp,
    cpu);

explicit inline float_4(
    const double_4& _Other) restrict(amp,
    cpu);

```

## Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V2*

The value to initialize element 2.

*\_V3*

The value to initialize element 3.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 4;
```

## See also

[Concurrency::graphics Namespace](#)

# int\_2 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of two integers.

## Syntax

```
class int_2;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">int_2 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>int_2::get_x</code>	
<code>int_2::get_xy</code>	
<code>int_2::get_y</code>	
<code>int_2::get_yx</code>	
<code>int_2::ref_g</code>	
<code>int_2::ref_r</code>	
<code>int_2::ref_x</code>	
<code>int_2::ref_y</code>	
<code>int_2::set_x</code>	
<code>int_2::set_xy</code>	
<code>int_2::set_y</code>	

NAME	DESCRIPTION
int_2::set_yx	

### Public Operators

NAME	DESCRIPTION
int_2::operator-	
int_2::operator--	
int_2::operator%=	
int_2::operator&=	
int_2::operator*=	
int_2::operator/=	
int_2::operator^=	
int_2::operator =	
int_2::operator~	
int_2::operator++	
int_2::operator+=	
int_2::operator<<=	
int_2::operator=	
int_2::operator-=	
int_2::operator>>=	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
int_2::g	
int_2::gr	
int_2::r	

NAME	DESCRIPTION
int_2::rg	
int_2::x	
int_2::xy	
int_2::y	
int_2::yx	

## Inheritance Hierarchy

`int_2`

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## int\_2

Default constructor, initializes all elements with 0.

```
int_2() restrict(amp,
    cpu);

int_2(
    int _V0,
    int _V1) restrict(amp,
    cpu);

int_2(
    int _V) restrict(amp,
    cpu);

int_2(
    const int_2& _Other) restrict(amp,
    cpu);

explicit inline int_2(
    const uint_2& _Other) restrict(amp,
    cpu);

explicit inline int_2(
    const float_2& _Other) restrict(amp,
    cpu);

explicit inline int_2(
    const unorm_2& _Other) restrict(amp,
    cpu);

explicit inline int_2(
    const norm_2& _Other) restrict(amp,
    cpu);

explicit inline int_2(
    const double_2& _Other) restrict(amp,
    cpu);
```

### Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 2;
```

## See also

[Concurrency::graphics Namespace](#)

# int\_3 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of three integers.

## Syntax

```
class int_3;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">int_3 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>int_3::get_x</code>	
<code>int_3::get_xy</code>	
<code>int_3::get_xyz</code>	
<code>int_3::get_xz</code>	
<code>int_3::get_xzy</code>	
<code>int_3::get_y</code>	
<code>int_3::get_yx</code>	
<code>int_3::get_yxz</code>	
<code>int_3::get_yz</code>	
<code>int_3::get_yzx</code>	
<code>int_3::get_z</code>	

NAME	DESCRIPTION
int_3::get_zx	
int_3::get_zxy	
int_3::get_zy	
int_3::get_zyx	
int_3::ref_b	
int_3::ref_g	
int_3::ref_r	
int_3::ref_x	
int_3::ref_y	
int_3::ref_z	
int_3::set_x	
int_3::set_xy	
int_3::set_xyz	
int_3::set_xz	
int_3::set_xzy	
int_3::set_y	
int_3::set_yx	
int_3::set_yxz	
int_3::set_yz	
int_3::set_yzx	
int_3::set_z	
int_3::set_zx	
int_3::set_zxy	
int_3::set_zy	
int_3::set_zyx	

## Public Operators

NAME	DESCRIPTION
int_3::operator-	
int_3::operator--	
int_3::operator% =	
int_3::operator& =	
int_3::operator* =	
int_3::operator/=	
int_3::operator^ =	
int_3::operator =	
int_3::operator~	
int_3::operator++	
int_3::operator+=	
int_3::operator<< =	
int_3::operator=	
int_3::operator-=	
int_3::operator>> =	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
int_3::b	
int_3::bg	
int_3::bgr	
int_3::br	
int_3::brg	
int_3::g	

NAME	DESCRIPTION
int_3::gb	
int_3::gbr	
int_3::gr	
int_3::grb	
int_3::r	
int_3::rb	
int_3::rbg	
int_3::rg	
int_3::rgb	
int_3::x	
int_3::xy	
int_3::xyz	
int_3::xz	
int_3::xzy	
int_3::y	
int_3::yx	
int_3::yxz	
int_3::yz	
int_3::yzx	
int_3::z	
int_3::zx	
int_3::zxy	
int_3::zy	
int_3::zyx	

## Inheritance Hierarchy

int\_3

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## int\_3

Default constructor, initializes all elements with 0.

## Syntax

```
int_3() restrict(amp,cpu);
int_3(
    int _V0,
    int _V1,
    int _V2
) restrict(amp,cpu);
int_3(
    int _V
) restrict(amp,cpu);
int_3(
    const int_3& _Other
) restrict(amp,cpu);
explicit inline int_3(
    const uint_3& _Other
) restrict(amp,cpu);
explicit inline int_3(
    const float_3& _Other
) restrict(amp,cpu);
explicit inline int_3(
    const unorm_3& _Other
) restrict(amp,cpu);
explicit inline int_3(
    const norm_3& _Other
) restrict(amp,cpu);
explicit inline int_3(
    const double_3& _Other
) restrict(amp,cpu);
```

### Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V2*

The value to initialize element 2.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

# Syntax

```
static const int size = 3;
```

## See also

[Concurrency::graphics Namespace](#)

# int\_4 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of four integers.

## Syntax

```
class int_4;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">int_4 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>int_4::get_w</code>	
<code>int_4::get_wx</code>	
<code>int_4::get_wxy</code>	
<code>int_4::get_wxyz</code>	
<code>int_4::get_wxz</code>	
<code>int_4::get_wxzy</code>	
<code>int_4::get_wy</code>	
<code>int_4::get_wyx</code>	
<code>int_4::get_wyxz</code>	
<code>int_4::get_wyz</code>	
<code>int_4::get_wyzx</code>	

NAME	DESCRIPTION
int_4::get_wz	
int_4::get_wzx	
int_4::get_wzxy	
int_4::get_wzy	
int_4::get_wzyx	
int_4::get_x	
int_4::get_xw	
int_4::get_xwy	
int_4::get_xwyz	
int_4::get_xwz	
int_4::get_xwzy	
int_4::get_xy	
int_4::get_xyw	
int_4::get_xywz	
int_4::get_xyz	
int_4::get_xyzw	
int_4::get_xz	
int_4::get_xzw	
int_4::get_xzwy	
int_4::get_xzy	
int_4::get_xzyw	
int_4::get_y	
int_4::get_yw	
int_4::get_ywx	
int_4::get_ywz	

NAME	DESCRIPTION
int_4::get_ywz	
int_4::get_ywzx	
int_4::get_yx	
int_4::get_yxw	
int_4::get_yxwz	
int_4::get_yxz	
int_4::get_yxzw	
int_4::get_yz	
int_4::get_yzw	
int_4::get_yzwx	
int_4::get_yzx	
int_4::get_yzxw	
int_4::get_z	
int_4::get_zw	
int_4::get_zwx	
int_4::get_zwxy	
int_4::get_zwy	
int_4::get_zwyx	
int_4::get_zx	
int_4::get_zxw	
int_4::get_zxwy	
int_4::get_zxy	
int_4::get_zxyw	
int_4::get_zy	
int_4::get_zyw	

NAME	DESCRIPTION
int_4::get_zywx	
int_4::get_zyx	
int_4::get_zyxw	
int_4::ref_a	
int_4::ref_b	
int_4::ref_g	
int_4::ref_r	
int_4::ref_w	
int_4::ref_x	
int_4::ref_y	
int_4::ref_z	
int_4::set_w	
int_4::set_wx	
int_4::set_wxy	
int_4::set_wxyz	
int_4::set_wxz	
int_4::set_wxzy	
int_4::set_wy	
int_4::set_wyx	
int_4::set_wyxz	
int_4::set_wyz	
int_4::set_wyzx	
int_4::set_wz	
int_4::set_wzx	
int_4::set_wzxy	

NAME	DESCRIPTION
int_4::set_wzy	
int_4::set_wzyx	
int_4::set_x	
int_4::set_xw	
int_4::set_xwy	
int_4::set_xwyz	
int_4::set_xwz	
int_4::set_xwzy	
int_4::set_xy	
int_4::set_xyw	
int_4::set_xywz	
int_4::set_xyz	
int_4::set_xyzw	
int_4::set_xz	
int_4::set_xzw	
int_4::set_xzwy	
int_4::set_xzy	
int_4::set_xzyw	
int_4::set_y	
int_4::set_yw	
int_4::set_ywx	
int_4::set_ywxyz	
int_4::set_ywz	
int_4::set_ywzx	
int_4::set_yx	

NAME	DESCRIPTION
int_4::set_yxw	
int_4::set_yxwz	
int_4::set_yxz	
int_4::set_yxzw	
int_4::set_yz	
int_4::set_yzw	
int_4::set_yzwx	
int_4::set_yzx	
int_4::set_yxzw	
int_4::set_z	
int_4::set_zw	
int_4::set_zwx	
int_4::set_zwxy	
int_4::set_zwy	
int_4::set_zwyx	
int_4::set_zx	
int_4::set_zxw	
int_4::set_zxwy	
int_4::set_zxy	
int_4::set_zxyw	
int_4::set_zy	
int_4::set_zyw	
int_4::set_zywx	
int_4::set_zyx	
int_4::set_zyxw	

**Public Operators**

NAME	DESCRIPTION
int_4::operator-	
int_4::operator--	
int_4::operator% =	
int_4::operator& =	
int_4::operator* =	
int_4::operator/=	
int_4::operator^ =	
int_4::operator =	
int_4::operator~	
int_4::operator++	
int_4::operator+=	
int_4::operator<< =	
int_4::operator=	
int_4::operator-=	
int_4::operator>> =	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
int_4::a	
int_4::ab	
int_4::abg	
int_4::abgr	
int_4::abr	
int_4::abrg	

NAME	DESCRIPTION
int_4::ag	
int_4::agb	
int_4::agbr	
int_4::agr	
int_4::agrb	
int_4::ar	
int_4::arb	
int_4::arbg	
int_4::arg	
int_4::argb	
int_4::b	
int_4::ba	
int_4::bag	
int_4::bagr	
int_4::bar	
int_4::barg	
int_4::bg	
int_4::bga	
int_4::bgar	
int_4::bgr	
int_4::bgra	
int_4::br	
int_4::bra	
int_4::brag	
int_4::brg	

NAME	DESCRIPTION
int_4::brga	
int_4::g	
int_4::ga	
int_4::gab	
int_4::gabr	
int_4::gar	
int_4::garb	
int_4::gb	
int_4::gba	
int_4::gbar	
int_4::gbr	
int_4::gbra	
int_4::gr	
int_4::gra	
int_4::grab	
int_4::grb	
int_4::grba	
int_4::r	
int_4::ra	
int_4::rab	
int_4::rabg	
int_4::rag	
int_4::ragb	
int_4::rb	
int_4::rba	

NAME	DESCRIPTION
int_4::rbag	
int_4::rbg	
int_4::rbga	
int_4::rg	
int_4::rga	
int_4::rgab	
int_4::rgb	
int_4::rgba	
int_4::w	
int_4::wx	
int_4::wxy	
int_4::wxyz	
int_4::wxz	
int_4::wxzy	
int_4::wy	
int_4::wyx	
int_4::wyxz	
int_4::wyz	
int_4::wyzx	
int_4::wz	
int_4::wzx	
int_4::wzxy	
int_4::wzy	
int_4::wzyx	
int_4::x	

NAME	DESCRIPTION
int_4::xw	
int_4::xwy	
int_4::xwyz	
int_4::xwz	
int_4::xwzy	
int_4::xy	
int_4::xyw	
int_4::xywz	
int_4::xyz	
int_4::xyzw	
int_4::xz	
int_4::xzw	
int_4::xzwy	
int_4::xzy	
int_4::xzyw	
int_4::y	
int_4::yw	
int_4::ywx	
int_4::ywxz	
int_4::ywz	
int_4::ywzx	
int_4::yx	
int_4::yxw	
int_4::yxwz	
int_4::yxz	

NAME	DESCRIPTION
int_4::yxzw	
int_4::yz	
int_4::yzw	
int_4::yzwx	
int_4::yzx	
int_4::yzxw	
int_4::z	
int_4::zw	
int_4::zwx	
int_4::zwxw	
int_4::zwy	
int_4::zwyx	
int_4::zx	
int_4::zxw	
int_4::zxwy	
int_4::zxy	
int_4::zxyw	
int_4::zy	
int_4::zyw	
int_4::zywx	
int_4::zyx	
int_4::zyxw	

## Inheritance Hierarchy

`int_4`

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## int\_4

Default constructor, initializes all elements with 0.

```
int_4() restrict(amp,
    cpu);

int_4(
    int _V0,
    int _V1,
    int _V2,
    int _V3) restrict(amp,
    cpu);

int_4(
    int _V) restrict(amp,
    cpu);

int_4(
    const int_4& _Other) restrict(amp,
    cpu);

explicit inline int_4(
    const uint_4& _Other) restrict(amp,
    cpu);

explicit inline int_4(
    const float_4& _Other) restrict(amp,
    cpu);

explicit inline int_4(
    const unorm_4& _Other) restrict(amp,
    cpu);

explicit inline int_4(
    const norm_4& _Other) restrict(amp,
    cpu);

explicit inline int_4(
    const double_4& _Other) restrict(amp,
    cpu);
```

### Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V2*

The value to initialize element 2.

*\_V3*

The value to initialize element 3.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 4;
```

## See also

[Concurrency::graphics Namespace](#)

# norm Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represent a norm number. Each element is a floating point number in the range of [-1.0f, 1.0f].

## Syntax

```
class norm;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">norm Constructor</a>	Overloaded. Default constructor. Initialize to 0.0f.

### Public Operators

NAME	DESCRIPTION
norm::operator-	
norm::operator--	
norm::operator float	Conversion operator. Convert the norm number to a floating point value.
norm::operator* =	
norm::operator/=	
norm::operator++	
norm::operator+=	
norm::operator=	
norm::operator-=	

## Inheritance Hierarchy

norm

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

# norm

Default constructor. Initialize to 0.0f.

```
norm(  
    void) restrict(amp,  
    cpu);  
  
explicit norm(  
    float _V) restrict(amp,  
    cpu);  
  
explicit norm(  
    unsigned int _V) restrict(amp,  
    cpu);  
  
explicit norm(  
    int _V) restrict(amp,  
    cpu);  
  
explicit norm(  
    double _V) restrict(amp,  
    cpu);  
  
norm(  
    const norm& _Other) restrict(amp,  
    cpu);  
  
norm(  
    const unorm& _Other) restrict(amp,  
    cpu);
```

## Parameters

*\_V*

The value used to initialize.

*\_Other*

The object used to initialize.

## See also

[Concurrency::graphics Namespace](#)

# norm\_2 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of two normal numbers.

## Syntax

```
class norm_2;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">norm_2 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>norm_2::get_x</code>	
<code>norm_2::get_xy</code>	
<code>norm_2::get_y</code>	
<code>norm_2::get_yx</code>	
<code>norm_2::ref_g</code>	
<code>norm_2::ref_r</code>	
<code>norm_2::ref_x</code>	
<code>norm_2::ref_y</code>	
<code>norm_2::set_x</code>	
<code>norm_2::set_xy</code>	
<code>norm_2::set_y</code>	

NAME	DESCRIPTION
norm_2::set_yx	

### Public Operators

NAME	DESCRIPTION
norm_2::operator-	
norm_2::operator--	
norm_2::operator* =	
norm_2::operator/=	
norm_2::operator++	
norm_2::operator+=	
norm_2::operator=	
norm_2::operator-=	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
norm_2::g	
norm_2::gr	
norm_2::r	
norm_2::rg	
norm_2::x	
norm_2::xy	
norm_2::y	
norm_2::yx	

## Inheritance Hierarchy

norm\_2

# Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## norm\_2

Default constructor, initializes all elements with 0.

```
norm_2() restrict(amp,
    cpu);

norm_2(
    norm _V0,
    norm _V1) restrict(amp,
    cpu);

norm_2(
    float _V0,
    float _V1) restrict(amp,
    cpu);

norm_2(
    unorm _V0,
    unorm _V1) restrict(amp,
    cpu);

norm_2(
    norm _V) restrict(amp,
    cpu);

explicit norm_2(
    float _V) restrict(amp,
    cpu);

norm_2(
    const norm_2& _Other) restrict(amp,
    cpu);

explicit inline norm_2(
    const uint_2& _Other) restrict(amp,
    cpu);

explicit inline norm_2(
    const int_2& _Other) restrict(amp,
    cpu);

explicit inline norm_2(
    const float_2& _Other) restrict(amp,
    cpu);

explicit inline norm_2(
    const unorm_2& _Other) restrict(amp,
    cpu);

explicit inline norm_2(
    const double_2& _Other) restrict(amp,
    cpu);
```

### Parameters

**\_V0**

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 2;
```

## See also

[Concurrency::graphics Namespace](#)

# norm\_3 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of three normal numbers.

## Syntax

```
class norm_3;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">norm_3 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>norm_3::get_x</code>	
<code>norm_3::get_xy</code>	
<code>norm_3::get_xyz</code>	
<code>norm_3::get_xz</code>	
<code>norm_3::get_xzy</code>	
<code>norm_3::get_y</code>	
<code>norm_3::get_yx</code>	
<code>norm_3::get_yxz</code>	
<code>norm_3::get_yz</code>	
<code>norm_3::get_yzx</code>	
<code>norm_3::get_z</code>	

NAME	DESCRIPTION
norm_3::get_zx	
norm_3::get_zxy	
norm_3::get_zy	
norm_3::get_zyx	
norm_3::ref_b	
norm_3::ref_g	
norm_3::ref_r	
norm_3::ref_x	
norm_3::ref_y	
norm_3::ref_z	
norm_3::set_x	
norm_3::set_xy	
norm_3::set_xyz	
norm_3::set_xz	
norm_3::set_xzy	
norm_3::set_y	
norm_3::set_yx	
norm_3::set_yxz	
norm_3::set_yz	
norm_3::set_yzx	
norm_3::set_z	
norm_3::set_zx	
norm_3::set_zxy	
norm_3::set_zy	
norm_3::set_zyx	

**Public Operators**

NAME	DESCRIPTION
norm_3::operator-	
norm_3::operator--	
norm_3::operator*= norm_3::operator/=	
norm_3::operator++ norm_3::operator+=	
norm_3::operator=	
norm_3::operator-=	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
norm_3::b	
norm_3::bg	
norm_3::bgr	
norm_3::br	
norm_3::brg	
norm_3::g	
norm_3::gb	
norm_3::gbr	
norm_3::gr	
norm_3::grb	
norm_3::r	
norm_3::rb	
norm_3::rbg	

NAME	DESCRIPTION
norm_3::rg	
norm_3::rgb	
norm_3::x	
norm_3::xy	
norm_3::xyz	
norm_3::xz	
norm_3::xzy	
norm_3::y	
norm_3::yx	
norm_3::yxz	
norm_3::yz	
norm_3::yzx	
norm_3::z	
norm_3::zx	
norm_3::zxy	
norm_3::zy	
norm_3::zyx	

## Inheritance Hierarchy

norm\_3

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## norm\_3 Constructor

Default constructor, initializes all elements with 0.

## Syntax

```

norm_3() restrict(amp,cpu);
norm_3(
    norm _V0,
    norm _V1,
    norm _V2
) restrict(amp,cpu);
norm_3(
    float _V0,
    float _V1,
    float _V2
) restrict(amp,cpu);
norm_3(
    unorm _V0,
    unorm _V1,
    unorm _V2
) restrict(amp,cpu);
norm_3(
    norm _V
) restrict(amp,cpu);
explicit norm_3(
    float _V
) restrict(amp,cpu);
norm_3(
    const norm_3& _Other
) restrict(amp,cpu);
explicit inline norm_3(
    const uint_3& _Other
) restrict(amp,cpu);
explicit inline norm_3(
    const int_3& _Other
) restrict(amp,cpu);
explicit inline norm_3(
    const float_3& _Other
) restrict(amp,cpu);
explicit inline norm_3(
    const unorm_3& _Other
) restrict(amp,cpu);
explicit inline norm_3(
    const double_3& _Other
) restrict(amp,cpu);

```

#### Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V2*

The value to initialize element 2.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size Constant

#### Syntax

```
static const int size = 3;
```

## See also

[Concurrency::graphics Namespace](#)

# norm\_4 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of four normal numbers.

## Syntax

```
class norm_4;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">norm_4 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>norm_4::get_w</code>	
<code>norm_4::get_wx</code>	
<code>norm_4::get_wxy</code>	
<code>norm_4::get_wxyz</code>	
<code>norm_4::get_wxz</code>	
<code>norm_4::get_wzy</code>	
<code>norm_4::get_wy</code>	
<code>norm_4::get_wyx</code>	
<code>norm_4::get_wyxz</code>	
<code>norm_4::get_wyz</code>	
<code>norm_4::get_wyzx</code>	

NAME	DESCRIPTION
norm_4::get_wz	
norm_4::get_wzx	
norm_4::get_wzxy	
norm_4::get_wzy	
norm_4::get_wzyx	
norm_4::get_x	
norm_4::get_xw	
norm_4::get_xwy	
norm_4::get_xwyz	
norm_4::get_xwz	
norm_4::get_xwzy	
norm_4::get_xy	
norm_4::get_xyw	
norm_4::get_xywz	
norm_4::get_xyz	
norm_4::get_xyzw	
norm_4::get_xz	
norm_4::get_xzw	
norm_4::get_xzwy	
norm_4::get_xzy	
norm_4::get_xzyw	
norm_4::get_y	
norm_4::get_yw	
norm_4::get_ywx	
norm_4::get_ywyz	

NAME	DESCRIPTION
norm_4::get_ywz	
norm_4::get_ywzx	
norm_4::get_yx	
norm_4::get_yxw	
norm_4::get_yxwz	
norm_4::get_yxz	
norm_4::get_yxzw	
norm_4::get_yz	
norm_4::get_yzw	
norm_4::get_yzwx	
norm_4::get_yzx	
norm_4::get_yzxw	
norm_4::get_z	
norm_4::get_zw	
norm_4::get_zwx	
norm_4::get_zwxy	
norm_4::get_zwy	
norm_4::get_zwyx	
norm_4::get_zx	
norm_4::get_zxw	
norm_4::get_zxwy	
norm_4::get_zxy	
norm_4::get_zxyw	
norm_4::get_zy	
norm_4::get_zyw	

NAME	DESCRIPTION
norm_4::get_zywx	
norm_4::get_zyx	
norm_4::get_zyxw	
norm_4::ref_a	
norm_4::ref_b	
norm_4::ref_g	
norm_4::ref_r	
norm_4::ref_w	
norm_4::ref_x	
norm_4::ref_y	
norm_4::ref_z	
norm_4::set_w	
norm_4::set_wx	
norm_4::set_wxy	
norm_4::set_wxyz	
norm_4::set_wxz	
norm_4::set_wxzy	
norm_4::set_wy	
norm_4::set_wyx	
norm_4::set_wyz	
norm_4::set_wyzx	
norm_4::set_wz	
norm_4::set_wzx	
norm_4::set_wzxy	

NAME	DESCRIPTION
norm_4::set_wzy	
norm_4::set_wzyx	
norm_4::set_x	
norm_4::set_xw	
norm_4::set_xwy	
norm_4::set_xwyz	
norm_4::set_xwz	
norm_4::set_xwzy	
norm_4::set_xy	
norm_4::set_xyw	
norm_4::set_xywz	
norm_4::set_xyz	
norm_4::set_xyzw	
norm_4::set_xz	
norm_4::set_xzw	
norm_4::set_xzwy	
norm_4::set_xzy	
norm_4::set_xzyw	
norm_4::set_y	
norm_4::set_yw	
norm_4::set_ywx	
norm_4::set_ywxz	
norm_4::set_ywz	
norm_4::set_ywzx	
norm_4::set_yx	

NAME	DESCRIPTION
norm_4::set_yxw	
norm_4::set_yxwz	
norm_4::set_yxz	
norm_4::set_yxzw	
norm_4::set_yz	
norm_4::set_yzw	
norm_4::set_yzwx	
norm_4::set_yzx	
norm_4::set_yzwx	
norm_4::set_z	
norm_4::set_zw	
norm_4::set_zwx	
norm_4::set_zwxy	
norm_4::set_zwy	
norm_4::set_zwyx	
norm_4::set_zx	
norm_4::set_zxw	
norm_4::set_zxwy	
norm_4::set_zxy	
norm_4::set_zxyw	
norm_4::set_zy	
norm_4::set_zyw	
norm_4::set_zywx	
norm_4::set_zyx	
norm_4::set_zyxw	

**Public Operators**

NAME	DESCRIPTION
norm_4::operator-	
norm_4::operator--	
norm_4::operator* =	
norm_4::operator/=	
norm_4::operator++	
norm_4::operator+=	
norm_4::operator=	
norm_4::operator-=	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
norm_4::a	
norm_4::ab	
norm_4::abg	
norm_4::abgr	
norm_4::abr	
norm_4::abrg	
norm_4::ag	
norm_4::agb	
norm_4::agbr	
norm_4::agr	
norm_4::agrb	
norm_4::ar	
norm_4::arb	

NAME	DESCRIPTION
norm_4::arbg	
norm_4::arg	
norm_4::argb	
norm_4::b	
norm_4::ba	
norm_4::bag	
norm_4::bagr	
norm_4::bar	
norm_4::barg	
norm_4::bg	
norm_4::bga	
norm_4::bgar	
norm_4::bgr	
norm_4::bgra	
norm_4::br	
norm_4::bra	
norm_4::brag	
norm_4::brg	
norm_4::brga	
norm_4::g	
norm_4::ga	
norm_4::gab	
norm_4::gabr	
norm_4::gar	
norm_4::garb	

NAME	DESCRIPTION
norm_4::gb	
norm_4::gba	
norm_4::gbar	
norm_4::gbr	
norm_4::gbra	
norm_4::gr	
norm_4::gra	
norm_4::grab	
norm_4::grb	
norm_4::grba	
norm_4::r	
norm_4::ra	
norm_4::rab	
norm_4::rabg	
norm_4::rag	
norm_4::ragb	
norm_4::rb	
norm_4::rba	
norm_4::rbag	
norm_4::rbg	
norm_4::rbga	
norm_4::rg	
norm_4::rga	
norm_4::rgab	
norm_4::rgb	

NAME	DESCRIPTION
norm_4::rgba	
norm_4::w	
norm_4::wx	
norm_4::wxy	
norm_4::wxyz	
norm_4::wxz	
norm_4::wxzy	
norm_4::wy	
norm_4::wyx	
norm_4::wyxz	
norm_4::wyz	
norm_4::wyzx	
norm_4::wz	
norm_4::wzx	
norm_4::wzxy	
norm_4::wzy	
norm_4::wzyx	
norm_4::x	
norm_4::xw	
norm_4::xwy	
norm_4::xwyz	
norm_4::xwz	
norm_4::xwzy	
norm_4::xy	
norm_4::xyw	

NAME	DESCRIPTION
norm_4::xywz	
norm_4::xyz	
norm_4::xyzw	
norm_4::xz	
norm_4::xzw	
norm_4::xzwy	
norm_4::xzy	
norm_4::xzyw	
norm_4::y	
norm_4::yw	
norm_4::ywx	
norm_4::ywxz	
norm_4::ywz	
norm_4::ywzx	
norm_4::yx	
norm_4::yxw	
norm_4::yxwz	
norm_4::yxz	
norm_4::yxzw	
norm_4::yz	
norm_4::yzw	
norm_4::yzwx	
norm_4::yzx	
norm_4::yzxw	
norm_4::z	

NAME	DESCRIPTION
norm_4::zw	
norm_4::zwx	
norm_4::zwxxy	
norm_4::zwy	
norm_4::zwyx	
norm_4::zx	
norm_4::zxw	
norm_4::zxwy	
norm_4::zxy	
norm_4::zxyw	
norm_4::zy	
norm_4::zyw	
norm_4::zywx	
norm_4::zyx	
norm_4::zyxw	

## Inheritance Hierarchy

norm\_4

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## norm\_4

Default constructor, initializes all elements with 0.

```

norm_4() restrict(amp,
    cpu);

norm_4(
    norm _V0,
    norm _V1,
    norm _V2,
    norm _V3) restrict(amp,
    cpu);

norm_4(
    float _V0,
    float _V1,
    float _V2,
    float _V3) restrict(amp,
    cpu);

norm_4(
    unorm _V0,
    unorm _V1,
    unorm _V2,
    unorm _V3) restrict(amp,
    cpu);

norm_4(
    norm _V) restrict(amp,
    cpu);

explicit norm_4(
    float _V) restrict(amp,
    cpu);

norm_4(
    const norm_4& _Other) restrict(amp,
    cpu);

explicit inline norm_4(
    const uint_4& _Other) restrict(amp,
    cpu);

explicit inline norm_4(
    const int_4& _Other) restrict(amp,
    cpu);

explicit inline norm_4(
    const float_4& _Other) restrict(amp,
    cpu);

explicit inline norm_4(
    const unorm_4& _Other) restrict(amp,
    cpu);

explicit inline norm_4(
    const double_4& _Other) restrict(amp,
    cpu);

```

## Parameters

**\_V0**

The value to initialize element 0.

**\_V1**

The value to initialize element 1.

**\_V2**

The value to initialize element 2.

*\_V3*

The value to initialize element 3.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 4;
```

## See also

[Concurrency::graphics Namespace](#)

# sampler Class

10/31/2018 • 2 minutes to read • [Edit Online](#)

The sampler class aggregates sampling configuration information to be used for texture sampling.

## Syntax

```
class sampler;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">sampler Constructor</a>	Overloaded. Constructs a sampler instance.

### Public Methods

NAME	DESCRIPTION
<a href="#">get_address_mode</a>	Returns the <code>address_mode</code> that's associated with the sampler object.
<a href="#">get_border_color</a>	Returns the border color that's associated with the sampler object.
<a href="#">get_filter_mode</a>	Returns the <code>filter_mode</code> that's associated with the sampler object.

### Public Operators

NAME	DESCRIPTION
<a href="#">operator=</a>	Overloaded. Assignment operator.

### Public Data Members

NAME	DESCRIPTION
<a href="#">address_mode</a>	Gets the address mode of the <code>sampler</code> object.
<a href="#">border_color</a>	Gets the border color of the <code>sampler</code> object.
<a href="#">filter_mode</a>	Gets the filter mode of the <code>sampler</code> object.

## Inheritance Hierarchy

`sampler`

# Requirements

**Header:** amp\_graphics.h

**Namespace:** concurrency::graphics

## sampler

Constructs an instance of the [sampler Class](#).

```
sampler() restrict(cpu); // [1] default constructor

sampler( // [2] constructor
    filter_mode _Filter_mode) restrict(cpu);

sampler( // [3] constructor
    address_mode _Address_mode,
    float_4 _Border_color = float_4(0.0f,
    0.0f,
    0.0f,
    0.0f)) restrict(cpu);

sampler( // [4] constructor
    filter_mode _Filter_mode,
    address_mode _Address_mode,
    float_4 _Border_color = float_4(0.0f,
    0.0f,
    0.0f,
    0.0f)) restrict(cpu);

sampler( // [5] copy constructor
    const sampler& _Other) restrict(amp,
    cpu);

sampler( // [6] move constructor
    sampler&& _Other) restrict(amp,
    cpu);
```

### Parameters

*\_Filter\_mode*

The filter mode to be used in sampling.

*\_Address\_mode*

The addressing mode to be used in sampling for all dimensions.

*\_Border\_color*

The border color to be used if the address mode is address\_border. The default value is

```
float_4(0.0f, 0.0f, 0.0f, 0.0f) .
```

*\_Other*

[5] Copy Constructor The `sampler` object to copy into the new `sampler` instance.

[6] Move Constructor The `sampler` object to move into the new `sampler` instance.

## address\_mode

Gets the address mode of the `sampler` object.

```
__declspec(property(get= get_address_mode)) Concurrency::graphics::address_mode address_mode;
```

## border\_color

Gets the border color of the `sampler` object.

```
__declspec(property(get= get_border_color)) Concurrency::graphics::float_4 border_color;
```

## filter\_mode

Gets the filter mode of the `sampler` object.

```
__declspec(property(get= get_filter_mode)) Concurrency::graphics::filter_mode filter_mode;
```

## get\_address\_mode

Returns the filter mode that's configured for this `sampler`.

```
Concurrency::graphics::address_mode get_address_mode() const __GPU;
```

### Return Value

The address mode that's configured for the sampler.

## get\_border\_color

Returns the border color that's configured for this `sampler`.

```
Concurrency::graphics::float_4 get_border_color() const restrict(amp, cpu);
```

### Return Value

A `float_4` that contains the border color.

## get\_filter\_mode

Returns the filter mode that's configured for this `sampler`.

```
Concurrency::graphics::filter_mode get_filter_mode() const restrict(amp, cpu);
```

### Return Value

The filter mode that's configured for the sampler.

## operator=

Assigns the value of another sampler object to an existing sampler.

```
sampler& operator= ( // [1] copy assignment operator
    const sampler& _Other) restrict(amp, cpu);

sampler& operator= ( // [2] move assignment operator
    sampler&& _Other) restrict(amp, cpu);
```

## Parameters

*\_Other*

[1] Copy Assignment Operator The `sampler` object to copy into this `sampler` .

[2] Move Assignment Operator The `sampler` object to move into this `sampler` .

## Return Value

A reference to this sampler instance.

## See also

[Concurrency::graphics Namespace](#)

# short\_vector Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

short\_vector provides metaprogramming definitions which are useful for programming short vectors generically.

## Syntax

```
template<
    typename _Scalar_type,
    int _Size
>
struct short_vector;
template<>
struct short_vector<unsigned int, 1>;
template<>
struct short_vector<unsigned int, 2>;
template<>
struct short_vector<unsigned int, 3>;
template<>
struct short_vector<unsigned int, 4>;
template<>
struct short_vector<int, 1>;
template<>
struct short_vector<int, 2>;
template<>
struct short_vector<int, 3>;
template<>
struct short_vector<int, 4>;
template<>
struct short_vector<float, 1>;
template<>
struct short_vector<float, 2>;
template<>
struct short_vector<float, 3>;
template<>
struct short_vector<float, 4>;
template<>
struct short_vector<unorm, 1>;
template<>
struct short_vector<unorm, 2>;
template<>
struct short_vector<unorm, 3>;
template<>
struct short_vector<unorm, 4>;
template<>
struct short_vector<norm, 1>;
template<>
struct short_vector<norm, 2>;
template<>
struct short_vector<norm, 3>;
template<>
struct short_vector<norm, 4>;
template<>
struct short_vector<double, 1>;
template<>
struct short_vector<double, 2>;
template<>
struct short_vector<double, 3>;
template<>
struct short_vector<double, 4>;
```

## Parameters

*\_Scalar\_type*

*\_Size*

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">short_vector::short_vector Constructor</a>	

## Inheritance Hierarchy

`short_vector`

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## short\_vector::short\_vector Constructor

```
short_vector();
```

## See also

[Concurrency::graphics Namespace](#)

# short\_vector\_traits Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

short\_vector\_traits allows retrieval of the underlying vector length and scalar type of a short vector type or a scalar type

## Syntax

```

template<
    typename T
>
struct short_vector_traits;
template<>
struct short_vector_traits<unsigned int>;
template<>
struct short_vector_traits<uint_2>;
template<>
struct short_vector_traits<uint_3>;
template<>
struct short_vector_traits<uint_4>;
template<>
struct short_vector_traits<int>;
template<>
struct short_vector_traits<int_2>;
template<>
struct short_vector_traits<int_3>;
template<>
struct short_vector_traits<int_4>;
template<>
struct short_vector_traits<float>;
template<>
struct short_vector_traits<float_2>;
template<>
struct short_vector_traits<float_3>;
template<>
struct short_vector_traits<float_4>;
template<>
struct short_vector_traits<unorm>;
template<>
struct short_vector_traits<unorm_2>;
template<>
struct short_vector_traits<unorm_3>;
template<>
struct short_vector_traits<unorm_4>;
template<>
struct short_vector_traits<norm>;
template<>
struct short_vector_traits<norm_2>;
template<>
struct short_vector_traits<norm_3>;
template<>
struct short_vector_traits<norm_4>;
template<>
struct short_vector_traits<double>;
template<>
struct short_vector_traits<double_2>;
template<>
struct short_vector_traits<double_3>;
template<>
struct short_vector_traits<double_4>;

```

### Parameters

T

## Members

### Public Typedefs

NAME	DESCRIPTION
value_type	

## Public Constructors

NAME	DESCRIPTION
<a href="#">short_vector_traits::short_vector_traits Constructor</a>	

## Public Constants

NAME	DESCRIPTION
<a href="#">short_vector_traits::size Constant</a>	

## Inheritance Hierarchy

`short_vector_traits`

## Requirements

**Header:** `amp_short_vectors.h`

**Namespace:** `Concurrency::graphics`

## `short_vector_traits::short_vector_traits` Constructor

```
short_vector_traits();
```

## `short_vector_traits::size` Constant

```
static int const size = 1;
```

## See also

[Concurrency::graphics Namespace](#)

# texture Class

3/4/2019 • 7 minutes to read • [Edit Online](#)

A texture is a data aggregate on an `accelerator_view` in the extent domain. It is a collection of variables, one for each element in an extent domain. Each variable holds a value corresponding to C++ primitive type ( `unsigned int`, `int`, `float`, `double` ), a scalar type ( `norm`, or `unorm` ), or a short vector type.

## Syntax

```
template <typename value_type, int _Rank>
class texture;
```

### Parameters

*value\_type*

The type of the elements in the texture.

*\_Rank*

The rank of the texture.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>scalar_type</code>	Scalar types.
<code>value_type</code>	Value types.

### Public Constructors

NAME	DESCRIPTION
<a href="#">texture Constructor</a>	Initializes a new instance of the <code>texture</code> class.
<a href="#">~texture Destructor</a>	Destroys the <code>texture</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">copy_to</a>	Copies the <code>texture</code> object to the destination, by doing a deep copy.
<a href="#">data</a>	Returns a CPU pointer to the raw data of this texture.
<a href="#">get</a>	Returns the value of the element at the specified index.

NAME	DESCRIPTION
<a href="#">get_associated_accelerator_view</a>	Returns the <a href="#">accelerator_view</a> that is the preferred target for this texture to be copied to.
<a href="#">get_depth_pitch</a>	Returns the number of bytes between each depth slice in a 3D staging texture on the CPU.
<a href="#">get_row_pitch</a>	Returns the number of bytes between each row in a 2D or 3D staging texture on the CPU.
<a href="#">set</a>	Sets the value of the element at the specified index.

### Public Operators

NAME	DESCRIPTION
<a href="#">operator()</a>	Returns the element value that is specified by the parameters.
<a href="#">operator[]</a>	Returns the element that is at the specified index.
<a href="#">operator=</a>	Copies the specified <a href="#">texture</a> object to this one.

### Public Constants

NAME	DESCRIPTION
<a href="#">rank Constant</a>	Gets the rank of the <code>texture</code> object.

### Public Data Members

NAME	DESCRIPTION
<a href="#">associated_accelerator_view</a>	Gets the <a href="#">accelerator_view</a> that is the preferred target for this texture to be copied to.
<a href="#">depth_pitch</a>	Gets the number of bytes between each depth slice in a 3D staging texture on the CPU.
<a href="#">row_pitch</a>	Gets the number of bytes between each row in a 2D or 3D staging texture on the CPU.

## Inheritance Hierarchy

`_Texture_base`

`texture`

## Requirements

**Header:** `amp_graphics.h`

**Namespace:** `Concurrency::graphics`

## ~texture

Destroys the `texture` object.

```
~texture() restrict(cpu);
```

## associated\_accelerator\_view

Gets the `accelerator_view` that is the preferred target for this texture to be copied to.

```
__declspec(property(get= get_associated_accelerator_view)) Concurrency::accelerator_view  
associated_accelerator_view;
```

## copy\_to

Copies the `texture` object to the destination, by doing a deep copy.

```
void copy_to(texture& _Dest) const;  
void copy_to(writeonly_texture_view<value_type, _Rank>& _Dest) const;
```

### Parameters

*\_Dest*

The object to copy to.

*\_Rank*

The rank of the texture.

*value\_type*

The type of the elements in the texture.

## data

Returns a CPU pointer to the raw data of this texture.

```
void* data() restrict(cpu);  
  
const void* data() const restrict(cpu);
```

### Return Value

A pointer to the raw data of the texture.

## depth\_pitch

Gets the number of bytes between each depth slice in a 3D staging texture on the CPU.

```
__declspec(property(get= get_depth_pitch)) unsigned int depth_pitch;
```

## get

Returns the value of the element at the specified index.

```
const value_type get(const index<_Rank>& _Index) const restrict(amp);
```

### Parameters

*\_Index*

The index of the element.

### Return Value

The value of the element at the specified index.

## get\_associated\_accelerator\_view

Returns the `accelerator_view` that is the preferred target for this texture to be copied to.

```
Concurrency::accelerator_view get_associated_accelerator_view() const restrict(cpu);
```

### Return Value

The `accelerator_view` that is the preferred target for this texture to be copied to.

## get\_depth\_pitch

Returns the number of bytes between each depth slice in a 3D staging texture on the CPU.

```
unsigned int get_depth_pitch() const restrict(cpu);
```

### Return Value

The number of bytes between each depth slice in a 3D staging texture on the CPU.

## get\_row\_pitch

Returns the number of bytes between each row in a 2-dimensional staging texture, or between each row of a depth slice in 3-dimensional staging texture.

```
unsigned int get_row_pitch() const restrict(cpu);
```

### Return Value

The number of bytes between each row in a 2-dimensional staging texture, or between each row of a depth slice in 3-dimensional staging texture.

## operator()

Returns the element value that is specified by the parameters.

```

const value_type operator() (
    const index<_Rank>& _Index) const restrict(amp);

const value_type operator() (
    int _I0) const restrict(amp);

const value_type operator() (
    int _I0,
    int _I1) const restrict(amp);

const value_type operator() (
    int _I0,
    int _I1,
    int _I2) const restrict(amp);

```

### Parameters

*\_Index*

The index.

*\_I0*

The most-significant component of the index.

*\_I1*

The next-to-most-significant component of the index.

*\_I2*

The least-significant component of the index.

*\_Rank*

The rank of the index.

### Return Value

The element value that is specified by the parameters.

## operator[]

Returns the element that is at the specified index.

```

const value_type operator[] (const index<_Rank>& _Index) const restrict(amp);

const value_type operator[] (int _I0) const restrict(amp);

```

### Parameters

*\_Index*

The index.

*\_I0*

The index.

### Return Value

The element that is at the specified index.

## operator=

Copies the specified [texture](#) object to this one.

```
texture& operator= (
    const texture& _Other);

texture& operator= (
    texture<value_type, _Rank>&& _Other);
```

### Parameters

*\_Other*

The `texture` object to copy from.

### Return Value

A reference to this `texture` object.

## rank

Gets the rank of the `texture` object.

```
static const int rank = _Rank;
```

## row\_pitch

Gets the number of bytes between each row in a 2D or 3D staging texture on the CPU.

```
__declspec(property(get= get_row_pitch)) unsigned int row_pitch;
```

## set

Sets the value of the element at the specified index.

```
void set(
    const index<_Rank>& _Index,
    const value_type& value) restrict(amp);
```

### Parameters

*\_Index*

The index of the element.

*\_Rank*

The rank of the index.

*value*

The new value of the element.

## texture

Initializes a new instance of the `texture` class.

```
texture(const Concurrency::extent<_Rank>& _Ext) restrict(cpu);

texture(int _E0) restrict(cpu);

texture(int _E0, int _E1) restrict(cpu);
```

```

texture(int _E0, int _E1, int _E2) restrict(cpu);

texture(
    const Concurrency::extent<_Rank>& _Ext,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

texture(
    int _E0,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

texture(
    int _E0,
    int _E1,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

texture(
    int _E0,
    int _E1,
    int _E2,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

template<typename _Input_iterator>
texture(
    const Concurrency::extent<_Rank>& _Ext,
    _Input_iterator _Src_first,
    _Input_iterator _Src_last) restrict(cpu);

template<typename _Input_iterator>
texture(
    int _E0, _Input_iterator _Src_first, _Input_iterator _Src_last) restrict(cpu);

template<typename _Input_iterator>
texture(
    int _E0,
    int _E1,
    _Input_iterator _Src_first,
    _Input_iterator _Src_last) restrict(cpu);

template<typename _Input_iterator>
texture(
    int _E0,
    int _E1,
    int _E2,
    _Input_iterator _Src_first,
    _Input_iterator _Src_last) restrict(cpu);

template<typename _Input_iterator>
texture(
    const Concurrency::extent<_Rank>& _Ext,
    _Input_iterator _Src_first,
    _Input_iterator _Src_last,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

template<typename _Input_iterator>
texture(
    int _E0,
    _Input_iterator _Src_first,
    _Input_iterator _Src_last,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

template<typename _Input_iterator>
texture(
    int _E0,
    int _E1,
    _Input_iterator _Src_first,
    _Input_iterator _Src_last,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

template<typename _Input_iterator>

```

```

texture(
    int _E0,
    int _E1,
    int _E2,
    _Input_iterator _Src_first,
    _Input_iterator _Src_last,
    const Concurrency::accelerator_view& _Av) restrict(cpu) ;

texture(
    int _E0,
    unsigned int _Bits_per_scalar_element) restrict(cpu);

texture(
    int _E0,
    int _E1,
    unsigned int _Bits_per_scalar_element) restrict(cpu);

texture(
    int _E0,
    int _E1,
    int _E2,
    unsigned int _Bits_per_scalar_element) restrict(cpu);

texture(
    const Concurrency::extent<_Rank>& _Ext,
    unsigned int _Bits_per_scalar_element,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

texture(
    int _E0,
    unsigned int _Bits_per_scalar_element,
    const Concurrency::accelerator_view& _Av) ;

texture(
    int _E0,
    int _E1,
    unsigned int _Bits_per_scalar_element,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

texture(
    int _E0,
    int _E1,
    int _E2,
    unsigned int _Bits_per_scalar_element,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

texture(
    const Concurrency::extent<_Rank>& _Ext,
    _In_ void* _Source,
    unsigned int _Src_byte_size,
    unsigned int _Bits_per_scalar_element) restrict(cpu);

texture(
    int _E0,
    _In_ void* _Source,
    unsigned int _Src_byte_size,
    unsigned int _Bits_per_scalar_element) restrict(cpu);

texture(
    int _E0,
    int _E1,
    _In_ void* _Source,
    unsigned int _Src_byte_size,
    unsigned int _Bits_per_scalar_element) restrict(cpu);

texture(
    int _E0,
    int _E1,
    int _E2,

```

```

    _In_ void* _Source,
    unsigned int _Src_byte_size,
    unsigned int _Bits_per_scalar_element) restrict(cpu);

texture(
    const Concurrency::extent<_Rank>& _Ext,
    _In_ void* _Source,
    unsigned int _Src_byte_size,
    unsigned int _Bits_per_scalar_element,
    const Concurrency::accelerator_view& _Av) ;

texture(
    int _E0,
    _In_ void* _Source,
    unsigned int _Src_byte_size,
    unsigned int _Bits_per_scalar_element,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

texture(
    int _E0,
    int _E1,
    _In_ void* _Source,
    unsigned int _Src_byte_size,
    unsigned int _Bits_per_scalar_element,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

texture(
    int _E0,
    int _E1,
    int _E2,
    _In_ void* _Source,
    unsigned int _Src_byte_size,
    unsigned int _Bits_per_scalar_element,
    const Concurrency::accelerator_view& _Av) restrict(cpu);

texture(
    const texture& _Src,
    const Concurrency::accelerator_view& _Acc_view);

texture(
    texture&& _Other);

texture(
    const Concurrency::extent<_Rank>& _Ext,
    unsigned int _Bits_per_scalar_element,
    const Concurrency::accelerator_view& _Av);

texture(
    const texture& _Src);

```

## Parameters

### *\_Acc\_view*

The [accelerator\\_view](#) that specifies the location of the texture.

### *\_Av*

The [accelerator\\_view](#) that specifies the location of the texture.

### *\_Associated\_av*

An [accelerator\\_view](#) that specifies the preferred target for copies to or from this texture.

### *\_Bits\_per\_scalar\_element*

The number of bits per each scalar element in the underlying scalar type of the texture. In general, supported values are 8, 16, 32, and 64. If 0 is specified, the number of bits is the same as the underlying scalar\_type. 64 is only valid for double-based textures.

*\_Ext*

The extent in each dimension of the texture.

*\_E0*

The most significant component of the texture.

*\_E1*

The next-to-most-significant component of the texture.

*\_E2*

The least significant component of the extent of the texture.

*\_Input\_iterator*

The type of the input iterator.

*\_Mipmap\_levels*

The number of mipmap levels in the underlying texture. If 0 is specified, the texture will have the full range of mipmap levels down to the smallest possible size for the specified extent.

*\_Rank*

The rank of the extent.

*\_Source*

A pointer to a host buffer.

*\_Src*

To texture to copy.

*\_Src\_byte\_size*

The number of bytes in the source buffer.

*\_Src\_first*

A beginning iterator into the source container.

*\_Src\_last*

An ending iterator into the source container.

*\_Other*

Other data source.

*\_Rank*

The rank of the section.

## See also

[Concurrency::graphics Namespace](#)

# texture\_view Class

3/4/2019 • 7 minutes to read • [Edit Online](#)

Provides read access and write access to a texture. `texture_view` can only be used to read textures whose value type is `int`, `unsigned int`, or `float` that have default 32-bit bpse. To read other texture formats, use `texture_view<const value_type, _Rank>`.

## Syntax

```
template<typename value_type,int _Rank>
class texture_view;

template<typename value_type, int _Rank>
class texture_view
    : public details::_Texture_base<value_type, _Rank>;

template<typename value_type, int _Rank>
class texture_view<const value_type, _Rank>
    : public details::_Texture_base<value_type, _Rank>;
```

### Parameters

*value\_type*

The type of the elements in the texture aggregate.

*\_Rank*

The rank of the `texture_view`.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	The type of the elements in the texture aggregates.
<code>coordinates_type</code>	The type of the coordinate used to specify a texel in the <code>texture_view</code> —that is, a <code>short_vector</code> that has the same rank as the associated texture that has a value type of <code>float</code> .
<code>gather_return_type</code>	The return type used for gather operations—that is, a rank 4 <code>short_vector</code> that holds the four homogenous color components gathered from the four texel values sampled.

### Public Constructors

NAME	DESCRIPTION
<a href="#">texture_view Constructor</a>	Overloaded. Constructs a <code>texture_view</code> instance.
<a href="#">~texture_view Destructor</a>	Destroys the <code>texture_view</code> instance.

## Public Methods

NAME	DESCRIPTION
<a href="#">gather_alpha</a>	Overloaded. Samples the texture at the specified coordinates by using the specified sampling configuration and returns the alpha (w) components of the four sampled texels.
<a href="#">gather_blue</a>	Overloaded. Samples the texture at the specified coordinates by using the specified sampling configuration and returns the blue (z) components of the four sampled texels.
<a href="#">gather_green</a>	Overloaded. Samples the texture at the specified coordinates by using the specified sampling configuration and returns the green (y) components of the four sampled texels.
<a href="#">gather_red</a>	Overloaded. Samples the texture at the specified coordinates by using the specified sampling configuration and returns the red (x) components of the four sampled texels.
<a href="#">get</a>	Overloaded. Gets the element value by index.
<a href="#">sample</a>	Overloaded. Samples the texture at the specified coordinates and level of detail by using the specified sampling configuration.
<a href="#">set</a>	Sets the value of an element by index.

## Public Operators

NAME	DESCRIPTION
<a href="#">operator()</a>	Overloaded. Gets the element value by index.
<a href="#">operator[]</a>	Overloaded. Gets the element value by index.
<a href="#">operator=</a>	Overloaded. Assignment operator.

## Public Data Members

NAME	DESCRIPTION
<a href="#">value_type</a>	The value type of the elements of the <code>texture_view</code> .

## Inheritance Hierarchy

`_Texture_base`

`texture_view`

## Requirements

**Header:** amp\_graphics.h

**Namespace:** concurrency::graphics

## ~texture\_view

Destroys the `texture_view` instance.

```
~texture_view() restrict(amp, cpu);
```

## texture\_view

Constructs a `texture_view` instance.

```
texture_view(// [1] constructor  
            texture<value_type, _Rank>& _Src) restrict(amp);  
  
texture_view(// [2] constructor  
            texture<value_type, _Rank>& _Src,  
            unsigned int _Mipmap_level = 0) restrict(cpu);  
  
texture_view(// [3] constructor  
            const texture<value_type, _Rank>& _Src) restrict(amp);  
  
texture_view(// [4] constructor  
            const texture<value_type, _Rank>& _Src,  
            unsigned int _Most_detailed_mip,  
            unsigned int _Mip_levels) restrict(cpu);  
  
texture_view(// [5] copy constructor  
            const texture_view<value_type, _Rank>& _Other) restrict(amp, cpu);  
  
texture_view(// [6] copy constructor  
            const texture_view<const value_type, _Rank>& _Other) restrict(amp, cpu);  
  
texture_view(// [7] copy constructor  
            const texture_view<const value_type, _Rank>& _Other,  
            unsigned int _Most_detailed_mip,  
            unsigned int _Mip_levels) restrict(cpu);
```

### Parameters

#### *\_Src*

[1, 2] Constructor The `texture` on which the writable `texture_view` is created.

[3, 4] Constructor The `texture` on which the non-writable `texture_view` is created.

#### *\_Other*

[5] Copy Constructor The source writable `texture_view`.

[6, 7] Copy Constructor The source non-writable `texture_view`.

#### *\_Mipmap\_level*

The specific mipmap level on the source `texture` that this writeable `texture_view` binds to. The default value is 0, which represents the top level (most detailed) mip level.

#### *\_Most\_detailed\_mip*

Top level (most detailed) mip level for the view, relative to the specified `texture_view` object.

#### *\_Mip\_levels*

The number of mipmap levels accessible through the `texture_view`.

## gather\_red

Samples the texture at the specified coordinates by using the specified sampling configuration and returns the red (x) components of the four sampled texels.

```
const gather_return_type gather_red(
    const sampler& _Sampler,
    const coordinates_type& _Coord) const restrict(amp);

template<
    address_mode _Address_mode = address_clamp
>
const gather_return_type gather_red(
    const coordinates_type& _Coord) const restrict(amp);
```

### Parameters

#### *\_Address\_mode*

The address mode to use to sample the `texture_view`. The address mode is the same for all dimensions.

#### *\_Sampler*

The sampler configuration to use to sample the `texture_view`.

#### *\_Coord*

The coordinates to take the sample from. Fractional coordinate values are used to interpolate between sample texels.

### Return Value

A rank 4 short vector containing the red (x) component of the 4 sampled texel values.

## gather\_green

Samples the texture at the specified coordinates by using the specified sampling configuration and returns the green (y) components of the four sampled texels.

```
const gather_return_type gather_green(
    const sampler& _Sampler,
    const coordinates_type& _Coord) const restrict(amp);

template<
    address_mode _Address_mode = address_clamp
>
const gather_return_type gather_green(
    const coordinates_type& _Coord) const restrict(amp);
```

### Parameters

#### *\_Address\_mode*

The address mode to use to sample the `texture_view`. The address mode is the same for all dimensions.

#### *\_Sampler*

The sampler configuration to use to sample the `texture_view`.

#### *\_Coord*

The coordinates to take the sample from. Fractional coordinate values are used to interpolate between sample texels.

### Return Value

A rank 4 short vector containing the green (y) component of the 4 sampled texel values.

## gather\_blue

Samples the texture at the specified coordinates by using the specified sampling configuration and returns the blue (z) components of the four sampled texels.

```
const gather_return_type gather_blue(
    const sampler& _Sampler,
    const coordinates_type& _Coord) const restrict(amp);

template<
    address_mode _Address_mode = address_clamp
>
const gather_return_type gather_blue(
    const coordinates_type& _Coord) const restrict(amp);
```

### Parameters

#### *\_Address\_mode*

The address mode to use to sample the `texture_view`. The address mode is the same for all dimensions.

#### *\_Sampler*

The sampler configuration to use to sample the `texture_view`.

#### *\_Coord*

The coordinates to take the sample from. Fractional coordinate values are used to interpolate between sample texels.

### Return Value

A rank 4 short vector containing the red (x) component of the 4 sampled texel values.

## gather\_alpha

Samples the texture at the specified coordinates by using the specified sampling configuration and returns the alpha (w) components of the four sampled texels.

```
const gather_return_type gather_alpha(
    const sampler& _Sampler,
    const coordinates_type& _Coord) const restrict(amp);

template<
    address_mode _Address_mode = address_clamp
>
const gather_return_type gather_alpha(
    const coordinates_type& _Coord) const restrict(amp);
```

### Parameters

#### *\_Address\_mode*

The address mode to use to sample the `texture_view`. The address mode is the same for all dimensions.

#### *\_Sampler*

The sampler configuration to use to sample the `texture_view`.

#### *\_Coord*

The coordinates to take the sample from. Fractional coordinate values are used to interpolate between sample texels.

### Return Value

A rank 4 short vector containing the alpha (w) component of the 4 sampled texel values.

# get

Gets the value of the element at the specified index.

```
const value_type get(
    const index<_Rank>& _Index) const restrict(amp);

value_type get(
    const index<_Rank>& _Index,
    unsigned int _Mip_level = 0) const restrict(amp);
```

## Parameters

### *\_Index*

The index of the element to get, possibly multi-dimensional.

### *\_Mip\_level*

The mipmap level from which we should get the value. The default value 0 represents the most detailed mipmap level.

## Return Value

The value of the element.

# operator=

Assigns a view of the same texture as the specified `texture_view` to this `texture_view` instance.

```
texture_view<value_type, _Rank>& operator= (// [1] copy constructor
    const texture_view<value_type, _Rank>& _Other) restrict(amp, cpu);

texture_view<const value_type, _Rank>& operator= (// [2] copy constructor
    const texture_view<value_type, _Rank>& _Other) restrict(cpu);

texture_view<const value_type, _Rank>& operator= (// [3] copy constructor
    const texture_view<const value_type, _Rank>& _Other) restrict(amp, cpu);
```

## Parameters

### *\_Other*

[1, 2] Copy Constructor A writable `texture_view` object.

[3] Copy Constructor A non-writable `texture_view` object.

## Return Value

A reference to this `texture_view` instance.

# operator[]

Returns the element value by index.

```
const value_type operator[] (const index<_Rank>& _Index) const restrict(amp);

const value_type operator[] (int _I0) const restrict(amp);

value_type operator[] (const index<_Rank>& _Index) const restrict(amp);

value_type operator[] (int _I0) const restrict(amp);
```

## Parameters

*\_Index*

The index, possibly multi-dimensional.

*\_I0*

The one-dimensional index.

## Return Value

The element value indexed by `_Index`.

# operator()

Returns the element value by index.

```
const value_type operator() (
    const index<_Rank>& _Index) const restrict(amp);

const value_type operator() (
    int _I0) const restrict(amp);

const value_type operator() (
    int _I0, int _I1) const restrict(amp);

const value_type operator() (
    int _I0,
    int _I1,
    int _I2) const restrict(amp);

value_type operator() (
    const index<_Rank>& _Index) const restrict(amp);

value_type operator() (
    int _I0) const restrict(amp);

value_type operator() (
    int _I0,
    int _I1) const restrict(amp);

value_type operator() (
    int _I0,
    int _I1,
    int _I2) const restrict(amp);
```

## Parameters

*\_Index*

The index, possibly multi-dimensional.

*\_I0*

The most-significant component of the index.

*\_I1*

The next-to-most-significant component of the index.

*\_I2*

The least-significant component of the index.

## Return Value

The element value indexed by `_Index`.

# sample

Samples the texture at the specified coordinates and level of detail by using the specified sampling configuration.

```
value_type sample(  
    const sampler& _Sampler,  
    const coordinates_type& _Coord,  
    float _Level_of_detail = 0.0f) const restrict(amp);  
  
template<  
    filter_mode _Filter_mode = filter_linear,  
    address_mode _Address_mode = address_clamp  
>  
value_type sample(  
    const coordinates_type& _Coord,  
    float _Level_of_detail = 0.0f) const restrict(amp);
```

## Parameters

### *\_Filter\_mode*

The filter mode to use to sample the texture\_view. The filter mode is the same for the minimization, maximization, and mipmap filters.

### *\_Address\_mode*

The address mode to use to sample the texture\_view. The address mode is the same for all dimensions.

### *\_Sampler*

The sampler configuration to use to sample the texture\_view.

### *\_Coord*

The coordinates to take the sample from. Fractional coordinate values are used to interpolate between texel values.

### *\_Level\_of\_detail*

The value specifies the mipmap level to sample from. Fractional values are used to interpolate between two mipmap levels. The default level of detail is 0, which represents the most detailed mip level.

## Return Value

The interpolated sample value.

# set

Sets the value of the element at the specified index to the specified value.

```
void set(  
    const index<Rank>& _Index,  
    const value_type& value) const restrict(amp);
```

## Parameters

### *\_Index*

The index of the element to set, possibly multi-dimensional.

### *value*

The value to set the element to.

# value\_type

The value type of the elements of the texture\_view.

```
typedef typename const value_type value_type;
```

## See also

[Concurrency::graphics Namespace](#)

# writeonly\_texture\_view Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Provides writeonly access to a texture.

## Syntax

```
template <
    typename value_type,
    int _Rank
>
class writeonly_texture_view;

template <
    typename value_type,
    int _Rank
>
class writeonly_texture_view<value_type, _Rank> : public details::_Texture_base<value_type, _Rank>;
```

### Parameters

*value\_type*

The type of the elements in the texture.

*\_Rank*

The rank of the texture.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>scalar_type</code>	
<code>value_type</code>	The type of the elements in the texture.

### Public Constructors

NAME	DESCRIPTION
<a href="#">writeonly_texture_view Constructor</a>	Initializes a new instance of the <code>writeonly_texture_view</code> class.
<a href="#">~writeonly_texture_view Destructor</a>	Destroys the <code>writeonly_texture_view</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">set</a>	Sets the value of the element at the specified index.

### Public Operators

NAME	DESCRIPTION
<code>operator=</code>	Copies the specified <code>writeonly_texture_view</code> object to this one.

### Public Constants

NAME	DESCRIPTION
<code>rank Constant</code>	Gets the rank of the <code>writeonly_texture_view</code> object.

## Inheritance Hierarchy

`_Texture_base`

`writeonly_texture_view`

## Requirements

**Header:** `amp_graphics.h`

**Namespace:** `Concurrency::graphics`

## `~writeonly_texture_view`

Destroys the `writeonly_texture_view` object.

```
~writeonly_texture_view() restrict(amp,cpu);
```

## `operator=`

Copies the specified `writeonly_texture_view` object to this one.

```
writeonly_texture_view<value_type, _Rank>& operator= (
    const writeonly_texture_view<value_type, _Rank>& _Other) restrict(amp,cpu);
```

### Parameters

*\_Other*

`writeonly_texture_view` object to copy from.

### Return Value

A reference to this `writeonly_texture_view` object.

## `rank`

Gets the rank of the `writeonly_texture_view` object.

```
static const int rank = _Rank;
```

## `set`

Sets the value of the element at the specified index.

```
void set(  
    const index<_Rank>& _Index,  
    const value_type& value) const restrict(amp);
```

### Parameters

*\_Index*

The index of the element.

*value*

The new value of the element.

## writeonly\_texture\_view

Initializes a new instance of the `writeonly_texture_view` class.

```
writeonly_texture_view(  
    texture<value_type,  
    _Rank>& _Src) restrict(amp);  
  
writeonly_texture_view(  
    const writeonly_texture_view<value_type,  
    _Rank>& _Src) restrict(amp,cpu);
```

### Parameters

*\_Rank*

The rank of the texture.

*value\_type*

The type of the elements in the texture.

*\_Src*

The texture that is used to create the `writeonly_texture_view`.

## See also

[Concurrency::graphics Namespace](#)

# uint\_2 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of two unsigned integers.

## Syntax

```
class uint_2;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">uint_2 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>uint_2::get_x</code>	
<code>uint_2::get_xy</code>	
<code>uint_2::get_y</code>	
<code>uint_2::get_yx</code>	
<code>uint_2::ref_g_Method</code>	
<code>uint_2::ref_r_Method</code>	
<code>uint_2::ref_x_Method</code>	
<code>uint_2::ref_y_Method</code>	
<code>uint_2::set_x</code>	
<code>uint_2::set_xy</code>	
<code>uint_2::set_y</code>	

NAME	DESCRIPTION
uint_2::set_yx	

### Public Operators

NAME	DESCRIPTION
uint_2::operator--	
uint_2::operator%=	
uint_2::operator&=	
uint_2::operator*=	
uint_2::operator/=	
uint_2::operator^=	
uint_2::operator =	
uint_2::operator~	
uint_2::operator++	
uint_2::operator+=	
uint_2::operator<<=	
uint_2::operator=	
uint_2::operator-=	
uint_2::operator>>=	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
uint_2::g	
uint_2::gr	
uint_2::r	
uint_2::rg	

NAME	DESCRIPTION
uint_2::x	
uint_2::xy	
uint_2::y	
uint_2::yx	

## Inheritance Hierarchy

uint\_2

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## uint\_2

Default constructor, initializes all elements with 0.

```

uint_2() restrict(amp,
    cpu);

uint_2(
    unsigned int _V0,
    unsigned int _V1) restrict(amp,
    cpu);

uint_2(
    unsigned int _V) restrict(amp,
    cpu);

uint_2(
    const uint_2& _Other) restrict(amp,
    cpu);

explicit inline uint_2(
    const int_2& _Other) restrict(amp,
    cpu);

explicit inline uint_2(
    const float_2& _Other) restrict(amp,
    cpu);

explicit inline uint_2(
    const unorm_2& _Other) restrict(amp,
    cpu);

explicit inline uint_2(
    const norm_2& _Other) restrict(amp,
    cpu);

explicit inline uint_2(
    const double_2& _Other) restrict(amp,
    cpu);

```

## Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 2;
```

## See also

[Concurrency::graphics Namespace](#)

# uint\_3 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of three unsigned integers.

## Syntax

```
class uint_3;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">uint_3 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>uint_3::get_x</code>	
<code>uint_3::get_xy</code>	
<code>uint_3::get_xyz</code>	
<code>uint_3::get_xz</code>	
<code>uint_3::get_xzy</code>	
<code>uint_3::get_y</code>	
<code>uint_3::get_yx</code>	
<code>uint_3::get_yxz</code>	
<code>uint_3::get_yz</code>	
<code>uint_3::get_yzx</code>	
<code>uint_3::get_z</code>	

NAME	DESCRIPTION
uint_3::get_zx	
uint_3::get_zxy	
uint_3::get_zy	
uint_3::get_zyx	
uint_t::ref_b	
uint_t::ref_g	
uint_t::ref_r	
uint_t::ref_x	
uint_t::ref_y	
uint_t::ref_z	
uint_3::set_x	
uint_3::set_xy	
uint_3::set_xyz	
uint_3::set_xz	
uint_3::set_xzy	
uint_3::set_y	
uint_3::set_yx	
uint_3::set_yxz	
uint_3::set_yz	
uint_3::set_yzx	
uint_3::set_z	
uint_3::set_zx	
uint_3::set_zxy	
uint_3::set_zy	
uint_3::set_zyx	

## Public Operators

NAME	DESCRIPTION
uint_3::operator--	
uint_3::operator% =	
uint_3::operator& =	
uint_3::operator* =	
uint_3::operator/ =	
uint_3::operator^ =	
uint_3::operator  =	
uint_3::operator~	
uint_3::operator++	
uint_3::operator+ =	
uint_3::operator<< =	
uint_3::operator=	
uint_3::operator- =	
uint_3::operator>> =	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
uint_3::b	
uint_3::bg	
uint_3::bgr	
uint_3::br	
uint_3::brg	
uint_3::g	
uint_3::gb	

NAME	DESCRIPTION
uint_3::gbr	
uint_3::gr	
uint_3::grb	
uint_3::r	
uint_3::rb	
uint_3::rbg	
uint_3::rg	
uint_3::rgb	
uint_3::x	
uint_3::xy	
uint_3::xyz	
uint_3::xz	
uint_3::xzy	
uint_3::y	
uint_3::yx	
uint_3::yz	
uint_3::yz	
uint_3::yzx	
uint_3::z	
uint_3::zx	
uint_3::zxy	
uint_3::zy	
uint_3::zyx	

## Inheritance Hierarchy

uint\_3

# Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## uint\_3

Default constructor, initializes all elements with 0.

```
uint_3() restrict(amp,
    cpu);

uint_3(
    unsigned int _V0,
    unsigned int _V1,
    unsigned int _V2) restrict(amp,
    cpu);

uint_3(
    unsigned int _V) restrict(amp,
    cpu);

uint_3(
    const uint_3& _Other) restrict(amp,
    cpu);

explicit inline uint_3(
    const int_3& _Other) restrict(amp,
    cpu);

explicit inline uint_3(
    const float_3& _Other) restrict(amp,
    cpu);

explicit inline uint_3(
    const unorm_3& _Other) restrict(amp,
    cpu);

explicit inline uint_3(
    const norm_3& _Other) restrict(amp,
    cpu);

explicit inline uint_3(
    const double_3& _Other) restrict(amp,
    cpu);
```

### Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V2*

The value to initialize element 2.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 3;
```

## See also

[Concurrency::graphics Namespace](#)

# uint\_4 Class

3/28/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of four unsigned integers.

## Syntax

```
class uint_4;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">uint_4 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>uint_4::get_w</code>	
<code>uint_4::get_wx</code>	
<code>uint_4::get_wxy</code>	
<code>uint_4::get_wxyz</code>	
<code>uint_4::get_wxz</code>	
<code>uint_4::get_wxzy</code>	
<code>uint_4::get_wy</code>	
<code>uint_4::get_wyx</code>	
<code>uint_4::get_wyxz</code>	
<code>uint_4::get_wyz</code>	
<code>uint_4::get_wyzx</code>	

NAME	DESCRIPTION
uint_4::get_wz	
uint_4::get_wzx	
uint_4::get_wzxy	
uint_4::get_wzy	
uint_4::get_wzyx	
uint_4::get_x	
uint_4::get_xw	
uint_4::get_xwy	
uint_4::get_xwyz	
uint_4::get_xwz	
uint_4::get_xwzy	
uint_4::get_xy	
uint_4::get_xyw	
uint_4::get_xywz	
uint_4::get_xyz	
uint_4::get_xyzw	
uint_4::get_xz	
uint_4::get_xzw	
uint_4::get_xzwy	
uint_4::get_xzy	
uint_4::get_xzyw	
uint_4::get_y	
uint_4::get_yw	
uint_4::get_ywx	
uint_4::get_ywzx	

NAME	DESCRIPTION
uint_4::get_ywz	
uint_4::get_ywzx	
uint_4::get_yx	
uint_4::get_yxw	
uint_4::get_yxwz	
uint_4::get_yxz	
uint_4::get_yxzw	
uint_4::get_yz	
uint_4::get_yzw	
uint_4::get_yzwx	
uint_4::get_yzx	
uint_4::get_yzxw	
uint_4::get_z	
uint_4::get_zw	
uint_4::get_zwx	
uint_4::get_zwxy	
uint_4::get_zwy	
uint_4::get_zwyx	
uint_4::get_zx	
uint_4::get_zxw	
uint_4::get_zxwy	
uint_4::get_zxy	
uint_4::get_zxyw	
uint_4::get_zy	
uint_4::get_zyw	

NAME	DESCRIPTION
uint_4::get_zywx	
uint_4::get_zyx	
uint_4::get_zyxw	
uint_4::ref_a	
uint_4::ref_b	
uint_4::ref_g	
uint_4::ref_r	
uint_4::ref_w	
uint_4::ref_x	
uint_4::ref_y	
uint_4::ref_z	
uint_4::set_w	
uint_4::set_wx	
uint_4::set_wxy	
uint_4::set_wxyz	
uint_4::set_wxz	
uint_4::set_wzy	
uint_4::set_wy	
uint_4::set_wyx	
uint_4::set_wyxz	
uint_4::set_wyz	
uint_4::set_wyzx	
uint_4::set_wz	
uint_4::set_wzx	
uint_4::set_wzxy	

NAME	DESCRIPTION
uint_4::set_wzy	
uint_4::set_wzyx	
uint_4::set_x	
uint_4::set_xw	
uint_4::set_xwy	
uint_4::set_xwyz	
uint_4::set_xwz	
uint_4::set_xwzy	
uint_4::set_xy	
uint_4::set_xyw	
uint_4::set_xywz	
uint_4::set_xyz	
uint_4::set_xyzw	
uint_4::set_xz	
uint_4::set_xzw	
uint_4::set_xzwy	
uint_4::set_xzy	
uint_4::set_xzyw	
uint_4::set_y	
uint_4::set_yw	
uint_4::set_ywx	
uint_4::set_ywzx	
uint_4::set_ywz	
uint_4::set_ywzx	
uint_4::set_yx	

NAME	DESCRIPTION
uint_4::set_yxw	
uint_4::set_yxwz	
uint_4::set_yxz	
uint_4::set_yxzw	
uint_4::set_yz	
uint_4::set_yzw	
uint_4::set_yzwx	
uint_4::set_yzx	
uint_4::set_yzxw	
uint_4::set_z	
uint_4::set_zw	
uint_4::set_zwx	
uint_4::set_zwxy	
uint_4::set_zwy	
uint_4::set_zwyx	
uint_4::set_zx	
uint_4::set_zxw	
uint_4::set_zxwy	
uint_4::set_zxy	
uint_4::set_zxyw	
uint_4::set_zy	
uint_4::set_zyw	
uint_4::set_zywx	
uint_4::set_zyx	
uint_4::set_zyxw	

## Public Operators

NAME	DESCRIPTION
uint_4::operator-	
uint_4::operator--	
uint_4::operator*= uint_4::operator/=	
uint_4::operator++	
uint_4::operator+=	
uint_4::operator=	
uint_4::operator-=	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
uint_4::a	
uint_4::ab	
uint_4::abg	
uint_4::abgr	
uint_4::abr	
uint_4::abrg	
uint_4::ag	
uint_4::agb	
uint_4::agbr	
uint_4::agr	
uint_4::agrb	
uint_4::ar	
uint_4::arb	

NAME	DESCRIPTION
uint_4::arbg	
uint_4::arg	
uint_4::argb	
uint_4::b	
uint_4::ba	
uint_4::bag	
uint_4::bagr	
uint_4::bar	
uint_4::barg	
uint_4::bg	
uint_4::bga	
uint_4::bgar	
uint_4::bgr	
uint_4::bgra	
uint_4::br	
uint_4::bra	
uint_4::brag	
uint_4::brg	
uint_4::brga	
uint_4::g	
uint_4::ga	
uint_4::gab	
uint_4::gabr	
uint_4::gar	
uint_4::garb	

NAME	DESCRIPTION
uint_4::gb	
uint_4::gba	
uint_4::gbar	
uint_4::gbr	
uint_4::gbra	
uint_4::gr	
uint_4::gra	
uint_4::grab	
uint_4::grb	
uint_4::grba	
uint_4::r	
uint_4::ra	
uint_4::rab	
uint_4::rabg	
uint_4::rag	
uint_4::ragb	
uint_4::rb	
uint_4::rba	
uint_4::rbag	
uint_4::rbg	
uint_4::rbga	
uint_4::rg	
uint_4::rga	
uint_4::rgab	
uint_4::rgb	

NAME	DESCRIPTION
uint_4::rgba	
uint_4::w	
uint_4::wx	
uint_4::wxy	
uint_4::wxyz	
uint_4::wxz	
uint_4::wxzy	
uint_4::wy	
uint_4::wyx	
uint_4::wyz	
uint_4::wyzx	
uint_4::wz	
uint_4::wzx	
uint_4::wzxy	
uint_4::wzy	
uint_4::wzyx	
uint_4::x	
uint_4::xw	
uint_4::xwy	
uint_4::xwyz	
uint_4::xwz	
uint_4::xwzy	
uint_4::xy	
uint_4::xyw	

NAME	DESCRIPTION
uint_4::xywz	
uint_4::xyz	
uint_4::xyzw	
uint_4::xz	
uint_4::xzw	
uint_4::xzwy	
uint_4::xzy	
uint_4::xzyw	
uint_4::y	
uint_4::yw	
uint_4::ywx	
uint_4::ywxz	
uint_4::y wz	
uint_4::ywzx	
uint_4::yx	
uint_4::yxw	
uint_4::yxwz	
uint_4::yxz	
uint_4::yxzw	
uint_4::yz	
uint_4::yzw	
uint_4::yzwx	
uint_4::yzx	
uint_4::yzxw	
uint_4::z	

NAME	DESCRIPTION
uint_4::zw	
uint_4::zwx	
uint_4::zwxxy	
uint_4::zwy	
uint_4::zwyx	
uint_4::zx	
uint_4::zwx	
uint_4::zwy	
uint_4::zxy	
uint_4::zxyw	
uint_4::zy	
uint_4::zyw	
uint_4::zywx	
uint_4::zyx	
uint_4::zyxw	

## Inheritance Hierarchy

`uint_4`

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## uint\_4

Default constructor, initializes all elements with 0.

### Syntax

```

uint_4() restrict(amp,cpu);
uint_4(
    unsigned int _V0,
    unsigned int _V1,
    unsigned int _V2,
    unsigned int _V3
) restrict(amp,cpu);
uint_4(
    unsigned int _V
) restrict(amp,cpu);
uint_4(
    const uint_4& _Other
) restrict(amp,cpu);
explicit inline uint_4(
    const int_4& _Other
) restrict(amp,cpu);
explicit inline uint_4(
    const float_4& _Other
) restrict(amp,cpu);
explicit inline uint_4(
    const unorm_4& _Other
) restrict(amp,cpu);
explicit inline uint_4(
    const norm_4& _Other
) restrict(amp,cpu);
explicit inline uint_4(
    const double_4& _Other
) restrict(amp,cpu);

```

#### Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V2*

The value to initialize element 2.

*\_V3*

The value to initialize element 3.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

#### Syntax

```
static const int size = 4;
```

## See also

[Concurrency::graphics Namespace](#)

# unorm Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represent a unorm number. Each element is a floating point number in the range of [0.0f, 1.0f].

## Syntax

```
class unorm;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">unorm Constructor</a>	Overloaded. Default constructor. Initialize to 0.0f.

### Public Operators

NAME	DESCRIPTION
<code>unorm::operator--</code>	
<code>unorm::operator float</code>	Conversion operator. Convert the unorm number to a floating point value.
<code>unorm::operator* =</code>	
<code>unorm::operator/=</code>	
<code>unorm::operator++</code>	
<code>unorm::operator+ =</code>	
<code>unorm::operator=</code>	
<code>unorm::operator- =</code>	

## Inheritance Hierarchy

`unorm`

## Requirements

**Header:** `amp_short_vectors.h`

**Namespace:** `Concurrency::graphics`

# unorm

Default constructor. Initialize to 0.0f.

```
unorm(
    void) restrict(amp,
    cpu);

explicit unorm(
    float _V) restrict(amp,
    cpu);

explicit unorm(
    unsigned int _V) restrict(amp,
    cpu);

explicit unorm(
    int _V) restrict(amp,
    cpu);

explicit unorm(
    double _V) restrict(amp,
    cpu);

unorm(
    const unorm& _Other) restrict(amp,
    cpu);

inline explicit unorm(
    const norm& _Other) restrict(amp,
    cpu);
```

## Parameters

*\_V*

The value used to initialize.

*\_Other*

The norm object used to initialize.

## See also

[Concurrency::graphics Namespace](#)

# unorm\_2 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of two unsigned normal numbers.

## Syntax

```
class unorm_2;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">unorm_2 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>unorm_2::get_x</code>	
<code>unorm_2::get_xy</code>	
<code>unorm_2::get_y</code>	
<code>unorm_2::get_yx</code>	
<code>unorm_2::ref_g</code>	
<code>unorm_2::ref_r</code>	
<code>unorm_2::ref_x</code>	
<code>unorm_2::ref_y</code>	
<code>unorm_2::set_x</code>	
<code>unorm_2::set_xy</code>	
<code>unorm_2::set_y</code>	

NAME	DESCRIPTION
unorm_2::set_yx	

### Public Operators

NAME	DESCRIPTION
unorm_2::operator--	
unorm_2::operator* =	
unorm_2::operator/=	
unorm_2::operator++	
unorm_2::operator+ =	
unorm_2::operator=	
unorm_2::operator- =	

### Public Constants

NAME	DESCRIPTION
unorm_2::size Constant	

### Public Data Members

NAME	DESCRIPTION
unorm_2::g	
unorm_2::gr	
unorm_2::r	
unorm_2::rg	
unorm_2::x	
unorm_2::xy	
unorm_2::y	
unorm_2::yx	

## Inheritance Hierarchy

unorm\_2

# Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## unorm\_2

Default constructor, initializes all elements with 0.

```
unorm_2() restrict(amp,
    cpu);

unorm_2(
    unorm _V0,
    unorm _V1) restrict(amp,
    cpu);

unorm_2(
    float _V0,
    float _V1) restrict(amp,
    cpu);

unorm_2(
    unorm _V) restrict(amp,
    cpu);

explicit unorm_2(
    float _V) restrict(amp,
    cpu);

unorm_2(
    const unorm_2& _Other) restrict(amp,
    cpu);

explicit inline unorm_2(
    const uint_2& _Other) restrict(amp,
    cpu);

explicit inline unorm_2(
    const int_2& _Other) restrict(amp,
    cpu);

explicit inline unorm_2(
    const float_2& _Other) restrict(amp,
    cpu);

explicit inline unorm_2(
    const norm_2& _Other) restrict(amp,
    cpu);

explicit inline unorm_2(
    const double_2& _Other) restrict(amp,
    cpu);
```

### Parameters

**\_V0**

The value to initialize element 0.

**\_V1**

The value to initialize element 1.

**\_V**

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 2;
```

## See also

[Concurrency::graphics Namespace](#)

# unorm\_3 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of three unsigned normal numbers.

## Syntax

```
class unorm_3;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">unorm_3 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>unorm_3::get_x</code>	
<code>unorm_3::get_xy</code>	
<code>unorm_3::get_xyz</code>	
<code>unorm_3::get_xz</code>	
<code>unorm_3::get_xzy</code>	
<code>unorm_3::get_y</code>	
<code>unorm_3::get_yx</code>	
<code>unorm_3::get_yxz</code>	
<code>unorm_3::get_yz</code>	
<code>unorm_3::get_yzx</code>	
<code>unorm_3::get_z</code>	

NAME	DESCRIPTION
unorm_3::get_zx	
unorm_3::get_zxy	
unorm_3::get_zy	
unorm_3::get_zyx	
Unorm_3::ref_b	
Unorm_3::ref_g	
Unorm_3::ref_r	
Unorm_3::ref_x	
Unorm_3::ref_y	
Unorm_3::ref_z	
unorm_3::set_x	
unorm_3::set_xy	
unorm_3::set_xyz	
unorm_3::set_xz	
unorm_3::set_xzy	
unorm_3::set_y	
unorm_3::set_yx	
unorm_3::set_yxz	
unorm_3::set_yz	
unorm_3::set_yzx	
unorm_3::set_z	
unorm_3::set_zx	
unorm_3::set_zxy	
unorm_3::set_zy	
unorm_3::set_zyx	

**Public Operators**

NAME	DESCRIPTION
unorm_3::operator--	
unorm_3::operator* =	
unorm_3::operator/=	
unorm_3::operator+ +	
unorm_3::operator+ =	
unorm_3::operator=	
unorm_3::operator- =	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
unorm_3::b	
unorm_3::bg	
unorm_3::bgr	
unorm_3::br	
unorm_3::brg	
unorm_3::g	
unorm_3::gb	
unorm_3::gbr	
unorm_3::gr	
unorm_3::grb	
unorm_3::r	
unorm_3::rb	
unorm_3::rbg	
unorm_3::rg	

NAME	DESCRIPTION
unorm_3::rgb	
unorm_3::x	
unorm_3::xy	
unorm_3::xyz	
unorm_3::xz	
unorm_3::xzy	
unorm_3::y	
unorm_3::yx	
unorm_3::yz	
unorm_3::yzx	
unorm_3::z	
unorm_3::zx	
unorm_3::zxy	
unorm_3::zy	
unorm_3::zyx	

## Inheritance Hierarchy

`unorm_3`

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## unorm\_3

Default constructor, initializes all elements with 0.

```

unorm_3() restrict(amp,
    cpu);

unorm_3(
    unorm _V0,
    unorm _V1,
    unorm _V2) restrict(amp,
    cpu);

unorm_3(
    float _V0,
    float _V1,
    float _V2) restrict(amp,
    cpu);

unorm_3(
    unorm _V) restrict(amp,
    cpu);

explicit unorm_3(
    float _V) restrict(amp,
    cpu);

unorm_3(
    const unorm_3& _Other) restrict(amp,
    cpu);

explicit inline unorm_3(
    const uint_3& _Other) restrict(amp,
    cpu);

explicit inline unorm_3(
    const int_3& _Other) restrict(amp,
    cpu);

explicit inline unorm_3(
    const float_3& _Other) restrict(amp,
    cpu);

explicit inline unorm_3(
    const norm_3& _Other) restrict(amp,
    cpu);

explicit inline unorm_3(
    const double_3& _Other) restrict(amp,
    cpu);

```

## Parameters

*\_V0*

The value to initialize element 0.

*\_V1*

The value to initialize element 1.

*\_V2*

The value to initialize element 2.

*\_V*

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 3;
```

## See also

[Concurrency::graphics Namespace](#)

# unorm\_4 Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a short vector of four unsigned normal numbers.

## Syntax

```
class unorm_4;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>value_type</code>	

### Public Constructors

NAME	DESCRIPTION
<a href="#">unorm_4 Constructor</a>	Overloaded. Default constructor, initializes all elements with 0.

### Public Methods

NAME	DESCRIPTION
<code>unorm_4::get_w</code>	
<code>unorm_4::get_wx</code>	
<code>unorm_4::get_wxy</code>	
<code>unorm_4::get_wxyz</code>	
<code>unorm_4::get_wxz</code>	
<code>unorm_4::get_wxzy</code>	
<code>unorm_4::get_wy</code>	
<code>unorm_4::get_wyx</code>	
<code>unorm_4::get_wyz</code>	
<code>unorm_4::get_wyz</code>	
<code>unorm_4::get_wyzx</code>	

NAME	DESCRIPTION
unorm_4::get_wz	
unorm_4::get_wzx	
unorm_4::get_wzxy	
unorm_4::get_wzy	
unorm_4::get_wzyx	
unorm_4::get_x	
unorm_4::get_xw	
unorm_4::get_xwy	
unorm_4::get_xwyz	
unorm_4::get_xwz	
unorm_4::get_xwzy	
unorm_4::get_xy	
unorm_4::get_xyw	
unorm_4::get_xywz	
unorm_4::get_xyz	
unorm_4::get_xyzw	
unorm_4::get_xz	
unorm_4::get_xzw	
unorm_4::get_xzwy	
unorm_4::get_xzy	
unorm_4::get_xzyw	
unorm_4::get_y	
unorm_4::get_yw	
unorm_4::get_ywx	
unorm_4::get_ywxz	

NAME	DESCRIPTION
unorm_4::get_ywz	
unorm_4::get_ywzx	
unorm_4::get_yx	
unorm_4::get_yxw	
unorm_4::get_yxwz	
unorm_4::get_yxz	
unorm_4::get_yxzw	
unorm_4::get_yz	
unorm_4::get_yzw	
unorm_4::get_yzwx	
unorm_4::get_yzx	
unorm_4::get_yzwx	
unorm_4::get_z	
unorm_4::get_zw	
unorm_4::get_zwx	
unorm_4::get_zwxy	
unorm_4::get_zwy	
unorm_4::get_zwyx	
unorm_4::get_zx	
unorm_4::get_zxw	
unorm_4::get_zxwy	
unorm_4::get_zxy	
unorm_4::get_zxyw	
unorm_4::get_zy	
unorm_4::get_zyw	

NAME	DESCRIPTION
unorm_4::get_zywx	
unorm_4::get_zyx	
unorm_4::get_zyxw	
unorm_4::ref_a	
unorm_4::ref_b	
unorm_4::ref_g	
unorm_4::ref_r	
unorm_4::ref_w	
unorm_4::ref_x	
unorm_4::ref_y	
unorm_4::ref_z	
unorm_4::set_w	
unorm_4::set_wx	
unorm_4::set_wxy	
unorm_4::set_wxyz	
unorm_4::set_wxz	
unorm_4::set_wxzy	
unorm_4::set_wy	
unorm_4::set_wyx	
unorm_4::set_wyz	
unorm_4::set_wyz	
unorm_4::set_wyzx	
unorm_4::set_wz	
unorm_4::set_wzx	
unorm_4::set_wzxy	

NAME	DESCRIPTION
unorm_4::set_wzy	
unorm_4::set_wzyx	
unorm_4::set_x	
unorm_4::set_xw	
unorm_4::set_xwy	
unorm_4::set_xwyz	
unorm_4::set_xwz	
unorm_4::set_xwzy	
unorm_4::set_xy	
unorm_4::set_xyw	
unorm_4::set_xywz	
unorm_4::set_xyz	
unorm_4::set_xyzw	
unorm_4::set_xz	
unorm_4::set_xzw	
unorm_4::set_xzwy	
unorm_4::set_xzy	
unorm_4::set_xzyw	
unorm_4::set_y	
unorm_4::set_yw	
unorm_4::set_ywx	
unorm_4::set_ywzx	
unorm_4::set_ywz	
unorm_4::set_ywzx	
unorm_4::set_yx	

NAME	DESCRIPTION
unorm_4::set_yxw	
unorm_4::set_yxwz	
unorm_4::set_yxz	
unorm_4::set_yxzw	
unorm_4::set_yz	
unorm_4::set_yzw	
unorm_4::set_yzwx	
unorm_4::set_yzx	
unorm_4::set_yzxw	
unorm_4::set_z	
unorm_4::set_zw	
unorm_4::set_zwx	
unorm_4::set_zwxy	
unorm_4::set_zwy	
unorm_4::set_zwyx	
unorm_4::set_zx	
unorm_4::set_zxw	
unorm_4::set_zxwy	
unorm_4::set_zxy	
unorm_4::set_zxyw	
unorm_4::set_zy	
unorm_4::set_zyw	
unorm_4::set_zywx	
unorm_4::set_zyx	
unorm_4::set_zyxw	

**Public Operators**

NAME	DESCRIPTION
unorm_4::operator-	
unorm_4::operator--	
unorm_4::operator* =	
unorm_4::operator/=	
unorm_4::operator+ +	
unorm_4::operator+ =	
unorm_4::operator=	
unorm_4::operator- =	

### Public Constants

NAME	DESCRIPTION
<a href="#">size Constant</a>	

### Public Data Members

NAME	DESCRIPTION
unorm_4::a	
unorm_4::ab	
unorm_4::abg	
unorm_4::abgr	
unorm_4::abr	
unorm_4::abrg	
unorm_4::ag	
unorm_4::agb	
unorm_4::agbr	
unorm_4::agr	
unorm_4::agrb	
unorm_4::ar	
unorm_4::arb	

NAME	DESCRIPTION
unorm_4::arbg	
unorm_4::arg	
unorm_4::argb	
unorm_4::b	
unorm_4::ba	
unorm_4::bag	
unorm_4::bagr	
unorm_4::bar	
unorm_4::barg	
unorm_4::bg	
unorm_4::bga	
unorm_4::bgar	
unorm_4::bgr	
unorm_4::bgra	
unorm_4::br	
unorm_4::bra	
unorm_4::brag	
unorm_4::brg	
unorm_4::brga	
unorm_4::g	
unorm_4::ga	
unorm_4::gab	
unorm_4::gabr	
unorm_4::gar	
unorm_4::garb	

NAME	DESCRIPTION
unorm_4::gb	
unorm_4::gba	
unorm_4::gbar	
unorm_4::gbr	
unorm_4::gbra	
unorm_4::gr	
unorm_4::gra	
unorm_4::grab	
unorm_4::grb	
unorm_4::grba	
unorm_4::r	
unorm_4::ra	
unorm_4::rab	
unorm_4::rabg	
unorm_4::rag	
unorm_4::ragb	
unorm_4::rb	
unorm_4::rba	
unorm_4::rbag	
unorm_4::rbg	
unorm_4::rbga	
unorm_4::rg	
unorm_4::rga	
unorm_4::rgab	
unorm_4::rgb	

NAME	DESCRIPTION
unorm_4::rgba	
unorm_4::w	
unorm_4::wx	
unorm_4::wxy	
unorm_4::wxyz	
unorm_4::wxz	
unorm_4::wxzy	
unorm_4::wy	
unorm_4::wyx	
unorm_4::wyz	
unorm_4::wyzx	
unorm_4::wz	
unorm_4::wzx	
unorm_4::wzxy	
unorm_4::wzy	
unorm_4::wzyx	
unorm_4::x	
unorm_4::xw	
unorm_4::xwy	
unorm_4::xwyz	
unorm_4::xwz	
unorm_4::xwzy	
unorm_4::xy	
unorm_4::xyw	

NAME	DESCRIPTION
unorm_4::xywz	
unorm_4::xyz	
unorm_4::xyzw	
unorm_4::xz	
unorm_4::xzw	
unorm_4::xzwy	
unorm_4::xzy	
unorm_4::xzyw	
unorm_4::y	
unorm_4::yw	
unorm_4::ywx	
unorm_4::ywxz	
unorm_4::y wz	
unorm_4::ywzx	
unorm_4::yx	
unorm_4::yxw	
unorm_4::yxwz	
unorm_4::yxz	
unorm_4::yxzw	
unorm_4::yz	
unorm_4::yzw	
unorm_4::yzwx	
unorm_4::yzx	
unorm_4::yzxw	
unorm_4::z	

NAME	DESCRIPTION
unorm_4::zw	
unorm_4::zwx	
unorm_4::zwxy	
unorm_4::zwy	
unorm_4::zwyx	
unorm_4::zx	
unorm_4::zxw	
unorm_4::zxwy	
unorm_4::zxy	
unorm_4::zxyw	
unorm_4::zy	
unorm_4::zyw	
unorm_4::zywx	
unorm_4::zyx	
unorm_4::zyxw	

## Inheritance Hierarchy

`unorm_4`

## Requirements

**Header:** amp\_short\_vectors.h

**Namespace:** Concurrency::graphics

## unorm\_4

Default constructor, initializes all elements with 0.

```

unorm_4() restrict(amp,
    cpu);

unorm_4(
    unorm _V0,
    unorm _V1,
    unorm _V2,
    unorm _V3) restrict(amp,
    cpu);

unorm_4(
    float _V0,
    float _V1,
    float _V2,
    float _V3) restrict(amp,
    cpu);

unorm_4(
    unorm _V) restrict(amp,
    cpu);

explicit unorm_4(
    float _V) restrict(amp,
    cpu);

unorm_4(
    const unorm_4& _Other) restrict(amp,
    cpu);

explicit inline unorm_4(
    const uint_4& _Other) restrict(amp,
    cpu);

explicit inline unorm_4(
    const int_4& _Other) restrict(amp,
    cpu);

explicit inline unorm_4(
    const float_4& _Other) restrict(amp,
    cpu);

explicit inline unorm_4(
    const norm_4& _Other) restrict(amp,
    cpu);

explicit inline unorm_4(
    const double_4& _Other) restrict(amp,
    cpu);

```

## Parameters

**\_V0**

The value to initialize element 0.

**\_V1**

The value to initialize element 1.

**\_V2**

The value to initialize element 2.

**\_V3**

The value to initialize element 3.

**\_V**

The value for initialization.

*\_Other*

The object used to initialize.

## size

```
static const int size = 4;
```

## See also

[Concurrency::graphics Namespace](#)

# Concurrency::precise\_math Namespace

3/4/2019 • 7 minutes to read • [Edit Online](#)

Functions in the `precise_math` namespace are C99 compliant. Both single precision and double precision versions of each function are included. For example, `acos` is the double-precision version and `acosf` is the single-precision version. These functions, including the single-precision functions, require extended double-precision support on the accelerator. You can use the [accelerator::supports\\_double\\_precision](#) to determine if you can run these functions on a specific accelerator.

## Syntax

```
namespace precise_math;
```

### Parameters

## Members

### Functions

NAME	DESCRIPTION
<a href="#">acos</a>	Overloaded. Calculates the arccosine of the argument
<a href="#">acosf</a>	Calculates the arccosine of the argument
<a href="#">acosh</a>	Overloaded. Calculates the inverse hyperbolic cosine of the argument
<a href="#">acoshf</a>	Calculates the inverse hyperbolic cosine of the argument
<a href="#">asin</a>	Overloaded. Calculates the arcsine of the argument
<a href="#">asinf</a>	Calculates the arcsine of the argument
<a href="#">asinh</a>	Overloaded. Calculates the inverse hyperbolic sine of the argument
<a href="#">asinhf</a>	Calculates the inverse hyperbolic sine of the argument
<a href="#">atan</a>	Overloaded. Calculates the arctangent of the argument
<a href="#">atan2</a>	Overloaded. Calculates the arctangent of $_Y/_X$
<a href="#">atan2f</a>	Calculates the arctangent of $_Y/_X$
<a href="#">atanf</a>	Calculates the arctangent of the argument
<a href="#">atanh</a>	Overloaded. Calculates the inverse hyperbolic tangent of the argument

NAME	DESCRIPTION
atanhf	Calculates the inverse hyperbolic tangent of the argument
cbrt	Overloaded. Computes the real cube root of the argument
cbrtf	Computes the real cube root of the argument
ceil	Overloaded. Calculates the ceiling of the argument
ceilf	Calculates the ceiling of the argument
copysign	Overloaded. Produces a value with the magnitude of <code>_X</code> and the sign of <code>_Y</code>
copysignf	Produces a value with the magnitude of <code>_X</code> and the sign of <code>_Y</code>
cos	Overloaded. Calculates the cosine of the argument
cosf	Calculates the cosine of the argument
cosh	Overloaded. Calculates the hyperbolic cosine value of the argument
coshf	Calculates the hyperbolic cosine value of the argument
cospi	Overloaded. Calculates the cosine value of $\pi * \_X$
cospif	Calculates the cosine value of $\pi * \_X$
erf	Overloaded. Computes the error function of <code>_X</code>
erfc	Overloaded. Computes the complementary error function of <code>_X</code>
erfcf	Computes the complementary error function of <code>_X</code>
erfcinv	Overloaded. Computes the inverse complementary error function of <code>_X</code>
erfcinvf	Computes the inverse complementary error function of <code>_X</code>
erff	Computes the error function of <code>_X</code>
erfinv	Overloaded. Computes the inverse error function of <code>_X</code>
erfinvf	Computes the inverse error function of <code>_X</code>
exp	Overloaded. Calculates the base-e exponential of the argument
exp10	Overloaded. Calculates the base-10 exponential of the argument

NAME	DESCRIPTION
<a href="#">exp10f</a>	Calculates the base-10 exponential of the argument
<a href="#">exp2</a>	Overloaded. Calculates the base-2 exponential of the argument
<a href="#">exp2f</a>	Calculates the base-2 exponential of the argument
<a href="#">expf</a>	Calculates the base-e exponential of the argument
<a href="#">expm1</a>	Overloaded. Calculates the base-e exponential of the argument, minus 1
<a href="#">expm1f</a>	Calculates the base-e exponential of the argument, minus 1
<a href="#">fabs</a>	Overloaded. Returns the absolute value of the argument
<a href="#">fabsf</a>	Returns the absolute value of the argument
<a href="#">fdim</a>	Overloaded. Determines the positive difference between the arguments
<a href="#">fdimf</a>	Determines the positive difference between the arguments
<a href="#">floor</a>	Overloaded. Calculates the floor of the argument
<a href="#">floorf</a>	Calculates the floor of the argument
<a href="#">fma</a>	Overloaded. Compute $(\_X * \_Y) + \_Z$ , rounded as one ternary operation
<a href="#">fmaf</a>	Compute $(\_X * \_Y) + \_Z$ , rounded as one ternary operation
<a href="#">fmax</a>	Overloaded. Determine the maximum numeric value of the arguments
<a href="#">fmaxf</a>	Determine the maximum numeric value of the arguments
<a href="#">fmin</a>	Overloaded. Determine the minimum numeric value of the arguments
<a href="#">fminf</a>	Determine the minimum numeric value of the arguments
<a href="#">fmod Function (C++ AMP)</a>	Overloaded. Calculates the floating-point remainder of $\_X/\_Y$
<a href="#">fmodf</a>	Calculates the floating-point remainder of $\_X/\_Y$
<a href="#">fpclassify</a>	Overloaded. Classifies the argument value as NaN, infinite, normal, subnormal, zero
<a href="#">frexp</a>	Overloaded. Gets the mantissa and exponent of $\_X$

NAME	DESCRIPTION
<a href="#">frexp</a>	Gets the mantissa and exponent of <code>_X</code>
<a href="#">hypot</a>	Overloaded. Computes the square root of the sum of the squares of <code>_X</code> and <code>_Y</code>
<a href="#">hypotf</a>	Computes the square root of the sum of the squares of <code>_X</code> and <code>_Y</code>
<a href="#">ilogb</a>	Overloaded. Extract the exponent of <code>_X</code> as a signed int value
<a href="#">ilogbf</a>	Extract the exponent of <code>_X</code> as a signed int value
<a href="#">isfinite</a>	Overloaded. Determines whether the argument has a finite value
<a href="#">isinf</a>	Overloaded. Determines whether the argument is an infinity
<a href="#">isnan</a>	Overloaded. Determines whether the argument is a NaN
<a href="#">isnormal</a>	Overloaded. Determines whether the argument is a normal
<a href="#">ldexp</a>	Overloaded. Computes a real number from the mantissa and exponent
<a href="#">ldexpf</a>	Computes a real number from the mantissa and exponent
<a href="#">lgamma</a>	Overloaded. Computes the natural logarithm of the absolute value of gamma of the argument
<a href="#">lgammaf</a>	Computes the natural logarithm of the absolute value of gamma of the argument
<a href="#">log</a>	Overloaded. Calculates the base-e logarithm of the argument
<a href="#">log10</a>	Overloaded. Calculates the base-10 logarithm of the argument
<a href="#">log10f</a>	Calculates the base-10 logarithm of the argument
<a href="#">log1p</a>	Overloaded. Calculates the base-e logarithm of 1 plus the argument
<a href="#">log1pf</a>	Calculates the base-e logarithm of 1 plus the argument
<a href="#">log2</a>	Overloaded. Calculates the base-2 logarithm of the argument
<a href="#">log2f</a>	Calculates the base-2 logarithm of the argument
<a href="#">logb</a>	Overloaded. Extracts the exponent of <code>_X</code> , as a signed integer value in floating-point format

NAME	DESCRIPTION
logbf	Extracts the exponent of <code>_X</code> , as a signed integer value in floating-point format
logf	Calculates the base-e logarithm of the argument
modf	Overloaded. Splits <code>_X</code> into fractional and integer parts.
modff	Splits <code>_X</code> into fractional and integer parts.
nan	Returns a quiet NaN
nanf	Returns a quiet NaN
nearbyint	Overloaded. Rounds the argument to an integer value in floating-point format, using the current rounding direction.
nearbyintf	Rounds the argument to an integer value in floating-point format, using the current rounding direction.
nextafter	Overloaded. Determine the next representable value, in the type of the function, after <code>_X</code> in the direction of <code>_Y</code>
nextafterf	Determine the next representable value, in the type of the function, after <code>_X</code> in the direction of <code>_Y</code>
phi	Overloaded. Returns the cumulative distribution function of the argument
phif	Returns the cumulative distribution function of the argument
pow	Overloaded. Calculates <code>_X</code> raised to the power of <code>_Y</code>
powf	Calculates <code>_X</code> raised to the power of <code>_Y</code>
probit	Overloaded. Returns the inverse cumulative distribution function of the argument
probitf	Returns the inverse cumulative distribution function of the argument
rcbrt	Overloaded. Returns the reciprocal of the cube root of the argument
rcbrtf	Returns the reciprocal of the cube root of the argument
remainder	Overloaded. Computes the remainder: <code>_X REM _Y</code>
remainderf	Computes the remainder: <code>_X REM _Y</code>

NAME	DESCRIPTION
remquo	Overloaded. Computes the same remainder as <code>_X REM _Y</code> . Also calculates the lower 23 bits of the integral quotient <code>_X/_Y</code> , and gives that value the same sign as <code>_X/_Y</code> . It stores this signed value in the integer pointed to by <code>_Quo</code> .
remquof	Computes the same remainder as <code>_X REM _Y</code> . Also calculates the lower 23 bits of the integral quotient <code>_X/_Y</code> , and gives that value the same sign as <code>_X/_Y</code> . It stores this signed value in the integer pointed to by <code>_Quo</code> .
round	Overloaded. Rounds <code>_X</code> to the nearest integer
roundf	Rounds <code>_X</code> to the nearest integer
rsqrt	Overloaded. Returns the reciprocal of the square root of the argument
rsqrtf	Returns the reciprocal of the square root of the argument
scalb	Overloaded. Multiplies <code>_X</code> by <code>FLT_RADIX</code> to the power <code>_Y</code>
scalbf	Multiplies <code>_X</code> by <code>FLT_RADIX</code> to the power <code>_Y</code>
scalbn	Overloaded. Multiplies <code>_X</code> by <code>FLT_RADIX</code> to the power <code>_Y</code>
scalbnf	Multiplies <code>_X</code> by <code>FLT_RADIX</code> to the power <code>_Y</code>
signbit	Overloaded. Determines whether the sign of <code>_X</code> is negative
signbitf	Determines whether the sign of <code>_X</code> is negative
sin	Overloaded. Calculates the sine value of the argument
sincos	Overloaded. Calculates sine and cosine value of <code>_X</code>
sincosf	Calculates sine and cosine value of <code>_X</code>
sinf	Calculates the sine value of the argument
sinh	Overloaded. Calculates the hyperbolic sine value of the argument
sinhf	Calculates the hyperbolic sine value of the argument
sinpi	Overloaded. Calculates the sine value of <code>pi * _X</code>
sinpif	Calculates the sine value of <code>pi * _X</code>
sqrt	Overloaded. Calculates the square root of the argument
sqrtf	Calculates the square root of the argument

NAME	DESCRIPTION
<a href="#">tan</a>	Overloaded. Calculates the tangent value of the argument
<a href="#">tanf</a>	Calculates the tangent value of the argument
<a href="#">tanh</a>	Overloaded. Calculates the hyperbolic tangent value of the argument
<a href="#">tanhf</a>	Calculates the hyperbolic tangent value of the argument
<a href="#">tanpi</a>	Overloaded. Calculates the tangent value of $\pi * \_X$
<a href="#">tanpif</a>	Calculates the tangent value of $\pi * \_X$
<a href="#">tgamma</a>	Overloaded. Computes the gamma function of $\_X$
<a href="#">tgammaf</a>	Computes the gamma function of $\_X$
<a href="#">trunc</a>	Overloaded. Truncates the argument to the integer component
<a href="#">truncf</a>	Truncates the argument to the integer component

## Requirements

**Header:** `amp_math.h`

**Namespace:** `Concurrency`

## See also

[Concurrency Namespace \(C++ AMP\)](#)

# Concurrency::precise\_math namespace functions

3/4/2019 • 24 minutes to read • [Edit Online](#)

acos	acosf	acosh
acoshf	asin	asinf
asinh	asinhf	atan
atan2	atan2f	atanf
atanh	atanhf	cbrt
cbrtf	ceil	ceilf
copysign	copysignf	cos
cosf	cosh	coshf
cospi	cospif	erf
erfc	erfcf	erfcinv
erfcinvf	erff	erfinv
erfinvf	exp	exp10
exp10f	exp2	exp2f
expf	expm1	expm1f
fabs	fabsf	floor
fdim	fdimf	
floorf	fma	fmaf
fmax	fmaxf	
fmin	fminf	fmod
fmodf	fpclassify	frexp
frexpf	hypot	hypotf
ilogb	ilogbf	isfinite

isinf	isnan	isnormal
ldexp	ldexpf	lgamma
lgammaf	log	log10
log10f	log1p	log1pf
log2	log2f	logb
logbf	logf	modf
modff	nan	nanf
nearbyint	nearbyintf	nextafter
nextafterf	phi	phif
pow	powf	probit
probitf	rcbrt	rcbrtf
remainder	remainderf	remquo
remquof	round	roundf
rsqrt	rsqrtf	scalb
scalbf	scalbn	scalbnf
signbit	signbitf	sin
sincos	sincosf	sinf
sinh	sinhf	sinpi
sinpif	sqrt	sqrtf
tan	tanf	tanh
tanhf	tanpi	tanpif
tgamma	tgammaf	trunc
truncf		

## acos

Calculates the arccosine of the argument

```
inline float acos(float _X) restrict(amp);  
  
inline double acos(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the arccosine value of the argument

## acosf

Calculates the arccosine of the argument

```
inline float acosf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the arccosine value of the argument

## acosh

Calculates the inverse hyperbolic cosine of the argument

```
inline float acosh(float _X) restrict(amp);  
  
inline double acosh(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the inverse hyperbolic cosine value of the argument

## acoshf

Calculates the inverse hyperbolic cosine of the argument

```
inline float acoshf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the inverse hyperbolic cosine value of the argument

# asin

Calculates the arcsine of the argument

```
inline float asin(float _X) restrict(amp);  
  
inline double asin(double _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the arcsine value of the argument

# asinf

Calculates the arcsine of the argument

```
inline float asinf(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the arcsine value of the argument

# asinh

Calculates the inverse hyperbolic sine of the argument

```
inline float asinh(float _X) restrict(amp);  
  
inline double asinh(double _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the inverse hyperbolic sine value of the argument

# asinhf

Calculates the inverse hyperbolic sine of the argument

```
inline float asinhf(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

### Return Value

Returns the inverse hyperbolic sine value of the argument

## atan

Calculates the arctangent of the argument

```
inline float atan(float _X) restrict(amp);  
  
inline double atan(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the arctangent value of the argument

## atan2

Calculates the arctangent of  $Y/X$

```
inline float atan2(  
    float _Y,  
    float _X) restrict(amp);  
  
inline double atan2(  
    double _Y,  
    double _X) restrict(amp);
```

### Parameters

`_Y`

Floating-point value

`_X`

Floating-point value

### Return Value

Returns the arctangent value of  $Y/X$

## atan2f

Calculates the arctangent of  $Y/X$

```
inline float atan2f(  
    float _Y,  
    float _X) restrict(amp);
```

### Parameters

`_Y`

Floating-point value

`_X`

Floating-point value

**Return Value**

Returns the arctangent value of  $_Y/_X$

## atanf

Calculates the arctangent of the argument

```
inline float atanf(float _X) restrict(amp);
```

**Parameters**

$_X$

Floating-point value

**Return Value**

Returns the arctangent value of the argument

## atanh

Calculates the inverse hyperbolic tangent of the argument

```
inline float atanh(float _X) restrict(amp);  
  
inline double atanh(double _X) restrict(amp);
```

**Parameters**

$_X$

Floating-point value

**Return Value**

Returns the inverse hyperbolic tangent value of the argument

## atanhf

Calculates the inverse hyperbolic tangent of the argument

```
inline float atanhf(float _X) restrict(amp);
```

**Parameters**

$_X$

Floating-point value

**Return Value**

Returns the inverse hyperbolic tangent value of the argument

## cbrt

Computes the real cube root of the argument

```
inline float cbrt(float _X) restrict(amp);  
  
inline double cbrt(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the real cube root of the argument

## cbrtf

Computes the real cube root of the argument

```
inline float cbrtf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the real cube root of the argument

## ceil

Calculates the ceiling of the argument

```
inline float ceil(float _X) restrict(amp);  
  
inline double ceil(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the ceiling of the argument

## ceilf

Calculates the ceiling of the argument

```
inline float ceilf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the ceiling of the argument

## copysign

Produces a value with the magnitude of `_X` and the sign of `_Y`

```
inline float copysign(  
    float _X,  
    float _Y) restrict(amp);  
  
inline double copysign(  
    double _X,  
    double _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Returns a value with the magnitude of `_X` and the sign of `_Y`

## copysignf

Produces a value with the magnitude of `_X` and the sign of `_Y`

```
inline float copysignf(  
    float _X,  
    float _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Returns a value with the magnitude of `_X` and the sign of `_Y`

## COS

Calculates the cosine of the argument

```
inline float cos(float _X) restrict(amp);  
  
inline double cos(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the cosine value of the argument

## cosf

Calculates the cosine of the argument

```
inline float cosf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the cosine value of the argument

## cosh

Calculates the hyperbolic cosine value of the argument

```
inline float cosh(float _X) restrict(amp);  
  
inline double cosh(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the hyperbolic cosine value of the argument

## coshf

Calculates the hyperbolic cosine value of the argument

```
inline float coshf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the hyperbolic cosine value of the argument

## cospi

Calculates the cosine value of  $\pi * \_X$

```
inline float cospi(float _X) restrict(amp);  
  
inline double cospi(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the cosine value of  $\pi * \_X$

# cospif

Calculates the cosine value of  $\pi * \_X$

```
inline float cospif(float _X) restrict(amp);
```

## Parameters

$\_X$

Floating-point value

## Return Value

Returns the cosine value of  $\pi * \_X$

# erf

Computes the error function of  $\_X$

```
inline float erf(float _X) restrict(amp);  
  
inline double erf(double _X) restrict(amp);
```

## Parameters

$\_X$

Floating-point value

## Return Value

Returns the error function of  $\_X$

# erfc

Computes the complementary error function of  $\_X$

```
inline float erfc(float _X) restrict(amp);  
  
inline double erfc(double _X) restrict(amp);
```

## Parameters

$\_X$

Floating-point value

## Return Value

Returns the complementary error function of  $\_X$

# erfcf

Computes the complementary error function of  $\_X$

```
inline float erfcf(float _X) restrict(amp);
```

## Parameters

$\_X$

Floating-point value

**Return Value**

Returns the complementary error function of  $_X$

## erfcinv

Computes the inverse complementary error function of  $_X$

```
inline float erfcinv(float _X) restrict(amp);  
  
inline double erfcinv(double _X) restrict(amp);
```

**Parameters**

$_X$

Floating-point value

**Return Value**

Returns the inverse complementary error function of  $_X$

## erfcinvf

Computes the inverse complementary error function of  $_X$

```
inline float erfcinvf(float _X) restrict(amp);
```

**Parameters**

$_X$

Floating-point value

**Return Value**

Returns the inverse complementary error function of  $_X$

## erff

Computes the error function of  $_X$

```
inline float erff(float _X) restrict(amp);
```

**Parameters**

$_X$

Floating-point value

**Return Value**

Returns the error function of  $_X$

## erfinv

Computes the inverse error function of  $_X$

```
inline float erfinv(float _X) restrict(amp);  
  
inline double erfinv(double _X) restrict(amp);
```

**Parameters**

`_X`

Floating-point value

**Return Value**

Returns the inverse error function of `_X`

## erfinvf

Computes the inverse error function of `_X`

```
inline float erfinvf(float _X) restrict(amp);
```

**Parameters**

`_X`

Floating-point value

**Return Value**

Returns the inverse error function of `_X`

## exp10

Calculates the base-10 exponential of the argument

```
inline float exp10(float _X) restrict(amp);  
  
inline double exp10(double _X) restrict(amp);
```

**Parameters**

`_X`

Floating-point value

**Return Value**

Returns the base-10 exponential of the argument

## exp10f

Calculates the base-10 exponential of the argument

```
inline float exp10f(float _X) restrict(amp);
```

**Parameters**

`_X`

Floating-point value

**Return Value**

Returns the base-10 exponential of the argument

## expm1

Calculates the base-e exponential of the argument, minus 1

```
inline float expm1(float exponent) restrict(amp);  
  
inline double expm1(double exponent) restrict(amp);
```

### Parameters

*exponent*

The exponential term  $n$  of the mathematical expression  $e^n$ , where  $e$  is the base of the natural logarithm.

### Return Value

Returns the base-e exponential of the argument, minus 1

## expm1f

Calculates the base-e exponential of the argument, minus 1

```
inline float expm1f(float exponent) restrict(amp);
```

### Parameters

*exponent*

The exponential term  $n$  of the mathematical expression  $e^n$ , where  $e$  is the base of the natural logarithm.

### Return Value

Returns the base-e exponential of the argument, minus 1

## exp

Calculates the base-e exponential of the argument

```
inline float exp(float _X) restrict(amp);  
  
inline double exp(double _X) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

### Return Value

Returns the base-e exponential of the argument

## expf

Calculates the base-e exponential of the argument

```
inline float expf(float _X) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

### Return Value

Returns the base-e exponential of the argument

## exp2

Calculates the base-2 exponential of the argument

```
inline float exp2(float _X) restrict(amp);  
  
inline double exp2(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the base-2 exponential of the argument

## exp2f

Calculates the base-2 exponential of the argument

```
inline float exp2f(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the base-2 exponential of the argument

## fabs

Returns the absolute value of the argument

```
inline float fabs(float _X) restrict(amp);  
  
inline double fabs(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the absolute value of the argument

## fabsf

Returns the absolute value of the argument

```
inline float fabsf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

## Return Value

Returns the absolute value of the argument

# fdim

Computes the positive difference between the arguments.

```
inline float fdim(  
    float _X,  
    float _Y  
) restrict(amp);  
inline double fdim(  
    double _X,  
    double _Y  
) restrict(amp);
```

## Parameters

`_X`

Floating-point value `_Y`

Floating-point value

## Return Value

The difference between `_X` and `_Y` if `_X` is greater than `_Y`; otherwise, `+0`.

# fdimf

Computes the positive difference between the arguments.

```
inline float fdimf(  
    float _X,  
    float _Y  
) restrict(amp);
```

## Parameters

`_X`

Floating-point value `_Y`

Floating-point value

## Return Value

The difference between `_X` and `_Y` if `_X` is greater than `_Y`; otherwise, `+0`.

# floor

Calculates the floor of the argument

```
inline float floor(float _X) restrict(amp);  
  
inline double floor(double _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the floor of the argument

## floorf

Calculates the floor of the argument

```
inline float floorf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the floor of the argument

## fma

Computes the product of the first and second specified arguments, then adds the third specified argument to the result; the entire computation is performed as a single operation.

```
inline float fma(  
    float _X,  
    float _Y,  
    float _Z  
) restrict(amp);  
  
inline double fma(  
    double _X,  
    double _Y,  
    double _Z  
) restrict(amp);
```

### Parameters

`_X`

The first floating-point argument. `_Y`

The second floating-point argument. `_Z`

The third floating-point argument.

### Return Value

The result of the expression  $(\_X * \_Y) + \_Z$ . The entire computation is performed as a single operation; that is, the sub-expressions are calculated to infinite precision, and only the final result is rounded.

## fmaf

Computes the product of the first and second specified arguments, then adds the third specified argument to the result; the entire computation is performed as a single operation.

```
inline float fmaf(  
    float _X,  
    float _Y,  
    float _Z  
) restrict(amp);
```

### Parameters

`_X`

The first floating-point argument. `_Y`

The second floating-point argument. `_Z`

The third floating-point argument.

### Return Value

The result of the expression  $(\_X * \_Y) + \_Z$ . The entire computation is performed as a single operation; that is, the sub-expressions are calculated to infinite precision, and only the final result is rounded.

## fmax

Determine the maximum numeric value of the arguments

```
inline float fmax(  
    float _X,  
    float _Y) restrict(amp);  
  
inline double fmax(  
    double _X,  
    double _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Return the maximum numeric value of the arguments

## fmaxf

Determine the maximum numeric value of the arguments

```
inline float fmaxf(  
    float _X,  
    float _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Return the maximum numeric value of the arguments

## fmin

Determine the minimum numeric value of the arguments

```
inline float fmin(  
    float _X,  
    float _Y) restrict(amp);  
  
inline double fmin(  
    double _X,  
    double _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Return the minimum numeric value of the arguments

## fminf

Determine the minimum numeric value of the arguments

```
inline float fminf(  
    float _X,  
    float _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Return the minimum numeric value of the arguments

## fmod Function (C++ AMP)

Computes the remainder of the first specified argument divided by the second specified argument.

```
inline float fmod(  
    float _X,  
    float _Y) restrict(amp);  
  
inline double fmod(  
    double _X,  
    double _Y) restrict(amp);
```

### Parameters

`_X`

The first floating-point argument.

`_Y`

The second floating-point argument.

## Return Value

The remainder of  $\_X$  divided by  $\_Y$ ; that is, the value of  $\_X - \_Y n$ , where  $n$  is a whole integer such that the magnitude of  $\_X - \_Y n$  is less than the magnitude of  $\_Y$ .

# fmodf

Computes the remainder of the first specified argument divided by the second specified argument.

```
inline float fmodf(  
    float _X,  
    float _Y) restrict(amp);
```

## Parameters

$\_X$

The first floating-point argument.

$\_Y$

The second floating-point argument.

## Return Value

The remainder of  $\_X$  divided by  $\_Y$ ; that is, the value of  $\_X - \_Y n$ , where  $n$  is a whole integer such that the magnitude of  $\_X - \_Y n$  is less than the magnitude of  $\_Y$ .

# fpclassify

Classifies the argument value as NaN, infinite, normal, subnormal, zero

```
inline int fpclassify(float _X) restrict(amp);  
  
inline int fpclassify(double _X) restrict(amp);
```

## Parameters

$\_X$

Floating-point value

## Return Value

Returns the value of the number classification macro appropriate to the value of the argument.

# frexp

Gets the mantissa and exponent of  $\_X$

```
inline float frexp(  
    float _X,  
    _Out_ int* _Exp) restrict(amp);  
  
inline double frexp(  
    double _X,  
    _Out_ int* _Exp) restrict(amp);
```

## Parameters

$\_X$

Floating-point value

*\_Exp*

Returns the integer exponent of *\_X* in floating-point value

### **Return Value**

Returns the mantissa *\_X*

## frexpf

Gets the mantissa and exponent of *\_X*

```
inline float frexpf(  
    float _X,  
    _Out_ int* _Exp) restrict(amp);
```

### **Parameters**

*\_X*

Floating-point value

*\_Exp*

Returns the integer exponent of *\_X* in floating-point value

### **Return Value**

Returns the mantissa *\_X*

## hypot

Computes the square root of the sum of the squares of *\_X* and *\_Y*

```
inline float hypot(  
    float _X,  
    float _Y) restrict(amp);  
  
inline double hypot(  
    double _X,  
    double _Y) restrict(amp);
```

### **Parameters**

*\_X*

Floating-point value

*\_Y*

Floating-point value

### **Return Value**

Returns the square root of the sum of the squares of *\_X* and *\_Y*

## hypotf

Computes the square root of the sum of the squares of *\_X* and *\_Y*

```
inline float hypotf(  
    float _X,  
    float _Y) restrict(amp);
```

### **Parameters**

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Returns the square root of the sum of the squares of `_X` and `_Y`

## ilogb

Extract the exponent of `_X` as a signed int value

```
inline int ilogb(float _X) restrict(amp);  
  
inline int ilogb(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the exponent of `_X` as a signed int value

## ilogbf

Extract the exponent of `_X` as a signed int value

```
inline int ilogbf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the exponent of `_X` as a signed int value

## isfinite

Determines whether the argument has a finite value

```
inline int isfinite(float _X) restrict(amp);  
  
inline int isfinite(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns a nonzero value if and only if the argument has a finite value

# isinf

Determines whether the argument is an infinity

```
inline int isinf(float _X) restrict(amp);  
  
inline int isinf(double _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns a nonzero value if and only if the argument has an infinite value

# isnan

Determines whether the argument is a NaN

```
inline int isnan(float _X) restrict(amp);  
  
inline int isnan(double _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns a nonzero value if and only if the argument has a NaN value

# isnormal

Determines whether the argument is a normal

```
inline int isnormal(float _X) restrict(amp);  
  
inline int isnormal(double _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns a nonzero value if and only if the argument has a normal value

# ldexp

Computes a real number from the specified mantissa and exponent.

```
inline float ldexp(  
    float _X,  
    int _Exp) restrict(amp);  
  
inline double ldexp(  
    double _X,  
    double _Exp) restrict(amp);
```

### Parameters

*\_X*

Floating-point value, mantissa

*\_Exp*

Integer value, exponent

### Return Value

Returns  $_X * 2^{\_Exp}$

## ldexpf

Computes a real number from the specified mantissa and exponent.

```
inline float ldexpf(  
    float _X,  
    int _Exp) restrict(amp);
```

### Parameters

*\_X*

Floating-point value, mantissa

*\_Exp*

Integer value, exponent

### Return Value

Returns  $_X * 2^{\_Exp}$

## lgamma

Computes the natural logarithm of the absolute value of gamma of the argument

```
inline float lgamma(  
    float _X,  
    _Out_ int* _Sign) restrict(amp);  
  
inline double lgamma(  
    double _X,  
    _Out_ int* _Sign) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

*\_Sign*

Returns the sign

### Return Value

Returns the natural logarithm of the absolute value of gamma of the argument

## lgammaf

Computes the natural logarithm of the absolute value of gamma of the argument

```
inline float lgammaf(  
    float _X,  
    _Out_ int* _Sign) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

*\_Sign*

Returns the sign

### Return Value

Returns the natural logarithm of the absolute value of gamma of the argument

## log

Calculates the base-e logarithm of the argument

```
inline float log(float _X) restrict(amp);  
  
inline double log(double _X) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

### Return Value

Returns the base-e logarithm of the argument

## log10

Calculates the base-10 logarithm of the argument

```
inline float log10(float _X) restrict(amp);  
  
inline double log10(double _X) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

### Return Value

Returns the base-10 logarithm of the argument

## log10f

Calculates the base-10 logarithm of the argument

```
inline float log10f(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the base-10 logarithm of the argument

## log1p

Calculates the base-e logarithm of 1 plus the argument

```
inline float log1p(float _X) restrict(amp);  
  
inline double log1p(double _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the base-e logarithm of 1 plus the argument

## log1pf

Calculates the base-e logarithm of 1 plus the argument

```
inline float log1pf(float _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the base-e logarithm of 1 plus the argument

## log2

Calculates the base-2 logarithm of the argument

```
inline float log2(float _X) restrict(amp);  
  
inline double log2(double _X) restrict(amp);
```

#### Parameters

`_X`

Floating-point value

#### Return Value

Returns the base-10 logarithm of the argument

## log2f

Calculates the base-2 logarithm of the argument

```
inline float log2f(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the base-10 logarithm of the argument

## logb

Extracts the exponent of `_X`, as a signed integer value in floating-point format

```
inline float logb(float _X) restrict(amp);  
  
inline double logb(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the signed exponent of `_X`

## logbf

Extracts the exponent of `_X`, as a signed integer value in floating-point format

```
inline float logbf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the signed exponent of `_X`

## logf

Calculates the base-e logarithm of the argument

```
inline float logf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the base-e logarithm of the argument

## modf

Splits the specified argument into fractional and integer parts.

```
inline float modf(  
    float _X,  
    _Out_ float* _Iptr) restrict(amp);  
  
inline double modf(  
    double _X,  
    _Out_ double* _Iptr) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

*\_Iptr*

[out] The integer portion of `_X`, as a floating-point value.

### Return Value

The signed fractional portion of `_X`.

## modff

Splits the specified argument into fractional and integer parts.

```
inline float modff(  
    float _X,  
    _Out_ float* _Iptr) restrict(amp);
```

### Parameters

*\_X*

Floating-point value

*\_Iptr*

The integer portion of `_X`, as a floating-point value.

### Return Value

Returns the signed fractional portion of `_X`.

## nan

Returns a quiet NaN

```
inline double nan(int _X) restrict(amp);
```

### Parameters

*\_X*

Integer value

### Return Value

Returns a quiet NaN, if available, with the content indicated in `_X`

## nanf

Returns a quiet NaN

```
inline float nanf(int _X) restrict(amp);
```

### Parameters

`_X`

Integer value

### Return Value

Returns a quiet NaN, if available, with the content indicated in `_X`

## nearbyint

Rounds the argument to an integer value in floating-point format, using the current rounding direction.

```
inline float nearbyint(float _X) restrict(amp);  
  
inline double nearbyint(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the rounded integer value.

## nearbyintf

Rounds the argument to an integer value in floating-point format, using the current rounding direction.

```
inline float nearbyintf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the rounded integer value.

## nextafter

Determine the next representable value, in the type of the function, after `_X` in the direction of `_Y`

```
inline float nextafter(  
    float _X,  
    float _Y) restrict(amp);  
  
inline double nextafter(  
    double _X,  
    double _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Returns the next representable value, in the type of the function, after `_X` in the direction of `_Y`

## nextafterf

Determine the next representable value, in the type of the function, after `_X` in the direction of `_Y`

```
inline float nextafterf(  
    float _X,  
    float _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Returns the next representable value, in the type of the function, after `_X` in the direction of `_Y`

## phi

Returns the cumulative distribution function of the argument

```
inline float phi(float _X) restrict(amp);  
  
inline double phi(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the cumulative distribution function of the argument

## phif

Returns the cumulative distribution function of the argument

```
inline float phif(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the cumulative distribution function of the argument

## pow

Calculates `_X` raised to the power of `_Y`

```
inline float pow(  
    float _X,  
    float _Y) restrict(amp);  
  
inline double pow(  
    double _X,  
    double _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value, base

`_Y`

Floating-point value, exponent

### Return Value

## powf

Calculates `_X` raised to the power of `_Y`

```
inline float powf(  
    float _X,  
    float _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value, base

`_Y`

Floating-point value, exponent

### Return Value

## probit

Returns the inverse cumulative distribution function of the argument

```
inline float probit(float _X) restrict(amp);  
  
inline double probit(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the inverse cumulative distribution function of the argument

## probitf

Returns the inverse cumulative distribution function of the argument

```
inline float probitf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the inverse cumulative distribution function of the argument

## rcbrt

Returns the reciprocal of the cube root of the argument

```
inline float rcbrt(float _X) restrict(amp);  
  
inline double rcbrt(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the reciprocal of the cube root of the argument

## rcbrtf

Returns the reciprocal of the cube root of the argument

```
inline float rcbrtf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the reciprocal of the cube root of the argument

## remainder

Computes the remainder: `_X REM _Y`

```
inline float remainder(  
    float _X,  
    float _Y) restrict(amp);  
  
inline double remainder(  
    double _X,  
    double _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Returns `_X REM _Y`

## remainderf

Computes the remainder: `_X REM _Y`

```
inline float remainderf(  
    float _X,  
    float _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Returns `_X REM _Y`

## remquo

Computes the remainder of the first specified argument divided by the second specified argument. Also computes the quotient of the significand of the first specified argument divided by the significand of the second specified argument, and returns the quotient using the location specified in the third argument.

```
inline float remquo(  
    float _X,  
    float _Y,  
    _Out_ int* _Quo) restrict(amp);  
  
inline double remquo(  
    double _X,  
    double _Y,  
    _Out_ int* _Quo) restrict(amp);
```

### Parameters

`_X`

The first floating-point argument.

*\_Y*

The second floating-point argument.

*\_Quo*

[out] The address of an integer that's used to return the quotient of the fractional bits of `_x` divided by the fractional bits of `_Y`.

#### Return Value

Returns the remainder of `_x` divided by `_Y`.

## remquof

Computes the remainder of the first specified argument divided by the second specified argument. Also computes the quotient of the significand of the first specified argument divided by the significand of the second specified argument, and returns the quotient using the location specified in the third argument.

```
inline float remquof(  
    float _X,  
    float _Y,  
    _Out_ int* _Quo) restrict(amp);
```

#### Parameters

*\_X*

The first floating-point argument.

*\_Y*

The second floating-point argument.

*\_Quo*

[out] The address of an integer that's used to return the quotient of the fractional bits of `_x` divided by the fractional bits of `_Y`.

#### Return Value

Returns the remainder of `_x` divided by `_Y`.

## round

Rounds `_X` to the nearest integer

```
inline float round(float _X) restrict(amp);  
  
inline double round(double _X) restrict(amp);
```

#### Parameters

*\_X*

Floating-point value

#### Return Value

Returns the nearest integer of `_X`

## roundf

Rounds `_X` to the nearest integer

```
inline float roundf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the nearest integer of `_X`

## rsqrt

Returns the reciprocal of the square root of the argument

```
inline float rsqrt(float _X) restrict(amp);  
  
inline double rsqrt(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the reciprocal of the square root of the argument

## rsqrtf

Returns the reciprocal of the square root of the argument

```
inline float rsqrtf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the reciprocal of the square root of the argument

## scalb

Multiplies `_X` by FLT\_RADIX to the power `_Y`

```
inline float scalb(  
    float _X,  
    float _Y) restrict(amp);  
  
inline double scalb(  
    double _X,  
    double _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Returns  $_X * (\text{FLT\_RADIX} ^ _Y)$

## scalbf

Multiplies `_X` by `FLT_RADIX` to the power `_Y`

```
inline float scalbf(  
    float _X,  
    float _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Floating-point value

### Return Value

Returns  $_X * (\text{FLT\_RADIX} ^ _Y)$

## scalbn

Multiplies `_X` by `FLT_RADIX` to the power `_Y`

```
inline float scalbn(  
    float _X,  
    int _Y) restrict(amp);  
  
inline double scalbn(  
    double _X,  
    int _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Integer value

### Return Value

Returns  $_X * (\text{FLT\_RADIX} ^ _Y)$

## scalbnf

Multiplies `_X` by `FLT_RADIX` to the power `_Y`

```
inline float scalbnf(  
    float _X,  
    int _Y) restrict(amp);
```

### Parameters

`_X`

Floating-point value

`_Y`

Integer value

### Return Value

Returns  $_X * (\text{FLT\_RADIX} ^ _Y)$

## signbit

Determines whether the sign of `_X` is negative

```
inline int signbit(float _X) restrict(amp);  
  
inline int signbit(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns a nonzero value if and only if the sign of `_X` is negative

## signbitf

Determines whether the sign of `_X` is negative

```
inline int signbitf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns a nonzero value if and only if the sign of `_X` is negative

## sin

Calculates the sine value of the argument

```
inline float sin(float _X) restrict(amp);  
  
inline double sin(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the sine value of the argument

# sinf

Calculates the sine value of the argument

```
inline float sinf(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the sine value of the argument

# sincos

Calculates sine and cosine value of `_X`

```
inline void sincos(  
    float _X,  
    _Out_ float* _S,  
    _Out_ float* _C) restrict(amp);  
  
inline void sincos(  
    double _X,  
    _Out_ double* _S,  
    _Out_ double* _C) restrict(amp);
```

## Parameters

`_X`

Floating-point value

`_S`

Returns the sine value of `_X`

`_C`

Returns the cosine value of `_X`

# sincosf

Calculates sine and cosine value of `_X`

```
inline void sincosf(  
    float _X,  
    _Out_ float* _S,  
    _Out_ float* _C) restrict(amp);
```

## Parameters

`_X`

Floating-point value

`_S`

Returns the sine value of `_X`

`_C`

Returns the cosine value of `_X`

# sinh

Calculates the hyperbolic sine value of the argument

```
inline float sinh(float _X) restrict(amp);  
  
inline double sinh(double _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the hyperbolic sine value of the argument

# sinhf

Calculates the hyperbolic sine value of the argument

```
inline float sinhf(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the hyperbolic sine value of the argument

# sinpi

Calculates the sine value of  $\pi * \_X$

```
inline float sinpi(float _X) restrict(amp);  
  
inline double sinpi(double _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

## Return Value

Returns the sine value of  $\pi * \_X$

# sinpif

Calculates the sine value of  $\pi * \_X$

```
inline float sinpif(float _X) restrict(amp);
```

## Parameters

`_X`

Floating-point value

### Return Value

Returns the sine value of  $\pi * \_X$

## sqrt

Calculates the square root of the argument

```
inline float sqrt(float _X) restrict(amp);  
  
inline double sqrt(double _X) restrict(amp);
```

### Parameters

$\_X$

Floating-point value

### Return Value

Returns the square root of the argument

## sqrtf

Calculates the square root of the argument

```
inline float sqrtf(float _X) restrict(amp);
```

### Parameters

$\_X$

Floating-point value

### Return Value

Returns the square root of the argument

## tan

Calculates the tangent value of the argument

```
inline float tan(float _X) restrict(amp);  
  
inline double tan(double _X) restrict(amp);
```

### Parameters

$\_X$

Floating-point value

### Return Value

Returns the tangent value of the argument

## tanf

Calculates the tangent value of the argument

```
inline float tanf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the tangent value of the argument

## tanh

Calculates the hyperbolic tangent value of the argument

```
inline float tanh(float _X) restrict(amp);  
  
inline double tanh(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the hyperbolic tangent value of the argument

## tanhf

Calculates the hyperbolic tangent value of the argument

```
inline float tanhf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the hyperbolic tangent value of the argument

## tanpi

Calculates the tangent value of  $\pi * \_X$

```
inline float tanpi(float _X) restrict(amp);  
  
inline double tanpi(double _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the tangent value of  $\pi * \_X$

## tanpif

Calculates the tangent value of  $\pi * \_X$

```
inline float tanpif(float _X) restrict(amp);
```

#### Parameters

$\_X$

Floating-point value

#### Return Value

Returns the tangent value of  $\pi * \_X$

## tgamma

Computes the gamma function of  $\_X$

```
inline float tgamma(float _X) restrict(amp);  
  
inline double tgamma(double _X) restrict(amp);
```

#### Parameters

$\_X$

Floating-point value

#### Return Value

Returns the result of gamma function of  $\_X$

## tgammaf

Computes the gamma function of  $\_X$

```
inline float tgammaf(float _X) restrict(amp);
```

#### Parameters

$\_X$

Floating-point value

#### Return Value

Returns the result of gamma function of  $\_X$

## trunc

Truncates the argument to the integer component

```
inline float trunc(float _X) restrict(amp);  
  
inline double trunc(double _X) restrict(amp);
```

#### Parameters

$\_X$

Floating-point value

#### Return Value

Returns the integer component of the argument

## truncf

Truncates the argument to the integer component

```
inline float truncf(float _X) restrict(amp);
```

### Parameters

`_X`

Floating-point value

### Return Value

Returns the integer component of the argument

## See also

[Concurrency::precise\\_math Namespace](#)

# Concurrency Runtime

3/20/2019 • 6 minutes to read • [Edit Online](#)

The Concurrency Runtime for C++ helps you write robust, scalable, and responsive parallel applications. It raises the level of abstraction so that you do not have to manage the infrastructure details that are related to concurrency. You can also use it to specify scheduling policies that meet the quality of service demands of your applications. Use these resources to help you start working with the Concurrency Runtime.

For reference documentation, see [Reference](#).

## TIP

The Concurrency Runtime relies heavily on C++11 features and adopts the more modern C++ style. To learn more, read [Welcome Back to C++](#).

## Choosing Concurrency Runtime Features

<a href="#">Overview</a>	Teaches why the Concurrency Runtime is important and describes its key features.
<a href="#">Comparing to Other Concurrency Models</a>	Shows how the Concurrency Runtime compares to other concurrency models, such as the Windows thread pool and OpenMP, so that you can use the concurrency model that best fits your application requirements.
<a href="#">Migrating from OpenMP to the Concurrency Runtime</a>	Compares OpenMP to the Concurrency Runtime and provides examples about how to migrate existing OpenMP code to use the Concurrency Runtime.
<a href="#">Parallel Patterns Library (PPL)</a>	Introduces you to the PPL, which provides parallel loops, tasks, and parallel containers.
<a href="#">Asynchronous Agents Library</a>	Introduces you to how to use asynchronous agents and message passing to easily incorporate dataflow and pipelining tasks in your applications.
<a href="#">Task Scheduler</a>	Introduces you to the Task Scheduler, which enables you to fine-tune the performance of your desktop apps that uses the Concurrency Runtime.

## Task Parallelism in the PPL

<p><a href="#">Task Parallelism</a></p> <p><a href="#">How to: Use parallel_invoke to Write a Parallel Sort Routine</a></p> <p><a href="#">How to: Use parallel_invoke to Execute Parallel Operations</a></p> <p><a href="#">How to: Create a Task that Completes After a Delay</a></p>	<p>Describes tasks and task groups, which can help you to write asynchronous code and decompose parallel work into smaller pieces.</p>
<p><a href="#">Walkthrough: Implementing Futures</a></p>	<p>Demonstrates how to combine Concurrency Runtime features to do something more.</p>
<p><a href="#">Walkthrough: Removing Work from a User-Interface Thread</a></p>	<p>Shows how to move the work that is performed by the UI thread in a MFC application to a worker thread.</p>
<p><a href="#">Best Practices in the Parallel Patterns Library</a></p> <p><a href="#">General Best Practices in the Concurrency Runtime</a></p>	<p>Provides tips and best practices for working with the PPL.</p>

## Data Parallelism in the PPL

<p><a href="#">Parallel Algorithms</a></p> <p><a href="#">How to: Write a parallel_for Loop</a></p> <p><a href="#">How to: Write a parallel_for_each Loop</a></p> <p><a href="#">How to: Perform Map and Reduce Operations in Parallel</a></p>	<p>Describes <code>parallel_for</code>, <code>parallel_for_each</code>, <code>parallel_invoke</code>, and other parallel algorithms. Use parallel algorithms to solve <i>data parallel</i> problems that involve collections of data.</p>
<p><a href="#">Parallel Containers and Objects</a></p> <p><a href="#">How to: Use Parallel Containers to Increase Efficiency</a></p> <p><a href="#">How to: Use combinable to Improve Performance</a></p> <p><a href="#">How to: Use combinable to Combine Sets</a></p>	<p>Describes the <code>combinable</code> class, as well as <code>concurrent_vector</code>, <code>concurrent_queue</code>, <code>concurrent_unordered_map</code>, and other parallel containers. Use parallel containers and objects when you require containers that provide thread-safe access to their elements.</p>
<p><a href="#">Best Practices in the Parallel Patterns Library</a></p> <p><a href="#">General Best Practices in the Concurrency Runtime</a></p>	<p>Provides tips and best practices for working with the PPL.</p>

## Canceling Tasks and Parallel Algorithms

<p><a href="#">Cancellation in the PPL</a></p>	<p>Describes the role of cancellation in the PPL, including how to initiate and respond to cancellation requests.</p>
<p><a href="#">How to: Use Cancellation to Break from a Parallel Loop</a></p> <p><a href="#">How to: Use Exception Handling to Break from a Parallel Loop</a></p>	<p>Demonstrates two ways to cancel data-parallel work.</p>

## Universal Windows Platform apps

<a href="#">Creating Asynchronous Operations in C++ for UWP Apps</a>	Describes some of the key points to keep in mind when you use the Concurrency Runtime to produce asynchronous operations in a UWP app.
<a href="#">Walkthrough: Connecting Using Tasks and XML HTTP Requests</a>	Shows how to combine PPL tasks with the <code>IXMLHttpRequest2</code> and <code>IXMLHttpRequest2Callback</code> interfaces to send HTTP GET and POST requests to a web service in a UWP app.
<a href="#">Windows Runtime app samples</a>	Contains downloadable code samples and demo apps for Windows 8.x. The C++ samples use Concurrency Runtime features such as PPL tasks to process data in the background to keep the UX responsive.

## Dataflow Programming in the Asynchronous Agents Library

<a href="#">Asynchronous Agents</a> <a href="#">Asynchronous Message Blocks</a> <a href="#">Message Passing Functions</a> <a href="#">How to: Implement Various Producer-Consumer Patterns</a> <a href="#">How to: Provide Work Functions to the call and transformer Classes</a> <a href="#">How to: Use transformer in a Data Pipeline</a> <a href="#">How to: Select Among Completed Tasks</a> <a href="#">How to: Send a Message at a Regular Interval</a> <a href="#">How to: Use a Message Block Filter</a>	Describes asynchronous agents, message blocks, and message-passing functions, which are the building blocks for performing dataflow operations in the Concurrency Runtime.
<a href="#">Walkthrough: Creating an Agent-Based Application</a> <a href="#">Walkthrough: Creating a Dataflow Agent</a>	Shows how to create basic agent-based applications.
<a href="#">Walkthrough: Creating an Image-Processing Network</a>	Shows how to create a network of asynchronous message blocks that perform image processing.
<a href="#">Walkthrough: Using join to Prevent Deadlock</a>	Uses the dining philosophers problem to illustrate how to use the Concurrency Runtime to prevent deadlock in your application.
<a href="#">Walkthrough: Creating a Custom Message Block</a>	Shows how to create a custom message block type that orders incoming messages by priority.
<a href="#">Best Practices in the Asynchronous Agents Library</a> <a href="#">General Best Practices in the Concurrency Runtime</a>	Provides tips and best practices for working with agents.

## Exception Handling and Debugging

<a href="#">Exception Handling</a>	Describes how to work with exceptions in the Concurrency Runtime.
<a href="#">Parallel Diagnostic Tools</a>	Teaches you how to fine-tune your applications and make the most effective use of the Concurrency Runtime.

## Tuning Performance

<a href="#">Parallel Diagnostic Tools</a>	Teaches you how to fine-tune your applications and make the most effective use of the Concurrency Runtime.
<a href="#">Scheduler Instances</a> <a href="#">How to: Manage a Scheduler Instance</a> <a href="#">Scheduler Policies</a> <a href="#">How to: Specify Specific Scheduler Policies</a> <a href="#">How to: Create Agents that Use Specific Scheduler Policies</a>	Shows how to work with manage scheduler instances and scheduler policies. For desktop apps, scheduler policies enable you to associate specific rules with specific types of workloads. For example, you can create one scheduler instance to run some tasks at an elevated thread priority and use the default scheduler to run other tasks at the normal thread priority.
<a href="#">Schedule Groups</a> <a href="#">How to: Use Schedule Groups to Influence Order of Execution</a>	Demonstrates how to use schedule groups to affinitize, or group, related tasks together. For example, you might require a high degree of locality among related tasks when those tasks benefit from executing on the same processor node.
<a href="#">Lightweight Tasks</a>	Explains how lightweight tasks are useful for creating work that does not require load-balancing or cancellation, and how they are also useful for adapting existing code for use with the Concurrency Runtime.
<a href="#">Contexts</a> <a href="#">How to: Use the Context Class to Implement a Cooperative Semaphore</a> <a href="#">How to: Use Oversubscription to Offset Latency</a>	Describes how to control the behavior of the threads that are managed by the Concurrency Runtime.
<a href="#">Memory Management Functions</a> <a href="#">How to: Use Alloc and Free to Improve Memory Performance</a>	Describes the memory management functions that the Concurrency Runtime provides to help you allocate and free memory in a concurrent manner.

## Additional Resources

<a href="#">Async programming patterns and tips in Hilo (Windows Store apps using C++ and XAML)</a>	Learn how we used the Concurrency Runtime to implement asynchronous operations in Hilo, a Windows Runtime app using C++ and XAML.
<a href="#">Parallel Programming in Native Code blog</a>	Provides additional in-depth blog articles about parallel programming in the Concurrency Runtime.

<a href="#">Parallel Computing in C++ and Native Code forum</a>	Enables you to participate in community discussions about the Concurrency Runtime.
<a href="#">Parallel Programming</a>	Teaches you about the parallel programming model that is available in the .NET Framework.

## See also

[Reference](#)

# Overview of the Concurrency Runtime

3/4/2019 • 6 minutes to read • [Edit Online](#)

This document provides an overview of the Concurrency Runtime. It describes the benefits of the Concurrency Runtime, when to use it, and how its components interact with each other and with the operating system and applications.

## Sections

This document contains the following sections:

- [Concurrency Runtime implementation history](#)
- [Why a Runtime for Concurrency is Important](#)
- [Architecture](#)
- [C++ Lambda Expressions](#)
- [Requirements](#)

## Concurrency Runtime implementation history

In Visual Studio 2010 through 2013, the Concurrency Runtime was incorporated within `msvcr100.dll` through `msvcr120.dll`. When the UCRT refactoring occurred in Visual Studio 2015, that DLL was refactored into three parts:

- `ucrtbase.dll` – C API, shipped in Windows 10 and serviced downlevel via Windows Update-
- `vcruntime140.dll` – Compiler support functions and EH runtime, shipped via Visual Studio
- `concrtdll.dll` – Concurrency Runtime, shipped via Visual Studio. Required for parallel containers and algorithms such as `concurrency::parallel_for`. Also, the STL requires this DLL on Windows XP to power synchronization primitives, because Windows XP does not have condition variables.

In Visual Studio 2015 and later, the Concurrency Runtime Task Scheduler is no longer the scheduler for the task class and related types in `ppltasks.h`. Those types now use the Windows ThreadPool for better performance and interoperability with Windows synchronization primitives.

## Why a Runtime for Concurrency is Important

A runtime for concurrency provides uniformity and predictability to applications and application components that run simultaneously. Two examples of the benefits of the Concurrency Runtime are *cooperative task scheduling* and *cooperative blocking*.

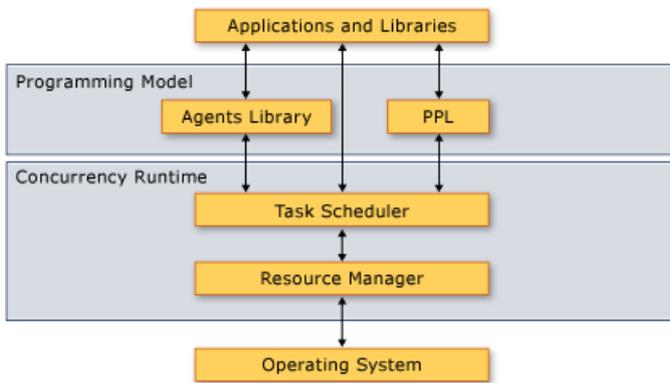
The Concurrency Runtime uses a cooperative task scheduler that implements a work-stealing algorithm to efficiently distribute work among computing resources. For example, consider an application that has two threads that are both managed by the same runtime. If one thread finishes its scheduled task, it can offload work from the other thread. This mechanism balances the overall workload of the application.

The Concurrency Runtime also provides synchronization primitives that use cooperative blocking to synchronize access to resources. For example, consider a task that must have exclusive access to a shared resource. By blocking cooperatively, the runtime can use the remaining quantum to perform another task as the first task waits for the resource. This mechanism promotes maximum usage of computing resources.

# Architecture

The Concurrency Runtime is divided into four components: the Parallel Patterns Library (PPL), the Asynchronous Agents Library, the Task Scheduler, and the Resource Manager. These components reside between the operating system and applications. The following illustration shows how the Concurrency Runtime components interact among the operating system and applications:

## Concurrency Runtime Architecture



### IMPORTANT

The Task Scheduler and Resource Manager components are not available from a Universal Windows Platform (UWP) app or when you use the task class or other types in `ppltasks.h`.

The Concurrency Runtime is highly *composable*, that is, you can combine existing functionality to do more. The Concurrency Runtime composes many features, such as parallel algorithms, from lower-level components.

The Concurrency Runtime also provides synchronization primitives that use cooperative blocking to synchronize access to resources. For more information about these synchronization primitives, see [Synchronization Data Structures](#).

The following sections provide a brief overview of what each component provides and when to use it.

### Parallel Patterns Library

The Parallel Patterns Library (PPL) provides general-purpose containers and algorithms for performing fine-grained parallelism. The PPL enables *imperative data parallelism* by providing parallel algorithms that distribute computations on collections or on sets of data across computing resources. It also enables *task parallelism* by providing task objects that distribute multiple independent operations across computing resources.

Use the Parallel Patterns Library when you have a local computation that can benefit from parallel execution. For example, you can use the `concurrency::parallel_for` algorithm to transform an existing `for` loop to act in parallel.

For more information about the Parallel Patterns Library, see [Parallel Patterns Library \(PPL\)](#).

### Asynchronous Agents Library

The Asynchronous Agents Library (or just *Agents Library*) provides both an actor-based programming model and message passing interfaces for coarse-grained dataflow and pipelining tasks. Asynchronous agents enable you to make productive use of latency by performing work as other components wait for data.

Use the Agents Library when you have multiple entities that communicate with each other asynchronously. For example, you can create an agent that reads data from a file or network connection and then uses the message passing interfaces to send that data to another agent.

For more information about the Agents Library, see [Asynchronous Agents Library](#).

### Task Scheduler

The Task Scheduler schedules and coordinates tasks at run time. The Task Scheduler is cooperative and uses a work-stealing algorithm to achieve maximum usage of processing resources.

The Concurrency Runtime provides a default scheduler so that you do not have to manage infrastructure details. However, to meet the quality needs of your application, you can also provide your own scheduling policy or associate specific schedulers with specific tasks.

For more information about the Task Scheduler, see [Task Scheduler](#).

### Resource Manager

The role of the Resource Manager is to manage computing resources, such as processors and memory. The Resource Manager responds to workloads as they change at run time by assigning resources to where they can be most effective.

The Resource Manager serves as an abstraction over computing resources and primarily interacts with the Task Scheduler. Although you can use the Resource Manager to fine-tune the performance of your libraries and applications, you typically use the functionality that is provided by the Parallel Patterns Library, the Agents Library, and the Task Scheduler. These libraries use the Resource Manager to dynamically rebalance resources as workloads change.

[\[Top\]](#)

## C++ Lambda Expressions

Many of the types and algorithms that are defined by the Concurrency Runtime are implemented as C++ templates. Some of these types and algorithms take as a parameter a routine that performs work. This parameter can be a lambda function, a function object, or a function pointer. These entities are also referred to as *work functions* or *work routines*.

Lambda expressions are an important new Visual C++ language feature because they provide a succinct way to define work functions for parallel processing. Function objects and function pointers enable you to use the Concurrency Runtime with your existing code. However, we recommend that you use lambda expressions when you write new code because of the safety and productivity benefits that they provide.

The following example compares the syntax of lambda functions, function objects, and function pointers in multiple calls to the `concurrency::parallel_for_each` algorithm. Each call to `parallel_for_each` uses a different technique to compute the square of each element in a `std::array` object.

```

// comparing-work-functions.cpp
// compile with: /EHsc
#include <pp1.h>
#include <array>
#include <iostream>

using namespace concurrency;
using namespace std;

// Function object (functor) class that computes the square of its input.
template<class Ty>
class SquareFunctor
{
public:
    void operator()(Ty& n) const
    {
        n *= n;
    }
};

// Function that computes the square of its input.
template<class Ty>
void square_function(Ty& n)
{
    n *= n;
}

int wmain()
{
    // Create an array object that contains 5 values.
    array<int, 5> values = { 1, 2, 3, 4, 5 };

    // Use a lambda function, a function object, and a function pointer to
    // compute the square of each element of the array in parallel.

    // Use a lambda function to square each element.
    parallel_for_each(begin(values), end(values), [](int& n){n *= n;});

    // Use a function object (functor) to square each element.
    parallel_for_each(begin(values), end(values), SquareFunctor<int>());

    // Use a function pointer to square each element.
    parallel_for_each(begin(values), end(values), &square_function<int>);

    // Print each element of the array to the console.
    for_each(begin(values), end(values), [](int& n) {
        wcout << n << endl;
    });
}

```

## Output

```

1
256
6561
65536
390625

```

For more information about lambda functions in C++, see [Lambda Expressions](#).

[\[Top\]](#)

## Requirements

The following table shows the header files that are associated with each component of the Concurrency Runtime:

COMPONENT	HEADER FILES
Parallel Patterns Library (PPL)	ppl.h concurrent_queue.h concurrent_vector.h
Asynchronous Agents Library	agents.h
Task Scheduler	concrth
Resource Manager	concrtrm.h

The Concurrency Runtime is declared in the [Concurrency](#) namespace. (You can also use [concurrency](#), which is an alias for this namespace.) The `concurrency::details` namespace supports the Concurrency Runtime framework, and is not intended to be used directly from your code.

The Concurrency Runtime is provided as part of the C Runtime Library (CRT). For more information about how to build an application that uses the CRT, see [CRT Library Features](#).

[\[Top\]](#)

# Exception Handling in the Concurrency Runtime

3/4/2019 • 16 minutes to read • [Edit Online](#)

The Concurrency Runtime uses C++ exception handling to communicate many kinds of errors. These errors include invalid use of the runtime, runtime errors such as failure to acquire a resource, and errors that occur in work functions that you provide to tasks and task groups. When a task or task group throws an exception, the runtime holds that exception and marshals it to the context that waits for the task or task group to finish. For components such as lightweight tasks and agents, the runtime does not manage exceptions for you. In these cases, you must implement your own exception-handling mechanism. This topic describes how the runtime handles exceptions that are thrown by tasks, task groups, lightweight tasks, and asynchronous agents, and how to respond to exceptions in your applications.

## Key Points

- When a task or task group throws an exception, the runtime holds that exception and marshals it to the context that waits for the task or task group to finish.
- When possible, surround every call to `concurrency::task::get` and `concurrency::task::wait` with a `try / catch` block to handle errors that you can recover from. The runtime terminates the app if a task throws an exception and that exception is not caught by the task, one of its continuations, or the main app.
- A task-based continuation always runs; it does not matter whether the antecedent task completed successfully, threw an exception, or was canceled. A value-based continuation does not run if the antecedent task throws or cancels.
- Because task-based continuations always run, consider whether to add a task-based continuation at the end of your continuation chain. This can help guarantee that your code observes all exceptions.
- The runtime throws `concurrency::task_canceled` when you call `concurrency::task::get` and that task is canceled.
- The runtime does not manage exceptions for lightweight tasks and agents.

## In this Document

- [Tasks and Continuations](#)
- [Task Groups and Parallel Algorithms](#)
- [Exceptions Thrown by the Runtime](#)
- [Multiple Exceptions](#)
- [Cancellation](#)
- [Lightweight Tasks](#)
- [Asynchronous Agents](#)

## Tasks and Continuations

This section describes how the runtime handles exceptions that are thrown by `concurrency::task` objects and their continuations. For more information about the task and continuation model, see [Task Parallelism](#).

When you throw an exception in the body of a work function that you pass to a `task` object, the runtime stores that exception and marshals it to the context that calls `concurrency::task::get` or `concurrency::task::wait`. The document [Task Parallelism](#) describes task-based versus value-based continuations, but to summarize, a value-based continuation takes a parameter of type `T` and a task-based continuation takes a parameter of type `task<T>`. If a task that throws has one or more value-based continuations, those continuations are not scheduled to run. The following example illustrates this behavior:

```
// eh-task.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    wcout << L"Running a task..." << endl;
    // Create a task that throws.
    auto t = create_task([]
    {
        throw exception();
    });

    // Create a continuation that prints its input value.
    auto continuation = t.then([]
    {
        // We do not expect this task to run because
        // the antecedent task threw.
        wcout << L"In continuation task..." << endl;
    });

    // Wait for the continuation to finish and handle any
    // error that occurs.
    try
    {
        wcout << L"Waiting for tasks to finish..." << endl;
        continuation.wait();

        // Alternatively, call get() to produce the same result.
        //continuation.get();
    }
    catch (const exception& e)
    {
        wcout << L"Caught exception." << endl;
    }
}
/* Output:
Running a task...
Waiting for tasks to finish...
Caught exception.
*/
```

A task-based continuation enables you to handle any exception that is thrown by the antecedent task. A task-based continuation always runs; it does not matter whether the task completed successfully, threw an exception, or was canceled. When a task throws an exception, its task-based continuations are scheduled to run. The following example shows a task that always throws. The task has two continuations; one is value-based and the other is task-based. The task-based exception always runs, and therefore can catch the exception that is thrown by the antecedent task. When the example waits for both continuations to finish, the exception is thrown again because the task exception is always thrown when `task::get` or `task::wait` is called.

```

// eh-continuations.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    wcout << L"Running a task..." << endl;
    // Create a task that throws.
    auto t = create_task([]() -> int
    {
        throw exception();
        return 42;
    });

    //
    // Attach two continuations to the task. The first continuation is
    // value-based; the second is task-based.

    // Value-based continuation.
    auto c1 = t.then([](int n)
    {
        // We don't expect to get here because the antecedent
        // task always throws.
        wcout << L"Received " << n << L'.' << endl;
    });

    // Task-based continuation.
    auto c2 = t.then([](task<int> previousTask)
    {
        // We do expect to get here because task-based continuations
        // are scheduled even when the antecedent task throws.
        try
        {
            wcout << L"Received " << previousTask.get() << L'.' << endl;
        }
        catch (const exception& e)
        {
            wcout << L"Caught exception from previous task." << endl;
        }
    });

    // Wait for the continuations to finish.
    try
    {
        wcout << L"Waiting for tasks to finish..." << endl;
        (c1 && c2).wait();
    }
    catch (const exception& e)
    {
        wcout << L"Caught exception while waiting for all tasks to finish." << endl;
    }
}
/* Output:
Running a task...
Waiting for tasks to finish...
Caught exception from previous task.
Caught exception while waiting for all tasks to finish.
*/

```

We recommend that you use task-based continuations to catch exceptions that you are able to handle. Because task-based continuations always run, consider whether to add a task-based continuation at the end of your continuation chain. This can help guarantee that your code observes all exceptions. The following example

shows a basic value-based continuation chain. The third task in the chain throws, and therefore any value-based continuations that follow it are not run. However, the final continuation is task-based, and therefore always runs. This final continuation handles the exception that is thrown by the third task.

We recommend that you catch the most specific exceptions that you can. You can omit this final task-based continuation if you don't have specific exceptions to catch. Any exception will remain unhandled and can terminate the app.

```

// eh-task-chain.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    int n = 1;
    create_task([n]
    {
        wcout << L"In first task. n = ";
        wcout << n << endl;

        return n * 2;

    }).then([](int n)
    {
        wcout << L"In second task. n = ";
        wcout << n << endl;

        return n * 2;

    }).then([](int n)
    {
        wcout << L"In third task. n = ";
        wcout << n << endl;

        // This task throws.
        throw exception();
        // Not reached.
        return n * 2;

    }).then([](int n)
    {
        // This continuation is not run because the previous task throws.
        wcout << L"In fourth task. n = ";
        wcout << n << endl;

        return n * 2;

    }).then([](task<int> previousTask)
    {
        // This continuation is run because it is value-based.
        try
        {
            // The call to task::get rethrows the exception.
            wcout << L"In final task. result = ";
            wcout << previousTask.get() << endl;
        }
        catch (const exception&)
        {
            wcout << L"<exception>" << endl;
        }
    }).wait();
}
/* Output:
    In first task. n = 1
    In second task. n = 2
    In third task. n = 4
    In final task. result = <exception>
*/

```

### TIP

You can use the `concurrency::task_completion_event::set_exception` method to associate an exception with a task completion event. The document [Task Parallelism](#) describes the `concurrency::task_completion_event` class in greater detail.

`concurrency::task_canceled` is an important runtime exception type that relates to `task`. The runtime throws `task_canceled` when you call `task::get` and that task is canceled. (Conversely, `task::wait` returns `task_status::canceled` and does not throw.) You can catch and handle this exception from a task-based continuation or when you call `task::get`. For more information about task cancellation, see [Cancellation in the PPL](#).

### Caution

Never throw `task_canceled` from your code. Call `concurrency::cancel_current_task` instead.

The runtime terminates the app if a task throws an exception and that exception is not caught by the task, one of its continuations, or the main app. If your application crashes, you can configure Visual Studio to break when C++ exceptions are thrown. After you diagnose the location of the unhandled exception, use a task-based continuation to handle it.

The section [Exceptions Thrown by the Runtime](#) in this document describes how to work with runtime exceptions in greater detail.

[\[Top\]](#)

## Task Groups and Parallel Algorithms

This section describes how the runtime handles exceptions that are thrown by task groups. This section also applies to parallel algorithms such as `concurrency::parallel_for`, because these algorithms build on task groups.

### Caution

Make sure that you understand the effects that exceptions have on dependent tasks. For recommended practices about how to use exception handling with tasks or parallel algorithms, see the [Understand how Cancellation and Exception Handling Affect Object Destruction](#) section in the Best Practices in the Parallel Patterns Library topic.

For more information about task groups, see [Task Parallelism](#). For more information about parallel algorithms, see [Parallel Algorithms](#).

When you throw an exception in the body of a work function that you pass to a `concurrency::task_group` or `concurrency::structured_task_group` object, the runtime stores that exception and marshals it to the context that calls `concurrency::task_group::wait`, `concurrency::structured_task_group::wait`, `concurrency::task_group::run_and_wait`, or `concurrency::structured_task_group::run_and_wait`. The runtime also stops all active tasks that are in the task group (including those in child task groups) and discards any tasks that have not yet started.

The following example shows the basic structure of a work function that throws an exception. The example uses a `task_group` object to print the values of two `point` objects in parallel. The `print_point` work function prints the values of a `point` object to the console. The work function throws an exception if the input value is `NULL`. The runtime stores this exception and marshals it to the context that calls `task_group::wait`.

```

// eh-task-group.cpp
// compile with: /EHsc
#include <ppl.h>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

// Defines a basic point with X and Y coordinates.
struct point
{
    int X;
    int Y;
};

// Prints the provided point object to the console.
void print_point(point* pt)
{
    // Throw an exception if the value is NULL.
    if (pt == NULL)
    {
        throw exception("point is NULL.");
    }

    // Otherwise, print the values of the point.
    wstringstream ss;
    ss << L"X = " << pt->X << L", Y = " << pt->Y << endl;
    wcout << ss.str();
}

int wmain()
{
    // Create a few point objects.
    point pt = {15, 30};
    point* pt1 = &pt;
    point* pt2 = NULL;

    // Use a task group to print the values of the points.
    task_group tasks;

    tasks.run([&] {
        print_point(pt1);
    });

    tasks.run([&] {
        print_point(pt2);
    });

    // Wait for the tasks to finish. If any task throws an exception,
    // the runtime marshals it to the call to wait.
    try
    {
        tasks.wait();
    }
    catch (const exception& e)
    {
        wcerr << L"Caught exception: " << e.what() << endl;
    }
}

```

This example produces the following output.

```
X = 15, Y = 30Caught exception: point is NULL.
```

For a complete example that uses exception handling in a task group, see [How to: Use Exception Handling to Break from a Parallel Loop](#).

[\[Top\]](#)

## Exceptions Thrown by the Runtime

An exception can result from a call to the runtime. Most exception types, except for `concurrency::task_canceled` and `concurrency::operation_timed_out`, indicate a programming error. These errors are typically unrecoverable, and therefore should not be caught or handled by application code. We suggest that you only catch or handle unrecoverable errors in your application code when you need to diagnose programming errors. However, understanding the exception types that are defined by the runtime can help you diagnose programming errors.

The exception handling mechanism is the same for exceptions that are thrown by the runtime as exceptions that are thrown by work functions. For example, the `concurrency::receive` function throws `operation_timed_out` when it does not receive a message in the specified time period. If `receive` throws an exception in a work function that you pass to a task group, the runtime stores that exception and marshals it to the context that calls `task_group::wait`, `structured_task_group::wait`, `task_group::run_and_wait`, or `structured_task_group::run_and_wait`.

The following example uses the `concurrency::parallel_invoke` algorithm to run two tasks in parallel. The first task waits five seconds and then sends a message to a message buffer. The second task uses the `receive` function to wait three seconds to receive a message from the same message buffer. The `receive` function throws `operation_timed_out` if it does not receive the message in the time period.

```

// eh-time-out.cpp
// compile with: /EHsc
#include <agents.h>
#include <ppl.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    single_assignment<int> buffer;
    int result;

    try
    {
        // Run two tasks in parallel.
        parallel_invoke(
            // This task waits 5 seconds and then sends a message to
            // the message buffer.
            [&] {
                wait(5000);
                send(buffer, 42);
            },
            // This task waits 3 seconds to receive a message.
            // The receive function throws operation_timed_out if it does
            // not receive a message in the specified time period.
            [&] {
                result = receive(buffer, 3000);
            }
        );

        // Print the result.
        wcout << L"The result is " << result << endl;
    }
    catch (operation_timed_out&)
    {
        wcout << L"The operation timed out." << endl;
    }
}

```

This example produces the following output.

```
The operation timed out.
```

To prevent abnormal termination of your application, make sure that your code handles exceptions when it calls into the runtime. Also handle exceptions when you call into external code that uses the Concurrency Runtime, for example, a third-party library.

[\[Top\]](#)

## Multiple Exceptions

If a task or parallel algorithm receives multiple exceptions, the runtime marshals only one of those exceptions to the calling context. The runtime does not guarantee which exception it marshals.

The following example uses the `parallel_for` algorithm to print numbers to the console. It throws an exception if the input value is less than some minimum value or greater than some maximum value. In this example, multiple work functions can throw an exception.

```

// eh-multiple.cpp
// compile with: /EHsc
#include <ppl.h>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

int wmain()
{
    const int min = 0;
    const int max = 10;

    // Print values in a parallel_for loop. Use a try-catch block to
    // handle any exceptions that occur in the loop.
    try
    {
        parallel_for(-5, 20, [min,max](int i)
        {
            // Throw an exception if the input value is less than the
            // minimum or greater than the maximum.

            // Otherwise, print the value to the console.

            if (i < min)
            {
                stringstream ss;
                ss << i << ": the value is less than the minimum.";
                throw exception(ss.str().c_str());
            }
            else if (i > max)
            {
                stringstream ss;
                ss << i << ": the value is greater than than the maximum.";
                throw exception(ss.str().c_str());
            }
            else
            {
                wstringstream ss;
                ss << i << endl;
                wcout << ss.str();
            }
        });
    }
    catch (exception& e)
    {
        // Print the error to the console.
        wcerr << L"Caught exception: " << e.what() << endl;
    }
}

```

The following shows sample output for this example.

```
8293104567Caught exception: -5: the value is less than the minimum.
```

[\[Top\]](#)

## Cancellation

Not all exceptions indicate an error. For example, a search algorithm might use exception handling to stop its associated task when it finds the result. For more information about how to use cancellation mechanisms in your code, see [Cancellation in the PPL](#).

[\[Top\]](#)

## Lightweight Tasks

A lightweight task is a task that you schedule directly from a `concurrency::Scheduler` object. Lightweight tasks carry less overhead than ordinary tasks. However, the runtime does not catch exceptions that are thrown by lightweight tasks. Instead, the exception is caught by the unhandled exception handler, which by default terminates the process. Therefore, use an appropriate error-handling mechanism in your application. For more information about lightweight tasks, see [Task Scheduler](#).

[\[Top\]](#)

## Asynchronous Agents

Like lightweight tasks, the runtime does not manage exceptions that are thrown by asynchronous agents.

The following example shows one way to handle exceptions in a class that derives from `concurrency::agent`. This example defines the `points_agent` class. The `points_agent::run` method reads `point` objects from the message buffer and prints them to the console. The `run` method throws an exception if it receives a `NULL` pointer.

The `run` method surrounds all work in a `try-catch` block. The `catch` block stores the exception in a message buffer. The application checks whether the agent encountered an error by reading from this buffer after the agent finishes.

```
// eh-agents.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Defines a point with x and y coordinates.
struct point
{
    int X;
    int Y;
};

// Informs the agent to end processing.
point sentinel = {0,0};

// An agent that prints point objects to the console.
class point_agent : public agent
{
public:
    explicit point_agent(unbounded_buffer<point*>& points)
        : _points(points)
    {
    }

    // Retrieves any exception that occurred in the agent.
    bool get_error(exception& e)
    {
        return try_receive(_error, e);
    }

protected:
    // Performs the work of the agent.
    void run()
    {
        // Perform processing in a try block.
        try
```

```

    {
        // Read from the buffer until we reach the sentinel value.
        while (true)
        {
            // Read a value from the message buffer.
            point* r = receive(_points);

            // In this example, it is an error to receive a
            // NULL point pointer. In this case, throw an exception.
            if (r == NULL)
            {
                throw exception("point must not be NULL");
            }
            // Break from the loop if we receive the
            // sentinel value.
            else if (r == &sentinel)
            {
                break;
            }
            // Otherwise, do something with the point.
            else
            {
                // Print the point to the console.
                wcout << L"X: " << r->X << L" Y: " << r->Y << endl;
            }
        }
    }
    // Store the error in the message buffer.
    catch (exception& e)
    {
        send(_error, e);
    }

    // Set the agent status to done.
    done();
}

private:
    // A message buffer that receives point objects.
    unbounded_buffer<point*> _points;

    // A message buffer that stores error information.
    single_assignment<exception> _error;
};

int wmain()
{
    // Create a message buffer so that we can communicate with
    // the agent.
    unbounded_buffer<point*> buffer;

    // Create and start a point_agent object.
    point_agent a(buffer);
    a.start();

    // Send several points to the agent.
    point r1 = {10, 20};
    point r2 = {20, 30};
    point r3 = {30, 40};

    send(buffer, &r1);
    send(buffer, &r2);
    // To illustrate exception handling, send the NULL pointer to the agent.
    send(buffer, reinterpret_cast<point*>(NULL));
    send(buffer, &r3);
    send(buffer, &sentinel);

    // Wait for the agent to finish.
    agent::wait(&a);
}

```

```

// Check whether the agent encountered an error.
exception e;
if (a.get_error(e))
{
    cout << "error occurred in agent: " << e.what() << endl;
}

// Print out agent status.
wcout << L"the status of the agent is: ";
switch (a.status())
{
case agent_created:
    wcout << L"created";
    break;
case agent_runnable:
    wcout << L"runnable";
    break;
case agent_started:
    wcout << L"started";
    break;
case agent_done:
    wcout << L"done";
    break;
case agent_canceled:
    wcout << L"canceled";
    break;
default:
    wcout << L"unknown";
    break;
}
wcout << endl;
}

```

This example produces the following output.

```

X: 10 Y: 20
X: 20 Y: 30
error occurred in agent: point must not be NULL
the status of the agent is: done

```

Because the `try - catch` block exists outside the `while` loop, the agent ends processing when it encounters the first error. If the `try - catch` block was inside the `while` loop, the agent would continue after an error occurs.

This example stores exceptions in a message buffer so that another component can monitor the agent for errors as it runs. This example uses a [concurrency::single\\_assignment](#) object to store the error. In the case where an agent handles multiple exceptions, the `single_assignment` class stores only the first message that is passed to it. To store only the last exception, use the [concurrency::overwrite\\_buffer](#) class. To store all exceptions, use the [concurrency::unbounded\\_buffer](#) class. For more information about these message blocks, see [Asynchronous Message Blocks](#).

For more information about asynchronous agents, see [Asynchronous Agents](#).

[\[Top\]](#)

## Summary

[\[Top\]](#)

## See also

Concurrency Runtime  
Task Parallelism  
Parallel Algorithms  
Cancellation in the PPL  
Task Scheduler  
Asynchronous Agents

# Parallel Diagnostic Tools (Concurrency Runtime)

3/4/2019 • 2 minutes to read • [Edit Online](#)

Visual Studio provides extensive support for debugging and profiling multi-threaded applications.

## Debugging

The Visual Studio debugger includes the **Parallel Stacks** window, **Parallel Tasks** window, and **Parallel Watch** window. For more information, see [Walkthrough: Debugging a Parallel Application](#) and [How to: Use the Parallel Watch Window](#).

## Profiling

The profiling tools provide three data views that display graphical, tabular and numerical information about how a multi-threaded application interacts with itself and with other programs. The views enable you to quickly identify areas of concern, and to navigate from points on the graphical displays to call stacks, call sites, and source code. For more information, see [Concurrency Visualizer](#).

## Event Tracing

The Concurrency Runtime uses [Event Tracing for Windows](#) (ETW) to notify instrumentation tools, such as profilers, when various events occur. These events include when a scheduler is activated or deactivated, when a context begins, ends, blocks, unblocks, or yields, and when a parallel algorithm begins or ends.

Tools such as the [Concurrency Visualizer](#) utilize this functionality; therefore, you typically do not have to work with these events directly. However, these events are useful when you are developing a custom profiler or when you use event tracing tools such as [Xperf](#).

The Concurrency Runtime raises these events only when tracing is enabled. Call the [concurrency::EnableTracing](#) function to enable event tracing and the [concurrency::DisableTracing](#) function to disable tracing.

The following table describes the events that the runtime raises when event tracing is enabled:

EVENT	DESCRIPTION	VALUE
<a href="#">concurrency::ConcRT_ProviderGuid</a>	The ETW provider identifier for the Concurrency Runtime.	f7b697a3-4db5-4d3b-be71-c4d284e6592f
<a href="#">concurrency::ContextEventGuid</a>	Marks events that are related to contexts.	5727a00f-50be-4519-8256-f7699871fecb
<a href="#">concurrency::PPLParallelForEventGuid</a>	Marks the entrance and exit to calls to the <a href="#">concurrency::parallel_for</a> algorithm.	31c8da6b-6165-4042-8b92-949e315f4d84
<a href="#">concurrency::PPLParallelForeachEventGuid</a>	Marks the entrance and exit to calls to the <a href="#">concurrency::parallel_for_each</a> algorithm.	5cb7d785-9d66-465d-bae1-4611061b5434
<a href="#">concurrency::PPLParallelInvokeEventGuid</a>	Marks the entrance and exit to calls to the <a href="#">concurrency::parallel_invoke</a> algorithm.	d1b5b133-ec3d-49f4-98a3-464d1a9e4682

EVENT	DESCRIPTION	VALUE
<a href="#">concurrency::SchedulerEventGuid</a>	Marks events that are related to the <a href="#">Task Scheduler</a> .	e2091f8a-1e0a-4731-84a2-0dd57c8a5261
<a href="#">concurrency::VirtualProcessorEventGuid</a>	Marks events that are related to virtual processors.	2f27805f-1676-4ecc-96fa-7eb09d44302f

The Concurrency Runtime defines, but does not currently raise, the following events. The runtime reserves these events for future use:

- [concurrency::ConcRTEventGuid](#)
- [concurrency::ScheduleGroupEventGuid](#)
- [concurrency::ChoreEventGuid](#)
- [concurrency::LockEventGuid](#)
- [concurrency::ResourceManagerEventGuid](#)

The [concurrency::ConcRT\\_EventType](#) enumeration specifies the possible operations that an event tracks. For example, at the entrance of the `parallel_for` algorithm, the runtime raises the `PPLParallelForEventGuid` event and provides `CONCRT_EVENT_START` as the operation. Before the `parallel_for` algorithm returns, the runtime again raises the `PPLParallelForEventGuid` event and provides `CONCRT_EVENT_END` as the operation.

The following example illustrates how to enable tracing for a call to `parallel_for`. The runtime does not trace the first call to `parallel_for` because tracing is not enabled. The call to `EnableTracing` enables the runtime to trace the second call to `parallel_for`.

```
// etw.cpp
// compile with: /EHsc
#include <ppl.h>

using namespace concurrency;

int wmain()
{
    // Perform some parallel work.
    // Event tracing is disabled at this point.
    parallel_for(0, 10000, [](int i) {
        // TODO: Perform work.
    });

    // Enable tracing for a second call to parallel_for.
    EnableTracing();
    parallel_for(0, 10000, [](int i) {
        // TODO: Perform work.
    });
    DisableTracing();
}
```

The runtime tracks the number of times that you call `EnableTracing` and `DisableTracing`. Therefore, if you call `EnableTracing` multiple times, you must call `DisableTracing` the same number of times in order to disable tracing.

## See also

[Concurrency Runtime](#)

# Creating Asynchronous Operations in C++ for UWP Apps

3/5/2019 • 23 minutes to read • [Edit Online](#)

This document describes some of the key points to keep in mind when you use the task class to produce Windows ThreadPool-based asynchronous operations in a Universal Windows Runtime (UWP) app.

The use of asynchronous programming is a key component in the Windows Runtime app model because it enables apps to remain responsive to user input. You can start a long-running task without blocking the UI thread, and you can receive the results of the task later. You can also cancel tasks and receive progress notifications as tasks run in the background. The document [Asynchronous programming in C++](#) provides an overview of the asynchronous pattern that's available in Visual C++ to create UWP apps. That document teaches how to both consume and create chains of asynchronous Windows Runtime operations. This section describes how to use the types in `ppltasks.h` to produce asynchronous operations that can be consumed by another Windows Runtime component and how to control how asynchronous work is executed. Also consider reading [Async programming patterns and tips in Hilo \(Windows Store apps using C++ and XAML\)](#) to learn how we used the task class to implement asynchronous operations in Hilo, a Windows Runtime app using C++ and XAML.

## NOTE

You can use the [Parallel Patterns Library](#) (PPL) and [Asynchronous Agents Library](#) in a UWP app. However, you cannot use the Task Scheduler or the Resource Manager. This document describes additional features that the PPL provides that are available only to a UWP app, and not to a desktop app.

## Key points

- Use [concurrency::create\\_async](#) to create asynchronous operations that can be used by other components (which might be written in languages other than C++).
- Use [concurrency::progress\\_reporter](#) to report progress notifications to components that call your asynchronous operations.
- Use cancellation tokens to enable internal asynchronous operations to cancel.
- The behavior of the `create_async` function depends on the return type of the work function that is passed to it. A work function that returns a task (either `task<T>` or `task<void>`) runs synchronously in the context that called `create_async`. A work function that returns `T` or `void` runs in an arbitrary context.
- You can use the [concurrency::task::then](#) method to create a chain of tasks that run one after another. In a UWP app, the default context for a task's continuations depends on how that task was constructed. If the task was created by passing an asynchronous action to the task constructor, or by passing a lambda expression that returns an asynchronous action, then the default context for all continuations of that task is the current context. If the task is not constructed from an asynchronous action, then an arbitrary context is used by default for the task's continuations. You can override the default context with the [concurrency::task\\_continuation\\_context](#) class.

## In this document

- [Creating Asynchronous Operations](#)

- [Example: Creating a C++ Windows Runtime Component](#)
- [Controlling the Execution Thread](#)
- [Example: Controlling Execution in a Windows Runtime App with C++ and XAML](#)

## Creating Asynchronous Operations

You can use the task and continuation model in the Parallel Patterns Library (PPL) to define background tasks as well as additional tasks that run when the previous task completes. This functionality is provided by the `concurrency::task` class. For more information about this model and the `task` class, see [Task Parallelism](#).

The Windows Runtime is a programming interface that you can use to create UWP apps that run only in a special operating system environment. Such apps use authorized functions, data types, and devices, and are distributed from the Microsoft Store. The Windows Runtime is represented by the *Application Binary Interface* (ABI). The ABI is an underlying binary contract that makes Windows Runtime APIs available to programming languages such as Visual C++.

By using the Windows Runtime, you can use the best features of various programming languages and combine them into one app. For example, you might create your UI in JavaScript and perform the computationally-intensive app logic in a C++ component. The ability to perform these computationally-intensive operations in the background is a key factor in keeping your UI responsive. Because the `task` class is specific to C++, you must use a Windows Runtime interface to communicate asynchronous operations to other components (which might be written in languages other than C++). The Windows Runtime provides four interfaces that you can use to represent asynchronous operations:

### [Windows::Foundation::IAsyncAction](#)

Represents an asynchronous action.

### [Windows::Foundation::IAsyncActionWithProgress<TProgress>](#)

Represents an asynchronous action that reports progress.

### [Windows::Foundation::IAsyncOperation<TResult>](#)

Represents an asynchronous operation that returns a result.

### [Windows::Foundation::IAsyncOperationWithProgress<TResult, TProgress>](#)

Represents an asynchronous operation that returns a result and reports progress.

The notion of an *action* means that the asynchronous task doesn't produce a value (think of a function that returns `void`). The notion of an *operation* means that the asynchronous task does produce a value. The notion of *progress* means that the task can report progress messages to the caller. JavaScript, the .NET Framework, and Visual C++ each provides its own way to create instances of these interfaces for use across the ABI boundary. For Visual C++, the PPL provides the `concurrency::create_async` function. This function creates a Windows Runtime asynchronous action or operation that represents the completion of a task. The `create_async` function takes a work function (typically a lambda expression), internally creates a `task` object, and wraps that task in one of the four asynchronous Windows Runtime interfaces.

#### NOTE

Use `create_async` only when you have to create functionality that can be accessed from another language or another Windows Runtime component. Use the `task` class directly when you know that the operation is both produced and consumed by C++ code in the same component.

The return type of `create_async` is determined by the type of its arguments. For example, if your work function doesn't return a value and doesn't report progress, `create_async` returns `IAsyncAction`. If your work function doesn't return a value and also reports progress, `create_async` returns `IAsyncActionWithProgress`. To report

progress, provide a `concurrency::progress_reporter` object as the parameter to your work function. The ability to report progress enables you to report what amount of work was performed and what amount still remains (for example, as a percentage). It also enables you to report results as they become available.

The `IAsyncAction`, `IAsyncActionWithProgress<TProgress>`, `IAsyncOperation<TResult>`, and `IAsyncActionOperationWithProgress<TProgress, TProgress>` interfaces each provide a `Cancel` method that enables you to cancel the asynchronous operation. The `task` class works with cancellation tokens. When you use a cancellation token to cancel work, the runtime does not start new work that subscribes to that token. Work that is already active can monitor its cancellation token and stop when it can. This mechanism is described in greater detail in the document [Cancellation in the PPL](#). You can connect task cancellation with the Windows Runtime `Cancel` methods in two ways. First, you can define the work function that you pass to `create_async` to take a `concurrency::cancellation_token` object. When the `Cancel` method is called, this cancellation token is cancelled and the normal cancellation rules apply to the underlying `task` object that supports the `create_async` call. If you do not provide a `cancellation_token` object, the underlying `task` object defines one implicitly. Define a `cancellation_token` object when you need to cooperatively respond to cancellation in your work function. The section [Example: Controlling Execution in a Windows Runtime App with C++ and XAML](#) shows an example of how to perform cancellation in a Universal Windows Platform (UWP) app with C# and XAML that uses a custom Windows Runtime C++ component.

### WARNING

In a chain of task continuations, always clean up state and then call `concurrency::cancel_current_task` when the cancellation token is cancelled. If you return early instead of calling `cancel_current_task`, the operation transitions to the completed state instead of the canceled state.

The following table summarizes the combinations that you can use to define asynchronous operations in your app.

TO CREATE THIS WINDOWS RUNTIME INTERFACE	RETURN THIS TYPE FROM <code>CREATE_ASYNC</code>	PASS THESE PARAMETER TYPES TO YOUR WORK FUNCTION TO USE AN IMPLICIT CANCELLATION TOKEN	PASS THESE PARAMETER TYPES TO YOUR WORK FUNCTION TO USE AN EXPLICIT CANCELLATION TOKEN
<code>IAsyncAction</code>	<code>void</code> or <code>task&lt;void&gt;</code>	(none)	( <code>cancellation_token</code> )
<code>IAsyncActionWithProgress&lt;TProgress&gt;</code>	<code>void</code> or <code>task&lt;void&gt;</code>	( <code>progress_reporter</code> )	( <code>progress_reporter</code> , <code>cancellation_token</code> )
<code>IAsyncOperation&lt;TResult&gt;</code>	<code>T</code> or <code>task&lt;T&gt;</code>	(none)	( <code>cancellation_token</code> )
<code>IAsyncActionOperationWithProgress&lt;TResult, TProgress&gt;</code>	<code>T</code> or <code>task&lt;T&gt;</code>	( <code>progress_reporter</code> )	( <code>progress_reporter</code> , <code>cancellation_token</code> )

You can return a value or a `task` object from the work function that you pass to the `create_async` function. These variations produce different behaviors. When you return a value, the work function is wrapped in a `task` so that it can be run on a background thread. In addition, the underlying `task` uses an implicit cancellation token. Conversely, if you return a `task` object, the work function runs synchronously. Therefore, if you return a `task` object, ensure that any lengthy operations in your work function also run as tasks to enable your app to remain responsive. In addition, the underlying `task` does not use an implicit cancellation token. Therefore, you need to define your work function to take a `cancellation_token` object if you require support for cancellation when you return a `task` object from `create_async`.

The following example shows the various ways to create an `IAsyncAction` object that can be consumed by another

Windows Runtime component.

```
// Creates an IAsyncAction object and uses an implicit cancellation token.
auto op1 = create_async([]
{
    // Define work here.
});

// Creates an IAsyncAction object and uses no cancellation token.
auto op2 = create_async([]
{
    return create_task([]
    {
        // Define work here.
    });
});

// Creates an IAsyncAction object and uses an explicit cancellation token.
auto op3 = create_async([](cancellation_token ct)
{
    // Define work here.
});

// Creates an IAsyncAction object that runs another task and also uses an explicit cancellation token.
auto op4 = create_async([](cancellation_token ct)
{
    return create_task([ct]()
    {
        // Define work here.
    });
});
```

## Example: Creating a C++ Windows Runtime Component and Consuming it from C#

Consider an app that uses XAML and C# to define the UI and a C++ Windows Runtime component to perform compute-intensive operations. In this example, the C++ component computes which numbers in a given range are prime. To illustrate the differences among the four Windows Runtime asynchronous task interfaces, start, in Visual Studio, by creating a **Blank Solution** and naming it `Primes`. Then add to the solution a **Windows Runtime Component** project and naming it `PrimesLibrary`. Add the following code to the generated C++ header file (this example renames `Class1.h` to `Primes.h`). Each `public` method defines one of the four asynchronous interfaces. The methods that return a value return a [Windows::Foundation::Collections::IVector<int>](#) object. The methods that report progress produce `double` values that define the percentage of overall work that has completed.

```

#pragma once

namespace PrimesLibrary
{
    public ref class Primes sealed
    {
    public:
        Primes();

        // Computes the numbers that are prime in the provided range and stores them in an internal variable.
        Windows::Foundation::IAsyncAction^ ComputePrimesAsync(int first, int last);

        // Computes the numbers that are prime in the provided range and stores them in an internal variable.
        // This version also reports progress messages.
        Windows::Foundation::IAsyncActionWithProgress<double>^ ComputePrimesWithProgressAsync(int first, int
last);

        // Gets the numbers that are prime in the provided range.
        Windows::Foundation::IAsyncOperation<Windows::Foundation::Collections::IVector<int>^>^
GetPrimesAsync(int first, int last);

        // Gets the numbers that are prime in the provided range. This version also reports progress messages.
        Windows::Foundation::IAsyncOperationWithProgress<Windows::Foundation::Collections::IVector<int>^,
double>^ GetPrimesWithProgressAsync(int first, int last);
    };
}

```

#### NOTE

By convention, asynchronous method names in the Windows Runtime typically end with "Async".

Add the following code to the generated C++ source file (this example renames Class1.cpp to Primes.cpp). The `is_prime` function determines whether its input is prime. The remaining methods implement the `Primes` class. Each call to `create_async` uses a signature that's compatible with the method from which it is called. For example, because `Primes::ComputePrimesAsync` returns `IAsyncAction`, the work function that's provided to `create_async` doesn't return a value and doesn't take a `progress_reporter` object as its parameter.

```

// PrimesLibrary.cpp
#include "pch.h"
#include "Primes.h"
#include <atomic>
#include <collection.h>
#include <ppltasks.h>
#include <concurrent_vector.h>

using namespace concurrency;
using namespace std;

using namespace Platform;
using namespace Platform::Collections;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;

using namespace PrimesLibrary;

Primes::Primes()
{
}

// Determines whether the input value is prime.
bool is_prime(int n)
{
    if (n < 2)

```

```

    if (n < 2)
    {
        return false;
    }
    for (int i = 2; i < n; ++i)
    {
        if ((n % i) == 0)
        {
            return false;
        }
    }
    return true;
}

// Adds the numbers that are prime in the provided range
// to the primes global variable.
IAsyncAction^ Primes::ComputePrimesAsync(int first, int last)
{
    return create_async([this, first, last]
    {
        // Ensure that the input values are in range.
        if (first < 0 || last < 0)
        {
            throw ref new InvalidArgumentException();
        }
        // Perform the computation in parallel.
        parallel_for(first, last + 1, [this](int n)
        {
            if (is_prime(n))
            {
                // Perhaps store the value somewhere...
            }
        });
    });
}

IAsyncActionWithProgress<double>^ Primes::ComputePrimesWithProgressAsync(int first, int last)
{
    return create_async([first, last](progress_reporter<double> reporter)
    {
        // Ensure that the input values are in range.
        if (first < 0 || last < 0)
        {
            throw ref new InvalidArgumentException();
        }
        // Perform the computation in parallel.
        atomic<long> operation = 0;
        long range = last - first + 1;
        double lastPercent = 0.0;
        parallel_for(first, last + 1, [&operation, range, &lastPercent, reporter](int n)
        {
            // Report progress message.
            double progress = 100.0 * (++operation) / range;
            if (progress >= lastPercent)
            {
                reporter.report(progress);
                lastPercent += 1.0;
            }

            if (is_prime(n))
            {
                // Perhaps store the value somewhere...
            }
        });
        reporter.report(100.0);
    });
}

IAsyncOperation<IVector<int>^>^ Primes::GetPrimesAsync(int first, int last)

```

```

{
    return create_async([this, first, last]() -> IVector<int>^
    {
        // Ensure that the input values are in range.
        if (first < 0 || last < 0)
        {
            throw ref new InvalidArgumentException();
        }
        // Perform the computation in parallel.
        concurrent_vector<int> primes;
        parallel_for(first, last + 1, [this, &primes](int n)
        {
            // If the value is prime, add it to the global vector.
            if (is_prime(n))
            {
                primes.push_back(n);
            }
        });
        // Sort the results.
        sort(begin(primes), end(primes), less<int>());

        // Copy the results to an IVector object. The IVector
        // interface makes collections of data available to other
        // Windows Runtime components.
        auto results = ref new Vector<int>();
        for (int prime : primes)
        {
            results->Append(prime);
        }
        return results;
    });
}

IAsyncOperationWithProgress<IVector<int>^, double>^ Primes::GetPrimesWithProgressAsync(int first, int last)
{
    return create_async([this, first, last](progress_reporter<double> reporter) -> IVector<int>^
    {
        // Ensure that the input values are in range.
        if (first < 0 || last < 0)
        {
            throw ref new InvalidArgumentException();
        }
        // Perform the computation in parallel.
        concurrent_vector<int> primes;
        long operation = 0;
        long range = last - first + 1;
        double lastPercent = 0.0;
        parallel_for(first, last + 1, [&primes, &operation, range, &lastPercent, reporter](int n)
        {
            // Report progress message.
            double progress = 100.0 * (++operation) / range;
            if (progress >= lastPercent)
            {
                reporter.report(progress);
                lastPercent += 1.0;
            }

            // If the value is prime, add it to the local vector.
            if (is_prime(n))
            {
                primes.push_back(n);
            }
        });
        reporter.report(100.0);

        // Sort the results.
        sort(begin(primes), end(primes), less<int>());

        // Copy the results to an IVector object. The IVector

```

```
// interface makes collections of data available to other
// Windows Runtime components.
auto results = ref new Vector<int>();
for (int prime : primes)
{
    results->Append(prime);
}
return results;
});
}
```

Each method first performs validation to ensure that the input parameters are non-negative. If an input value is negative, the method throws [Platform::InvalidArgumentException](#). Error handling is explained later in this section.

To consume these methods from a UWP app, use the Visual C# **Blank App (XAML)** template to add a second project to the Visual Studio solution. This example names the project `Primes`. Then, from the `Primes` project, add a reference to the `PrimesLibrary` project.

Add the following code to `MainPage.xaml`. This code defines the UI so that you can call the C++ component and display results.

```

<Page
  x:Class="Primes.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:Primes"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="300"/>
      <ColumnDefinition Width="300"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="125"/>
      <RowDefinition Height="125"/>
      <RowDefinition Height="125"/>
    </Grid.RowDefinitions>

    <StackPanel Grid.Column="0" Grid.Row="0">
      <Button Name="b1" Click="computePrimes">Compute Primes</Button>
      <TextBlock Name="tb1"></TextBlock>
    </StackPanel>

    <StackPanel Grid.Column="1" Grid.Row="0">
      <Button Name="b2" Click="computePrimesWithProgress">Compute Primes with Progress</Button>
      <ProgressBar Name="pb1" HorizontalAlignment="Left" Width="100"></ProgressBar>
      <TextBlock Name="tb2"></TextBlock>
    </StackPanel>

    <StackPanel Grid.Column="0" Grid.Row="1">
      <Button Name="b3" Click="getPrimes">Get Primes</Button>
      <TextBlock Name="tb3"></TextBlock>
    </StackPanel>

    <StackPanel Grid.Column="1" Grid.Row="1">
      <Button Name="b4" Click="getPrimesWithProgress">Get Primes with Progress</Button>
      <ProgressBar Name="pb4" HorizontalAlignment="Left" Width="100"></ProgressBar>
      <TextBlock Name="tb4"></TextBlock>
    </StackPanel>

    <StackPanel Grid.Column="0" Grid.Row="2">
      <Button Name="b5" Click="getPrimesHandleErrors">Get Primes and Handle Errors</Button>
      <ProgressBar Name="pb5" HorizontalAlignment="Left" Width="100"></ProgressBar>
      <TextBlock Name="tb5"></TextBlock>
    </StackPanel>

    <StackPanel Grid.Column="1" Grid.Row="2">
      <Button Name="b6" Click="getPrimesCancellation">Get Primes with Cancellation</Button>
      <Button Name="cancelButton" Click="cancelGetPrimes" IsEnabled="false">Cancel</Button>
      <ProgressBar Name="pb6" HorizontalAlignment="Left" Width="100"></ProgressBar>
      <TextBlock Name="tb6"></TextBlock>
    </StackPanel>
  </Grid>
</Page>

```

Add the following code to the `MainPage` class in `MainPage.xaml`. This code defines a `Primes` object and the button event handlers.

```

private PrimesLibrary.Primes primesLib = new PrimesLibrary.Primes();

private async void computePrimes(object sender, RoutedEventArgs e)
{
    b1.IsEnabled = false;
    th1.Text = "Working...":

```

```

        tb1.Text = "Working...";
    }

    var asyncAction = primesLib.ComputePrimesAsync(0, 100000);

    await asyncAction;

    tb1.Text = "Done";
    b1.IsEnabled = true;
}

private async void computePrimesWithProgress(object sender, RoutedEventArgs e)
{
    b2.IsEnabled = false;
    tb2.Text = "Working...";

    var asyncAction = primesLib.ComputePrimesWithProgressAsync(0, 100000);
    asyncAction.Progress = new AsyncActionProgressHandler<double>((action, progress) =>
    {
        pb1.Value = progress;
    });

    await asyncAction;

    tb2.Text = "Done";
    b2.IsEnabled = true;
}

private async void getPrimes(object sender, RoutedEventArgs e)
{
    b3.IsEnabled = false;
    tb3.Text = "Working...";

    var asyncOperation = primesLib.GetPrimesAsync(0, 100000);

    await asyncOperation;

    tb3.Text = "Found " + asyncOperation.GetResults().Count + " primes";
    b3.IsEnabled = true;
}

private async void getPrimesWithProgress(object sender, RoutedEventArgs e)
{
    b4.IsEnabled = false;
    tb4.Text = "Working...";

    var asyncOperation = primesLib.GetPrimesWithProgressAsync(0, 100000);
    asyncOperation.Progress = new AsyncOperationProgressHandler<IList<int>, double>((operation, progress) =>
    {
        pb4.Value = progress;
    });

    await asyncOperation;

    tb4.Text = "Found " + asyncOperation.GetResults().Count + " primes";
    b4.IsEnabled = true;
}

private async void getPrimesHandleErrors(object sender, RoutedEventArgs e)
{
    b5.IsEnabled = false;
    tb5.Text = "Working...";

    var asyncOperation = primesLib.GetPrimesWithProgressAsync(-1000, 100000);
    asyncOperation.Progress = new AsyncOperationProgressHandler<IList<int>, double>((operation, progress) =>
    {
        pb5.Value = progress;
    });

    try
    {

```

```

    {
        await asyncOperation;
        tb5.Text = "Found " + asyncOperation.GetResults().Count + " primes";
    }
    catch (ArgumentException ex)
    {
        tb5.Text = "ERROR: " + ex.Message;
    }

    b5.IsEnabled = true;
}

private IAsyncOperationWithProgress<IList<int>, double> asyncCancelableOperation;

private async void getPrimesCancellation(object sender, RoutedEventArgs e)
{
    b6.IsEnabled = false;
    cancelButton.IsEnabled = true;
    tb6.Text = "Working...";

    asyncCancelableOperation = primesLib.GetPrimesWithProgressAsync(0, 200000);
    asyncCancelableOperation.Progress = new AsyncOperationProgressHandler<IList<int>, double>((operation,
progress) =>
    {
        pb6.Value = progress;
    });

    try
    {
        await asyncCancelableOperation;
        tb6.Text = "Found " + asyncCancelableOperation.GetResults().Count + " primes";
    }
    catch (System.Threading.Tasks.TaskCanceledException)
    {
        tb6.Text = "Operation canceled";
    }

    b6.IsEnabled = true;
    cancelButton.IsEnabled = false;
}

private void cancelGetPrimes(object sender, RoutedEventArgs e)
{
    cancelButton.IsEnabled = false;
    asyncCancelableOperation.Cancel();
}

```

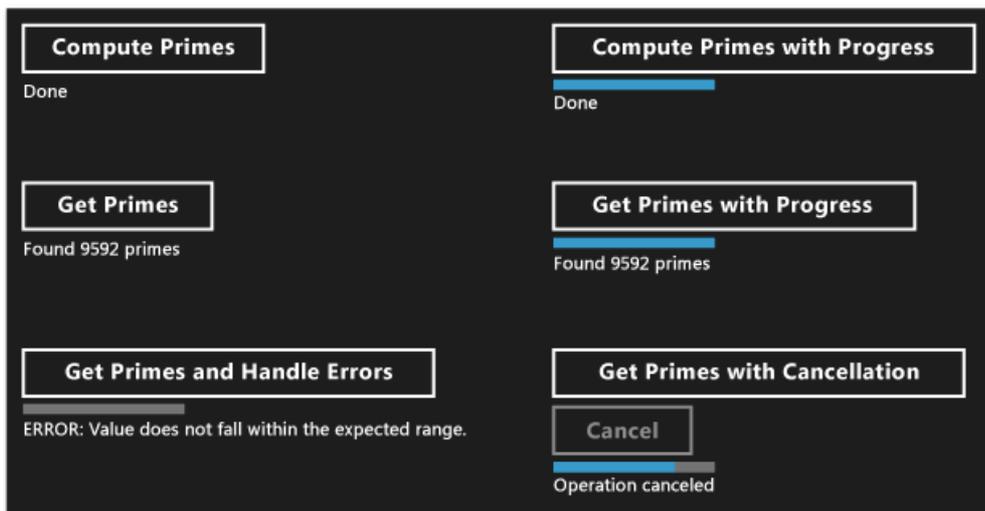
These methods use the `async` and `await` keywords to update the UI after the asynchronous operations complete. For information about asynchronous coding in UWP apps, see [Threading and async programming](#).

The `getPrimesCancellation` and `cancelGetPrimes` methods work together to enable the user to cancel the operation. When the user chooses the **Cancel** button, the `cancelGetPrimes` method calls `IAsyncOperationWithProgress<TResult, TProgress>::Cancel` to cancel the operation. The Concurrency Runtime, which manages the underlying asynchronous operation, throws an internal exception type that's caught by the Windows Runtime to communicate that cancellation has completed. For more information about the cancellation model, see [Cancellation](#).

#### IMPORTANT

To enable the PPL to correctly report to the Windows Runtime that it has canceled the operation, do not catch this internal exception type. This means that you should also not catch all exceptions (`catch (...)`). If you must catch all exceptions, rethrow the exception to ensure that the Windows Runtime can complete the cancellation operation.

The following illustration shows the `Primes` app after each option has been chosen.



For examples that use `create_async` to create asynchronous tasks that can be consumed by other languages, see [Using C++ in the Bing Maps Trip Optimizer sample](#) and [Windows 8 Asynchronous Operations in C++ with PPL](#).

## Controlling the Execution Thread

The Windows Runtime uses the COM threading model. In this model, objects are hosted in different apartments, depending on how they handle their synchronization. Thread-safe objects are hosted in the multi-threaded apartment (MTA). Objects that must be accessed by a single thread are hosted in a single-threaded apartment (STA).

In an app that has a UI, the ASTA (Application STA) thread is responsible for pumping window messages and is the only thread in the process that can update the STA-hosted UI controls. This has two consequences. First, to enable the app to remain responsive, all CPU-intensive and I/O operations should not be run on the ASTA thread. Second, results that come from background threads must be marshaled back to the ASTA to update the UI. In a C++ UWP app, `MainPage` and other XAML pages all run on the ATSA. Therefore, task continuations that are declared on the ASTA are run there by default so you can update controls directly in the continuation body. However, if you nest a task in another task, any continuations on that nested task run in the MTA. Therefore, you need to consider whether to explicitly specify on what context these continuations run.

A task that's created from an asynchronous operation, such as `IAsyncOperation<TResult>`, uses special semantics that can help you ignore the threading details. Although an operation might run on a background thread (or it may not be backed by a thread at all), its continuations are by default guaranteed to run on the apartment that started the continuation operations (in other words, from the apartment that called `task::then`). You can use the `concurrency::task_continuation_context` class to control the execution context of a continuation. Use these static helper methods to create `task_continuation_context` objects:

- Use `concurrency::task_continuation_context::use_arbitrary` to specify that the continuation runs on a background thread.
- Use `concurrency::task_continuation_context::use_current` to specify that the continuation runs on the thread that called `task::then`.

You can pass a `task_continuation_context` object to the `task::then` method to explicitly control the execution context of the continuation or you can pass the task to another apartment and then call the `task::then` method to implicitly control the execution context.

### IMPORTANT

Because the main UI thread of UWP apps run under STA, continuations that you create on that STA by default run on the STA. Accordingly, continuations that you create on the MTA run on the MTA.

The following section shows an app that reads a file from disk, finds the most common words in that file, and then shows the results in the UI. The final operation, updating the UI, occurs on the UI thread.

### IMPORTANT

This behavior is specific to UWP apps. For desktop apps, you do not control where continuations run. Instead, the scheduler chooses a worker thread on which to run each continuation.

### IMPORTANT

Do not call `concurrency::task::wait` in the body of a continuation that runs on the STA. Otherwise, the runtime throws `concurrency::invalid_operation` because this method blocks the current thread and can cause the app to become unresponsive. However, you can call the `concurrency::task::get` method to receive the result of the antecedent task in a task-based continuation.

## Example: Controlling Execution in a Windows Runtime App with C++ and XAML

Consider a C++ XAML app that reads a file from disk, finds the most common words in that file, and then shows the results in the UI. To create this app, start, in Visual Studio, by creating a **Blank App (Universal Windows)** project and naming it `CommonWords`. In your app manifest, specify the **Documents Library** capability to enable the app to access the Documents folder. Also add the Text (.txt) file type to the declarations section of the app manifest. For more information about app capabilities and declarations, see [App packages and deployment](#).

Update the `Grid` element in `MainPage.xaml` to include a `ProgressRing` element and a `TextBlock` element. The `ProgressRing` indicates that the operation is in progress and the `TextBlock` shows the results of the computation.

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <ProgressRing x:Name="Progress"/>
  <TextBlock x:Name="Results" FontSize="16"/>
</Grid>
```

Add the following `#include` statements to `pch.h`.

```
#include <sstream>
#include <ppltasks.h>
#include <concurrent_unordered_map.h>
```

Add the following method declarations to the `MainPage` class (`MainPage.h`).

```

private:
    // Splits the provided text string into individual words.
    concurrency::task<std::vector<std::wstring>> MakeWordList(Platform::String^ text);

    // Finds the most common words that are at least the provided minimum length.
    concurrency::task<std::vector<std::pair<std::wstring, size_t>>> FindCommonWords(const
std::vector<std::wstring>& words, size_t min_length, size_t count);

    // Shows the most common words on the UI.
    void ShowResults(const std::vector<std::pair<std::wstring, size_t>>& commonWords);

```

Add the following `using` statements to MainPage.cpp.

```

using namespace concurrency;
using namespace std;
using namespace Windows::Storage;
using namespace Windows::Storage::Streams;

```

In MainPage.cpp, implement the `MainPage::MakeWordList`, `MainPage::FindCommonWords`, and `MainPage::ShowResults` methods. The `MainPage::MakeWordList` and `MainPage::FindCommonWords` perform computationally-intensive operations. The `MainPage::ShowResults` method displays the result of the computation in the UI.

```

// Splits the provided text string into individual words.
task<vector<wstring>> MainPage::MakeWordList(String^ text)
{
    return create_task([text]() -> vector<wstring>
    {
        vector<wstring> words;

        // Add continuous sequences of alphanumeric characters to the string vector.
        wstring current_word;
        for (wchar_t ch : text)
        {
            if (!iswalnum(ch))
            {
                if (current_word.length() > 0)
                {
                    words.push_back(current_word);
                    current_word.clear();
                }
            }
            else
            {
                current_word += ch;
            }
        }

        return words;
    });
}

// Finds the most common words that are at least the provided minimum length.
task<vector<pair<wstring, size_t>>> MainPage::FindCommonWords(const vector<wstring>& words, size_t min_length,
size_t count)
{
    return create_task([words, min_length, count]() -> vector<pair<wstring, size_t>>
    {
        typedef pair<wstring, size_t> pair;

        // Counts the occurrences of each word.
        concurrent_unordered_map<wstring, size_t> counts;

        parallel_for_each(begin(words), end(words), [&counts, min_length](const wstring& word)

```

```

    {
        // Increment the count of words that are at least the minimum length.
        if (word.length() >= min_length)
        {
            // Increment the count.
            InterlockedIncrement(&counts[word]);
        }
    });

    // Copy the contents of the map to a vector and sort the vector by the number of occurrences of each
word.
    vector<pair> wordvector;
    copy(begin(counts), end(counts), back_inserter(wordvector));

    sort(begin(wordvector), end(wordvector), [](const pair& x, const pair& y)
    {
        return x.second > y.second;
    });

    size_t size = min(wordvector.size(), count);
    wordvector.erase(begin(wordvector) + size, end(wordvector));

    return wordvector;
});
}

// Shows the most common words on the UI.
void MainPage::ShowResults(const vector<pair<wstring, size_t>>& commonWords)
{
    wstringstream ss;
    ss << "The most common words that have five or more letters are:";
    for (auto commonWord : commonWords)
    {
        ss << endl << commonWord.first << L" (" << commonWord.second << L')';
    }

    // Update the UI.
    Results->Text = ref new String(ss.str().c_str());
}

```

Modify the `MainPage` constructor to create a chain of continuation tasks that displays in the UI the common words in the book *The Iliad* by Homer. The first two continuation tasks, which split the text into individual words and find common words, can be time consuming and are therefore explicitly set to run in the background. The final continuation task, which updates the UI, specifies no continuation context, and therefore follows the apartment threading rules.

```

MainPage::MainPage()
{
    InitializeComponent();

    // To run this example, save the contents of http://www.gutenberg.org/files/6130/6130-0.txt to your
    Documents folder.
    // Name the file "The Iliad.txt" and save it under UTF-8 encoding.

    // Enable the progress ring.
    Progress->IsActive = true;

    // Find the most common words in the book "The Iliad".

    // Get the file.
    create_task(KnownFolders::DocumentsLibrary->GetFileAsync("The Iliad.txt")).then([](StorageFile^ file)
    {
        // Read the file text.
        return FileIO::ReadTextAsync(file, UnicodeEncoding::Utf8);

        // By default, all continuations from a Windows Runtime async operation run on the
        // thread that calls task.then. Specify use_arbitrary to run this continuation
        // on a background thread.
    }, task_continuation_context::use_arbitrary()).then([this](String^ file)
    {
        // Create a word list from the text.
        return MakeWordList(file);

        // By default, all continuations from a Windows Runtime async operation run on the
        // thread that calls task.then. Specify use_arbitrary to run this continuation
        // on a background thread.
    }, task_continuation_context::use_arbitrary()).then([this](vector<wstring> words)
    {
        // Find the most common words.
        return FindCommonWords(words, 5, 9);

        // By default, all continuations from a Windows Runtime async operation run on the
        // thread that calls task.then. Specify use_arbitrary to run this continuation
        // on a background thread.
    }, task_continuation_context::use_arbitrary()).then([this](vector<pair<wstring, size_t>> commonWords)
    {
        // Stop the progress ring.
        Progress->IsActive = false;

        // Show the results.
        ShowResults(commonWords);

        // We don't specify a continuation context here because we want the continuation
        // to run on the STA thread.
    });
}

```

#### NOTE

This example demonstrates how to specify execution contexts and how to compose a chain of continuations. Recall that by default a task that's created from an asynchronous operation runs its continuations on the apartment that called `task::then`. Therefore, this example uses `task_continuation_context::use_arbitrary` to specify that operations that do not involve the UI be performed on a background thread.

The following illustration shows the results of the `CommonWords` app.

```
The most common words that have five or more letters are:
```

```
their (954)  
shall (441)  
Hector (433)  
which (425)  
great (398)  
Achilles (386)  
through (301)  
these (266)  
chief (264)
```

In this example, it's possible to support cancellation because the `task` objects that support `create_async` use an implicit cancellation token. Define your work function to take a `cancellation_token` object if your tasks need to respond to cancellation in a cooperative manner. For more info about cancellation in the PPL, see [Cancellation in the PPL](#)

## See also

[Concurrency Runtime](#)

# Comparing the Concurrency Runtime to Other Concurrency Models

3/4/2019 • 8 minutes to read • [Edit Online](#)

This document describes the differences between the features and programming models of the Concurrency Runtime and other technologies. By understanding how the benefits of the Concurrency Runtime compare to the benefits of other programming models, you can select the technology that best satisfies the requirements of your applications.

If you are currently using another programming model, such as the Windows thread pool or OpenMP, there are situations where it can be appropriate to migrate to the Concurrency Runtime. For example, the topic [Migrating from OpenMP to the Concurrency Runtime](#) describes when it can be appropriate to migrate from OpenMP to the Concurrency Runtime. However, if you are satisfied with application performance and current debugging support, migration is not required.

You can use the features and productivity benefits of the Concurrency Runtime to complement your existing application that uses another concurrency model. The Concurrency Runtime cannot guarantee load balancing when multiple task schedulers compete for the same computing resources. However, when workloads do not overlap, this effect is minimal.

## Sections

- [Comparing Preemptive Scheduling to Cooperative Scheduling](#)
- [Comparing the Concurrency Runtime to the Windows API](#)
- [Comparing the Concurrency Runtime to OpenMP](#)

## Comparing Preemptive Scheduling to Cooperative Scheduling

The preemptive model and cooperative scheduling models are two common ways to enable multiple tasks to share computing resources, for example, processors or hardware threads.

### Preemptive and Cooperative Scheduling

*Preemptive scheduling* is a round-robin, priority-based mechanism that gives every task exclusive access to a computing resource for a given time period, and then switches to another task. Preemptive scheduling is common in multitasking operating systems such as Windows. *Cooperative scheduling* is a mechanism that gives every task exclusive access to a computing resource until the task finishes or until the task yields its access to the resource. The Concurrency Runtime uses cooperative scheduling together with the preemptive scheduler of the operating system to achieve maximum usage of processing resources.

### Differences Between Preemptive and Cooperative Schedulers

Preemptive schedulers seek to give multiple threads equal access to computing resources to ensure that every thread makes progress. On computers that have many computing resources, ensuring fair access becomes less problematic; however, ensuring efficient utilization of the resources becomes more problematic.

A preemptive kernel-mode scheduler requires the application code to rely on the operating system to make scheduling decisions. Conversely, a user-mode cooperative scheduler enables application code to make its own scheduling decisions. Because cooperative scheduling enables many scheduling decisions to be made by the application, it reduces much of the overhead that is associated with kernel-mode synchronization. A cooperative scheduler typically defers scheduling decisions to the operating system kernel when it has no other work to

schedule. A cooperative scheduler also defers to the operating system scheduler when there is a blocking operation that is communicated to the kernel, but that operation is not communicated to the user-mode scheduler.

### **Cooperative Scheduling and Efficiency**

To a preemptive scheduler, all work that has the same priority level is equal. A preemptive scheduler typically schedules threads in the order in which they are created. Furthermore, a preemptive scheduler gives every thread a time slice in a round-robin manner, based on thread priority. Although this mechanism provides fairness (every thread makes forward progress), it comes at some cost of efficiency. For example, many computation-intensive algorithms do not require fairness. Instead, it is important that related tasks finish in the least overall time. Cooperative scheduling enables an application to more efficiently schedule work. For example, consider an application that has many threads. Scheduling threads that do not share resources to run concurrently can reduce synchronization overhead and thereby increase efficiency. Another efficient way to schedule tasks is to run pipelines of tasks (where each task acts on the output of the previous one) on the same processor so that the input of each pipeline stage is already loaded into the memory cache.

### **Using Preemptive and Cooperative Scheduling Together**

Cooperative scheduling does not solve all scheduling problems. For example, tasks that do not fairly yield to other tasks can consume all available computing resources and prevent other tasks from making progress. The Concurrency Runtime uses the efficiency benefits of cooperative scheduling to complement the fairness guarantees of preemptive scheduling. By default, the Concurrency Runtime provides a cooperative scheduler that uses a work-stealing algorithm to efficiently distribute work among computing resources. However, the Concurrency Runtime scheduler also relies on the preemptive scheduler of the operating system to fairly distribute resources among applications. You can also create custom schedulers and scheduler policies in your applications to produce fine-grained control over thread execution.

[\[Top\]](#)

## **Comparing the Concurrency Runtime to the Windows API**

The Microsoft Windows application programming interface, which is typically referred to as the Windows API (and formerly known as Win32), provides a programming model that enables concurrency in your applications. The Concurrency Runtime builds on the Windows API to provide additional programming models that are not available from the underlying operating system.

The Concurrency Runtime builds on the Windows API thread model to perform parallel work. It also uses the Windows API memory management and thread-local storage mechanisms. On Windows 7 and Windows Server 2008 R2, it uses Windows API support for user-schedulable threads and computers that have more than 64 hardware threads. The Concurrency Runtime extends the Windows API model by providing a cooperative task scheduler and a work-stealing algorithm to maximize the use of computing resources, and by enabling multiple simultaneous scheduler instances.

### **Programming Languages**

The Windows API uses the C programming language to expose the programming model. The Concurrency Runtime provides a C++ programming interface that takes advantage of the newest features in the C++ language. For example, lambda functions provide a succinct, type-safe mechanism for defining parallel work functions. For more information about the newest C++ features that the Concurrency Runtime uses, see [Overview](#).

### **Threads and Thread Pools**

The central concurrency mechanism in the Windows API is the thread. You typically use the [CreateThread](#) function to create threads. Although threads are relatively easy to create and use, the operating system allocates a significant amount of time and other resources to manage them. Additionally, although each thread is guaranteed to receive the same execution time as any other thread at the same priority level, the associated overhead requires that you create sufficiently large tasks. For smaller or more fine-grained tasks, the overhead that is associated with concurrency can outweigh the benefit of running the tasks in parallel.

Thread pools are one way to reduce the cost of thread management. Custom thread pools and the thread pool implementation that is provided by the Windows API both enable small work items to efficiently run in parallel. The Windows thread pool maintains work items in a first-in, first-out (FIFO) queue. Each work item is started in the order in which it was added to the pool.

The Concurrency Runtime implements a work-stealing algorithm to extend the FIFO scheduling mechanism. The algorithm moves tasks that have not yet started to threads that run out of work items. Although the work-stealing algorithm can balance workloads, it can also cause work items to be reordered. This reordering process can cause a work item to start in a different order than it was submitted. This is useful with recursive algorithms, where there is a better chance that data is shared among newer tasks than among older ones. Getting the new items to run first means fewer cache misses and possibly fewer page faults.

From the perspective of the operating system, work stealing is unfair. However, when an application implements an algorithm or task to run in parallel, fairness among the sub-tasks does not always matter. What does matter is how quickly the overall task finishes. For other algorithms, FIFO is the appropriate scheduling strategy.

### **Behavior on Various Operating Systems**

On Windows XP and Windows Vista, applications that use the Concurrency Runtime behave similarly, except that heap performance is improved on Windows Vista.

In Windows 7 and Windows Server 2008 R2, the operating system further supports concurrency and scalability. For example, these operating systems support computers that have more than 64 hardware threads. An existing application that uses the Windows API must be modified to take advantage of these new features. However, an application that uses the Concurrency Runtime automatically uses these features and does not require modifications.

[base.user-mode\\_scheduling](#)

[\[Top\]](#)

## Comparing the Concurrency Runtime to OpenMP

The Concurrency Runtime enables a variety of programming models. These models may overlap or complement the models of other libraries. This section compares the Concurrency Runtime to [OpenMP](#).

The OpenMP programming model is defined by an open standard and has well-defined bindings to the Fortran and C/C++ programming languages. OpenMP versions 2.0 and 2.5 are well-suited for parallel algorithms that are iterative; that is, they perform parallel iteration over an array of data. OpenMP is most efficient when the degree of parallelism is pre-determined and matches the available resources on the system. The OpenMP model is an especially good match for high-performance computing, where very large computational problems are distributed across the processing resources of a single computer. In this scenario, the hardware environment is known and the developer can reasonably expect to have exclusive access to computing resources when the algorithm is executed.

However, other, less constrained computing environments may not be a good match for OpenMP. For example, recursive problems (such as the quicksort algorithm or searching a tree of data) are more difficult to implement by using OpenMP. The Concurrency Runtime complements the capabilities of OpenMP by providing the [Parallel Patterns Library](#) (PPL) and the [Asynchronous Agents Library](#). Unlike OpenMP, the Concurrency Runtime provides a dynamic scheduler that adapts to available resources and adjusts the degree of parallelism as workloads change.

Many of the features in the Concurrency Runtime can be extended. You can also combine existing features to compose new ones. Because OpenMP relies on compiler directives, it cannot be extended easily.

For more information about how the Concurrency Runtime compares to OpenMP and how to migrate existing OpenMP code to use the Concurrency Runtime, see [Migrating from OpenMP to the Concurrency Runtime](#).

[\[Top\]](#)

## See also

[Concurrency Runtime](#)

[Overview](#)

[Parallel Patterns Library \(PPL\)](#)

[Asynchronous Agents Library](#)

[OpenMP](#)

# Migrating from OpenMP to the Concurrency Runtime

5/8/2019 • 4 minutes to read • [Edit Online](#)

The Concurrency Runtime enables a variety of programming models. These models may overlap or complement the models of other libraries. The documents in this section compare [OpenMP](#) to the Concurrency Runtime and provide examples about how to migrate existing OpenMP code to use the Concurrency Runtime.

The OpenMP programming model is defined by an open standard and has well-defined bindings to the Fortran and C/C++ programming languages. OpenMP versions 2.0 and 2.5, which are supported by the Microsoft C++ compiler, are well-suited for parallel algorithms that are iterative; that is, they perform parallel iteration over an array of data. OpenMP 3.0 supports non-iterative tasks in addition to iterative tasks.

OpenMP is most efficient when the degree of parallelism is pre-determined and matches the available resources on the system. The OpenMP model is an especially good match for high-performance computing, where very large computational problems are distributed across the processing resources of one computer. In this scenario, the hardware environment is generally fixed and the developer can reasonably expect to have exclusive access to all computing resources when the algorithm is executed.

However, less constrained computing environments may not be a good match for OpenMP. For example, recursive problems (such as the quicksort algorithm or searching a tree of data) are more difficult to implement by using OpenMP 2.0 and 2.5. The Concurrency Runtime complements the capabilities of OpenMP by providing the [Asynchronous Agents Library](#) and the [Parallel Patterns Library](#) (PPL). The Asynchronous Agents Library supports coarse-grained task parallelism; the PPL supports more fine-grained parallel tasks. The Concurrency Runtime provides the infrastructure that is required to perform operations in parallel so that you can focus on the logic of your application. However, because the Concurrency Runtime enables a variety of programming models, its scheduling overhead can be greater than other concurrency libraries such as OpenMP. Therefore, we recommend that you test performance incrementally when you convert your existing OpenMP code to use the Concurrency Runtime.

## When to Migrate from OpenMP to the Concurrency Runtime

It may be advantageous to migrate existing OpenMP code to use the Concurrency Runtime in the following cases.

CASES	ADVANTAGES OF THE CONCURRENCY RUNTIME
You require an extensible concurrent programming framework.	Many of the features in the Concurrency Runtime can be extended. You can also combine existing features to compose new ones. Because OpenMP relies on compiler directives, it cannot be easily extended.
Your application would benefit from cooperative blocking.	When a task blocks because it requires a resource that is not yet available, the Concurrency Runtime can perform other tasks while the first task waits for the resource.
Your application would benefit from dynamic load balancing.	The Concurrency Runtime uses a scheduling algorithm that adjusts the allocation of computing resources as workloads change. In OpenMP, when the scheduler allocates computing resources to a parallel region, those resource allocations are fixed throughout the computation.

CASES	ADVANTAGES OF THE CONCURRENCY RUNTIME
You require exception handling support.	The PPL lets you catch exceptions both inside and outside of a parallel region or loop. In OpenMP, you must handle the exception inside of the parallel region or loop.
You require a cancellation mechanism.	The PPL enables applications to cancel both individual tasks and parallel trees of work. OpenMP requires the application to implement its own cancellation mechanism.
You require parallel code to finish in a different context from which it starts.	The Concurrency Runtime lets you start a task in one context, and then wait on or cancel that task in another context. In OpenMP, all parallel work must finish in the context from which it starts.
You require enhanced debugging support.	Visual Studio provides the <b>Parallel Stacks</b> and <b>Parallel Tasks</b> windows so that you can more easily debug multithreaded applications.  For more information about debugging support for the Concurrency Runtime, see <a href="#">Using the Tasks Window</a> , <a href="#">Using the Parallel Stacks Window</a> , and <a href="#">Walkthrough: Debugging a Parallel Application</a> .

## When Not to Migrate from OpenMP to the Concurrency Runtime

The following cases describe when it might not be appropriate to migrate existing OpenMP code to use the Concurrency Runtime.

CASES	EXPLANATION
Your application already meets your requirements.	If you are satisfied with application performance and current debugging support, migration might not be appropriate.
Your parallel loop bodies perform little work.	The overhead of the Concurrency Runtime task scheduler might not overcome the benefits of executing the loop body in parallel, especially when the loop body is relatively small.
Your application is written in C.	Because the Concurrency Runtime uses many C++ features, it might not be suitable when you cannot write code that enables the C application to fully use it.

## Related Topics

[How to: Convert an OpenMP parallel for Loop to Use the Concurrency Runtime](#)

Given a basic loop that uses the OpenMP `parallel` and `for` directives, demonstrates how to convert it to use the Concurrency Runtime `concurrency::parallel_for` algorithm.

[How to: Convert an OpenMP Loop that Uses Cancellation to Use the Concurrency Runtime](#)

Given an OpenMP `parallelfor` loop that does not require all iterations to run, demonstrates how to convert it to use the Concurrency Runtime cancellation mechanism.

[How to: Convert an OpenMP Loop that Uses Exception Handling to Use the Concurrency Runtime](#)

Given an OpenMP `parallelfor` loop that performs exception handling, demonstrates how to convert it to use the Concurrency Runtime exception handling mechanism.

[How to: Convert an OpenMP Loop that Uses a Reduction Variable to Use the Concurrency Runtime](#)

Given an OpenMP [parallelfor](#) loop that uses the [reduction](#) clause, demonstrates how to convert it to use the Concurrency Runtime.

## See also

[Concurrency Runtime](#)

[OpenMP](#)

[Parallel Patterns Library \(PPL\)](#)

[Asynchronous Agents Library](#)

# How to: Convert an OpenMP parallel for Loop to Use the Concurrency Runtime

3/4/2019 • 3 minutes to read • [Edit Online](#)

This example demonstrates how to convert a basic loop that uses the OpenMP `parallel` and `for` directives to use the Concurrency Runtime `concurrency::parallel_for` algorithm.

## Example

This example uses both OpenMP and the Concurrency Runtime to compute the count of prime numbers in an array of random values.

```
// conrct-omp-count-primes.cpp
// compile with: /EHsc /openmp
#include <ppl.h>
#include <random>
#include <array>
#include <iostream>

using namespace concurrency;
using namespace std;

// Determines whether the input value is prime.
bool is_prime(int n)
{
    if (n < 2)
        return false;
    for (int i = 2; i < n; ++i)
    {
        if ((n % i) == 0)
            return false;
    }
    return true;
}

// Uses OpenMP to compute the count of prime numbers in an array.
void omp_count_primes(int* a, size_t size)
{
    if (size == 0)
        return;

    size_t count = 0;
    #pragma omp parallel for
    for (int i = 0; i < static_cast<int>(size); ++i)
    {
        if (is_prime(a[i])) {
            #pragma omp atomic
            ++count;
        }
    }

    wcout << L"found " << count
           << L" prime numbers." << endl;
}

// Uses the Concurrency Runtime to compute the count of prime numbers in an array.
void conrct_count_primes(int* a, size_t size)
{
    if (size == 0)
```

```

    return;

    combinable<size_t> counts;
    parallel_for<size_t>(0, size, [&](size_t i)
    {
        if (is_prime(a[i])) {
            counts.local()++;
        }
    });

    wcout << L"found " << counts.combine(plus<size_t>())
          << L" prime numbers." << endl;
}

int wmain()
{
    // The length of the array.
    const size_t size = 1000000;

    // Create an array and initialize it with random values.
    int* a = new int[size];

    mt19937 gen(42);
    for (size_t i = 0; i < size; ++i) {
        a[i] = gen();
    }

    // Count prime numbers by using OpenMP and the Concurrency Runtime.

    wcout << L"Using OpenMP..." << endl;
    omp_count_primes(a, size);

    wcout << L"Using the Concurrency Runtime..." << endl;
    concrt_count_primes(a, size);

    delete[] a;
}

```

This example produces the following output.

```

Using OpenMP...
found 107254 prime numbers.
Using the Concurrency Runtime...
found 107254 prime numbers.

```

The `parallel_for` algorithm and OpenMP 3.0 allow for the index type to be a signed integral type or an unsigned integral type. The `parallel_for` algorithm also makes sure that the specified range does not overflow a signed type. OpenMP versions 2.0 and 2.5 allow for signed integral index types only. OpenMP also does not validate the index range.

The version of this example that uses the Concurrency Runtime also uses a `concurrency::combinable` object in place of the `atomic` directive to increment the counter value without requiring synchronization.

For more information about `parallel_for` and other parallel algorithms, see [Parallel Algorithms](#). For more information about the `combinable` class, see [Parallel Containers and Objects](#).

## Example

This example modifies the previous one to act on an `std::array` object instead of on a native array. Because OpenMP versions 2.0 and 2.5 allow for signed integral index types only in a `parallel_for` construct, you cannot use iterators to access the elements of a C++ Standard Library container in parallel. The Parallel Patterns Library (PPL) provides the `concurrency::parallel_for_each` algorithm, which performs tasks, in parallel, on an iterative

container such as those provided by the C++ Standard Library. It uses the same partitioning logic that the `parallel_for` algorithm uses. The `parallel_for_each` algorithm resembles the C++ Standard Library `std::for_each` algorithm, except that the `parallel_for_each` algorithm executes the tasks concurrently.

```
// Uses OpenMP to compute the count of prime numbers in an
// array object.
template<size_t Size>
void omp_count_primes(const array<int, Size>& a)
{
    if (a.size() == 0)
        return;

    size_t count = 0;
    int size = static_cast<int>(a.size());
    #pragma omp parallel for
        for (int i = 0; i < size; ++i)
        {
            if (is_prime(a[i])) {
                #pragma omp atomic
                ++count;
            }
        }

    wcout << L"found " << count
          << L" prime numbers." << endl;
}

// Uses the Concurrency Runtime to compute the count of prime numbers in an
// array object.
template<size_t Size>
void conrcrt_count_primes(const array<int, Size>& a)
{
    if (a.size() == 0)
        return;

    combinable<size_t> counts;
    parallel_for_each(begin(a), end(a), [&counts](int n)
    {
        if (is_prime(n)) {
            counts.local()++;
        }
    });

    wcout << L"found " << counts.combine(plus<size_t>())
          << L" prime numbers." << endl;
}
```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `conrcrt-omp-count-primes.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc /openmp conrcrt-omp-count-primes.cpp
```

## See also

[Migrating from OpenMP to the Concurrency Runtime](#)

[Parallel Algorithms](#)

[Parallel Containers and Objects](#)

# How to: Convert an OpenMP Loop that Uses Cancellation to Use the Concurrency Runtime

3/4/2019 • 4 minutes to read • [Edit Online](#)

Some parallel loops do not require that all iterations be executed. For example, an algorithm that searches for a value can terminate after the value is found. OpenMP does not provide a mechanism to break out of a parallel loop. However, you can use a Boolean value, or flag, to enable an iteration of the loop to indicate that the solution has been found. The Concurrency Runtime provides functionality that enables one task to cancel other tasks that have not yet started.

This example demonstrates how to convert an OpenMP `parallelfor` loop that does not require for all iterations to run to use the Concurrency Runtime cancellation mechanism.

## Example

This example uses both OpenMP and the Concurrency Runtime to implement a parallel version of the `std::any_of` algorithm. The OpenMP version of this example uses a flag to coordinate among all parallel loop iterations that the condition has been met. The version that uses the Concurrency Runtime uses the `concurrency::structured_task_group::cancel` method to stop the overall operation when the condition is met.

```
// conrct-omp-parallel-any-of.cpp
// compile with: /EHsc /openmp
#include <ppl.h>
#include <array>
#include <random>
#include <iostream>

using namespace concurrency;
using namespace std;

// Uses OpenMP to determine whether a condition exists in
// the specified range of elements.
template <class InIt, class Predicate>
bool omp_parallel_any_of(InIt first, InIt last, const Predicate& pr)
{
    typedef typename std::iterator_traits<InIt>::value_type item_type;

    // A flag that indicates that the condition exists.
    bool found = false;

    #pragma omp parallel for
    for (int i = 0; i < static_cast<int>(last-first); ++i)
    {
        if (!found)
        {
            item_type& cur = *(first + i);

            // If the element satisfies the condition, set the flag to
            // cancel the operation.
            if (pr(cur)) {
                found = true;
            }
        }
    }

    return found;
}
```

```

// Uses the Concurrency Runtime to determine whether a condition exists in
// the specified range of elements.
template <class InIt, class Predicate>
bool concrt_parallel_any_of(InIt first, InIt last, const Predicate& pr)
{
    typedef typename std::iterator_traits<InIt>::value_type item_type;

    structured_task_group tasks;

    // Create a predicate function that cancels the task group when
    // an element satisfies the condition.
    auto for_each_predicate = [&pr, &tasks](const item_type& cur) {
        if (pr(cur)) {
            tasks.cancel();
        }
    };

    // Create a task that calls the predicate function in parallel on each
    // element in the range.
    auto task = make_task([&](){
        parallel_for_each(first, last, for_each_predicate);
    });

    // The condition is satisfied if the task group is in the cancelled state.
    return tasks.run_and_wait(task) == canceled;
}

int wmain()
{
    // The length of the array.
    const size_t size = 100000;

    // Create an array and initialize it with random values.
    array<int, size> a;
    generate(begin(a), end(a), mt19937(42));

    // Search for a value in the array by using OpenMP and the Concurrency Runtime.

    const int what = 9114046;
    auto predicate = [what](int n) -> bool {
        return (n == what);
    };

    wcout << L"Using OpenMP..." << endl;
    if (omp_parallel_any_of(begin(a), end(a), predicate))
    {
        wcout << what << L" is in the array." << endl;
    }
    else
    {
        wcout << what << L" is not in the array." << endl;
    }

    wcout << L"Using the Concurrency Runtime..." << endl;
    if (concr_parallel_any_of(begin(a), end(a), predicate))
    {
        wcout << what << L" is in the array." << endl;
    }
    else
    {
        wcout << what << L" is not in the array." << endl;
    }
}

```

This example produces the following output.

```
Using OpenMP...
9114046 is in the array.
Using the Concurrency Runtime...
9114046 is in the array.
```

In the version of that uses OpenMP, all iterations of the loop execute, even when the flag is set. Furthermore, if a task were to have any child tasks, the flag would also have to be available to those child tasks to communicate cancellation. In the Concurrency Runtime, when a task group is cancelled, the runtime cancels the entire tree of work, including child tasks. The `concurrency::parallel_for_each` algorithm uses tasks to perform work. Therefore, when one iteration of the loop cancels the root task, the entire tree of computation is also cancelled. When a tree of work is cancelled, the runtime does not start new tasks. However, the runtime allows tasks that have already started to finish. Therefore, in the case of the `parallel_for_each` algorithm, active loop iterations can clean up their resources.

In both versions of this example, if the array contains more than one copy of the value to search for, multiple loop iterations can each simultaneously set the result and cancel the overall operation. You can use a synchronization primitive, such as a critical section, if your problem requires that only one task performs work when a condition is met.

You can also use exception handling to cancel tasks that use the PPL. For more information about cancellation, see [Cancellation in the PPL](#).

For more information about `parallel_for_each` and other parallel algorithms, see [Parallel Algorithms](#).

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `concr-omp-parallel-any-of.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc /openmp concr-omp-parallel-any-of.cpp
```

## See also

[Migrating from OpenMP to the Concurrency Runtime](#)

[Cancellation in the PPL](#)

[Parallel Algorithms](#)

# How to: Convert an OpenMP Loop that Uses Exception Handling to Use the Concurrency Runtime

3/4/2019 • 4 minutes to read • [Edit Online](#)

This example demonstrates how to convert an OpenMP `parallelfor` loop that performs exception handling to use the Concurrency Runtime exception handling mechanism.

In OpenMP, an exception that is thrown in a parallel region must be caught and handled in the same region by the same thread. An exception that escapes the parallel region is caught by the unhandled exception handler, which terminates the process by default.

In the Concurrency Runtime, when you throw an exception in the body of a work function that you pass to a task group such as a `concurrency::task_group` or `concurrency::structured_task_group` object, or to a parallel algorithm such as `concurrency::parallel_for`, the runtime stores that exception and marshals it to the context that waits for the task group or algorithm to finish. For task groups, the waiting context is the context that calls `concurrency::task_group::wait`, `concurrency::structured_task_group::wait`, `concurrency::task_group::run_and_wait`, or `concurrency::structured_task_group::run_and_wait`. For a parallel algorithm, the waiting context is the context that called that algorithm. The runtime also stops all active tasks that are in the task group, including those in child task groups, and discards any tasks that have not yet started.

## Example

This example demonstrates how to handle exceptions in an OpenMP `parallel` region and in a call to `parallel_for`. The `do_work` function performs a memory allocation request that does not succeed and therefore throws an exception of type `std::bad_alloc`. In the version that uses OpenMP, the thread that throws the exception must also catch it. In other words, each iteration of an OpenMP parallel loop must handle the exception. In the version that uses the Concurrency Runtime, the main thread catches an exception that is thrown by another thread.

```
// conrct-omp-exceptions.cpp
// compile with: /EHsc /openmp
#include <ppl.h>
#include <new>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

// Demonstrates a function that performs a memory allocation request
// that does not succeed.
void do_work(int)
{
    // The following memory allocation causes this function to
    // throw std::bad_alloc.
    char* ptr = new char[(~unsigned int((int)0)/2) - 1];

    // TODO: Assuming that the allocation succeeds, perform some work
    // and free the allocated memory.

    delete[] ptr;
}

// Demonstrates an OpenMP parallel loop that performs exception handling.
void omp_exception_handling()
{
```

```

#pragma omp parallel for
  for(int i = 0; i < 10; i++)
  {
    try {
      // Perform a unit of work.
      do_work(i);
    }
    catch (exception const& e) {
      // Print the error to the console.
      wstringstream ss;
      ss << L"An error of type '" << typeid(e).name()
        << L"' occurred." << endl;
      wcout << ss.str();
    }
  }
}

// Demonstrates an Concurrency Runtime parallel loop that performs exception handling.
void concrt_exception_handling()
{
  try {
    parallel_for(0, 10, [](int i)
    {
      // Perform a unit of work.
      do_work(i);
    });
  }
  catch (exception const& e) {
    // Print the error to the console.
    wcout << L"An error of type '" << typeid(e).name()
      << L"' occurred." << endl;
  }
}

int wmain()
{
  wcout << L"Using OpenMP..." << endl;
  omp_exception_handling();

  wcout << L"Using the Concurrency Runtime..." << endl;
  concrt_exception_handling();
}

```

This example produces the following output.

```

Using OpenMP...
An error of type 'class std::bad_alloc' occurred.
Using the Concurrency Runtime...
An error of type 'class std::bad_alloc' occurred.

```

In the version of this example that uses OpenMP, the exception occurs in and is handled by each loop iteration. In the version that uses the Concurrency Runtime, the runtime stores the exception, stops all active tasks, discards any tasks that have not yet started, and marshals the exception to the context that calls `parallel_for`.

If you require that the version that uses OpenMP terminates after the exception occurs, you could use a Boolean flag to signal to other loop iterations that the error occurred. As in the example in the topic [How to: Convert an](#)

[OpenMP Loop that Uses Cancellation to Use the Concurrency Runtime](#), subsequent loop iterations would do nothing if the flag is set. Conversely, if you require that the loop that uses the Concurrency Runtime continues after the exception occurs, handle the exception in the parallel loop body itself.

Other components of the Concurrency Runtime, such as asynchronous agents and lightweight tasks, do not transport exceptions. Instead, unhandled exceptions are caught by the unhandled exception handler, which terminates the process by default. For more information about exception handling, see [Exception Handling](#).

For more information about `parallel_for` and other parallel algorithms, see [Parallel Algorithms](#).

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `concr-omp-exceptions.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc /openmp concr-omp-exceptions.cpp
```

## See also

[Migrating from OpenMP to the Concurrency Runtime](#)

[Exception Handling](#)

[Parallel Algorithms](#)

# How to: Convert an OpenMP Loop that Uses a Reduction Variable to Use the Concurrency Runtime

3/4/2019 • 2 minutes to read • [Edit Online](#)

This example demonstrates how to convert an OpenMP `paralelfor` loop that uses the `reduction` clause to use the Concurrency Runtime.

The OpenMP `reduction` clause lets you specify one or more thread-private variables that are subject to a reduction operation at the end of the parallel region. OpenMP predefines a set of reduction operators. Each reduction variable must be a scalar (for example, `int`, `long`, and `float`). OpenMP also defines several restrictions on how reduction variables are used in a parallel region.

The Parallel Patterns Library (PPL) provides the `concurrency::combinable` class, which provides reusable, thread-local storage that lets you perform fine-grained computations and then merge those computations into a final result. The `combinable` class is a template that acts on both scalar and complex types. To use the `combinable` class, perform sub-computations in the body of a parallel construct and then call the `concurrency::combinable::combine` or `concurrency::combinable::combine_each` method to produce the final result. The `combine` and `combine_each` methods each take a *combine function* that specifies how to combine each pair of elements. Therefore, the `combinable` class is not restricted to a fixed set of reduction operators.

## Example

This example uses both OpenMP and the Concurrency Runtime to compute the sum of the first 35 Fibonacci numbers.

```

// concrct-omp-fibonacci-reduction.cpp
// compile with: /EHsc /openmp
#include <pp1.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Computes the nth Fibonacci number.
// This function illustrates a lengthy operation and is therefore
// not optimized for performance.
int fibonacci(int n)
{
    if (n < 2)
        return n;

    // Compute the components in parallel.
    int n1, n2;
    parallel_invoke(
        [n,&n1] { n1 = fibonacci(n-1); },
        [n,&n2] { n2 = fibonacci(n-2); }
    );

    return n1 + n2;
}

// Uses OpenMP to compute the sum of Fibonacci numbers in parallel.
void omp_parallel_fibonacci_sum(int count)
{
    int sum = 0;
    #pragma omp parallel for reduction(+ : sum)
    for (int i = 0; i < count; ++i)
    {
        sum += fibonacci(i);
    }

    wcout << L"The sum of the first " << count << L" Fibonacci numbers is "
        << sum << L'.' << endl;
}

// Uses the Concurrency Runtime to compute the sum of Fibonacci numbers in parallel.
void concrct_parallel_fibonacci_sum(int count)
{
    combinable<int> sums;
    parallel_for(0, count, [&sums](int i)
    {
        sums.local() += fibonacci(i);
    });

    wcout << L"The sum of the first " << count << L" Fibonacci numbers is "
        << sums.combine(plus<int>()) << L'.' << endl;
}

int wmain()
{
    const int count = 35;

    wcout << L"Using OpenMP..." << endl;
    omp_parallel_fibonacci_sum(count);

    wcout << L"Using the Concurrency Runtime..." << endl;
    concrct_parallel_fibonacci_sum(count);
}

```

This example produces the following output.

```
Using OpenMP...
The sum of the first 35 Fibonacci numbers is 14930351.
Using the Concurrency Runtime...
The sum of the first 35 Fibonacci numbers is 14930351.
```

For more information about the `combinable` class, see [Parallel Containers and Objects](#).

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `concr-omp-fibonacci-reduction.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc /openmp concr-omp-fibonacci-reduction.cpp
```

## See also

[Migrating from OpenMP to the Concurrency Runtime](#)  
[Parallel Containers and Objects](#)

# Parallel Patterns Library (PPL)

3/4/2019 • 3 minutes to read • [Edit Online](#)

The Parallel Patterns Library (PPL) provides an imperative programming model that promotes scalability and ease-of-use for developing concurrent applications. The PPL builds on the scheduling and resource management components of the Concurrency Runtime. It raises the level of abstraction between your application code and the underlying threading mechanism by providing generic, type-safe algorithms and containers that act on data in parallel. The PPL also lets you develop applications that scale by providing alternatives to shared state.

The PPL provides the following features:

- *Task Parallelism*: a mechanism that works on top of the Windows ThreadPool to execute several work items (tasks) in parallel
- *Parallel algorithms*: generic algorithms that works on top of the Concurrency Runtime to act on collections of data in parallel
- *Parallel containers and objects*: generic container types that provide safe concurrent access to their elements

## Example

The PPL provides a programming model that resembles the C++ Standard Library. The following example demonstrates many features of the PPL. It computes several Fibonacci numbers serially and in parallel. Both computations act on a `std::array` object. The example also prints to the console the time that is required to perform both computations.

The serial version uses the C++ Standard Library `std::for_each` algorithm to traverse the array and stores the results in a `std::vector` object. The parallel version performs the same task, but uses the PPL `concurrency::parallel_for_each` algorithm and stores the results in a `concurrency::concurrent_vector` object. The `concurrent_vector` class enables each loop iteration to concurrently add elements without the requirement to synchronize write access to the container.

Because `parallel_for_each` acts concurrently, the parallel version of this example must sort the `concurrent_vector` object to produce the same results as the serial version.

Note that the example uses a naïve method to compute the Fibonacci numbers; however, this method illustrates how the Concurrency Runtime can improve the performance of long computations.

```
// parallel-fibonacci.cpp
// compile with: /EHsc
#include <windows.h>
#include <ppl.h>
#include <concurrent_vector.h>
#include <array>
#include <vector>
#include <tuple>
#include <algorithm>
#include <iostream>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
```

```

// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}

// Computes the nth Fibonacci number.
int fibonacci(int n)
{
    if(n < 2)
        return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

int wmain()
{
    __int64 elapsed;

    // An array of Fibonacci numbers to compute.
    array<int, 4> a = { 24, 26, 41, 42 };

    // The results of the serial computation.
    vector<tuple<int,int>> results1;

    // The results of the parallel computation.
    concurrent_vector<tuple<int,int>> results2;

    // Use the for_each algorithm to compute the results serially.
    elapsed = time_call([&]
    {
        for_each (begin(a), end(a), [&](int n) {
            results1.push_back(make_tuple(n, fibonacci(n)));
        });
    });
    wcout << L"serial time: " << elapsed << L" ms" << endl;

    // Use the parallel_for_each algorithm to perform the same task.
    elapsed = time_call([&]
    {
        parallel_for_each (begin(a), end(a), [&](int n) {
            results2.push_back(make_tuple(n, fibonacci(n)));
        });

        // Because parallel_for_each acts concurrently, the results do not
        // have a pre-determined order. Sort the concurrent_vector object
        // so that the results match the serial version.
        sort(begin(results2), end(results2));
    });
    wcout << L"parallel time: " << elapsed << L" ms" << endl << endl;

    // Print the results.
    for_each (begin(results2), end(results2), [](tuple<int,int>& pair) {
        wcout << L"fib(" << get<0>(pair) << L"): " << get<1>(pair) << endl;
    });
}

```

The following sample output is for a computer that has four processors.

```
serial time: 9250 ms
parallel time: 5726 ms
```

```
fib(24): 46368
fib(26): 121393
fib(41): 165580141
fib(42): 267914296
```

Each iteration of the loop requires a different amount of time to finish. The performance of `parallel_for_each` is bounded by the operation that finishes last. Therefore, you should not expect linear performance improvements between the serial and parallel versions of this example.

## Related Topics

TITLE	DESCRIPTION
<a href="#">Task Parallelism</a>	Describes the role of tasks and task groups in the PPL.
<a href="#">Parallel Algorithms</a>	Describes how to use parallel algorithms such as <code>parallel_for</code> and <code>parallel_for_each</code> .
<a href="#">Parallel Containers and Objects</a>	Describes the various parallel containers and objects that are provided by the PPL.
<a href="#">Cancellation in the PPL</a>	Explains how to cancel the work that is being performed by a parallel algorithm.
<a href="#">Concurrency Runtime</a>	Describes the Concurrency Runtime, which simplifies parallel programming, and contains links to related topics.

# Task Parallelism (Concurrency Runtime)

11/8/2018 • 27 minutes to read • [Edit Online](#)

In the Concurrency Runtime, a *task* is a unit of work that performs a specific job and typically runs in parallel with other tasks. A task can be decomposed into additional, more fine-grained tasks that are organized into a *task group*.

You use tasks when you write asynchronous code and want some operation to occur after the asynchronous operation completes. For example, you could use a task to asynchronously read from a file and then use another task—a *continuation task*, which is explained later in this document—to process the data after it becomes available. Conversely, you can use task groups to decompose parallel work into smaller pieces. For example, suppose you have a recursive algorithm that divides the remaining work into two partitions. You can use task groups to run these partitions concurrently, and then wait for the divided work to complete.

## TIP

When you want to apply the same routine to every element of a collection in parallel, use a parallel algorithm, such as `concurrency::parallel_for`, instead of a task or task group. For more information about parallel algorithms, see [Parallel Algorithms](#).

## Key Points

- When you pass variables to a lambda expression by reference, you must guarantee that the lifetime of that variable persists until the task finishes.
- Use tasks (the `concurrency::task` class) when you write asynchronous code. The task class uses the Windows ThreadPool as its scheduler, not the Concurrency Runtime.
- Use task groups (the `concurrency::task_group` class or the `concurrency::parallel_invoke` algorithm) when you want to decompose parallel work into smaller pieces and then wait for those smaller pieces to complete.
- Use the `concurrency::task::then` method to create continuations. A *continuation* is a task that runs asynchronously after another task completes. You can connect any number of continuations to form a chain of asynchronous work.
- A task-based continuation is always scheduled for execution when the antecedent task finishes, even when the antecedent task is canceled or throws an exception.
- Use `concurrency::when_all` to create a task that completes after every member of a set of tasks completes. Use `concurrency::when_any` to create a task that completes after one member of a set of tasks completes.
- Tasks and task groups can participate in the Parallel Patterns Library (PPL) cancellation mechanism. For more information, see [Cancellation in the PPL](#).
- To learn how the runtime handles exceptions that are thrown by tasks and task groups, see [Exception Handling](#).

## In this Document

- [Using Lambda Expressions](#)

- [The task Class](#)
- [Continuation Tasks](#)
- [Value-Based Versus Task-Based Continuations](#)
- [Composing Tasks](#)
  - [The when\\_all Function](#)
  - [The when\\_any Function](#)
- [Delayed Task Execution](#)
- [Task Groups](#)
- [Comparing task\\_group to structured\\_task\\_group](#)
- [Example](#)
- [Robust Programming](#)

## Using Lambda Expressions

Because of their succinct syntax, lambda expressions are a common way to define the work that is performed by tasks and task groups. Here are some usage tips:

- Because tasks typically run on background threads, be aware of the object lifetime when you capture variables in lambda expressions. When you capture a variable by value, a copy of that variable is made in the lambda body. When you capture by reference, a copy is not made. Therefore, ensure that the lifetime of any variable that you capture by reference outlives the task that uses it.
- When you pass a lambda expression to a task, don't capture variables that are allocated on the stack by reference.
- Be explicit about the variables you capture in lambda expressions so that you can identify what you're capturing by value versus by reference. For this reason we recommend that you do not use the `[=]` or `[&]` options for lambda expressions.

A common pattern is when one task in a continuation chain assigns to a variable, and another task reads that variable. You can't capture by value because each continuation task would hold a different copy of the variable. For stack-allocated variables, you also can't capture by reference because the variable may no longer be valid.

To solve this problem, use a smart pointer, such as `std::shared_ptr`, to wrap the variable and pass the smart pointer by value. In this way, the underlying object can be assigned to and read from, and will outlive the tasks that use it. Use this technique even when the variable is a pointer or a reference-counted handle (`^`) to a Windows Runtime object. Here's a basic example:

```

// lambda-task-lifetime.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <iostream>
#include <string>

using namespace concurrency;
using namespace std;

task<wstring> write_to_string()
{
    // Create a shared pointer to a string that is
    // assigned to and read by multiple tasks.
    // By using a shared pointer, the string outlives
    // the tasks, which can run in the background after
    // this function exits.
    auto s = make_shared<wstring>(L"Value 1");

    return create_task([s]
    {
        // Print the current value.
        wcout << L"Current value: " << *s << endl;
        // Assign to a new value.
        *s = L"Value 2";

    }).then([s]
    {
        // Print the current value.
        wcout << L"Current value: " << *s << endl;
        // Assign to a new value and return the string.
        *s = L"Value 3";
        return *s;
    });
}

int wmain()
{
    // Create a chain of tasks that work with a string.
    auto t = write_to_string();

    // Wait for the tasks to finish and print the result.
    wcout << L"Final value: " << t.get() << endl;
}

/* Output:
Current value: Value 1
Current value: Value 2
Final value: Value 3
*/

```

For more information about lambda expressions, see [Lambda Expressions](#).

## The task Class

You can use the `concurrency::task` class to compose tasks into a set of dependent operations. This composition model is supported by the notion of *continuations*. A continuation enables code to be executed when the previous, or *antecedent*, task completes. The result of the antecedent task is passed as the input to the one or more continuation tasks. When an antecedent task completes, any continuation tasks that are waiting on it are scheduled for execution. Each continuation task receives a copy of the result of the antecedent task. In turn, those continuation tasks may also be antecedent tasks for other continuations, thereby creating a chain of tasks. Continuations help you create arbitrary-length chains of tasks that have specific dependencies among them. In addition, a task can participate in cancellation either before a task starts or in a cooperative manner while it is running. For more information about this cancellation model, see

## Cancellation in the PPL.

`task` is a template class. The type parameter `T` is the type of the result that is produced by the task. This type can be `void` if the task does not return a value. `T` cannot use the `const` modifier.

When you create a task, you provide a *work function* that performs the task body. This work function comes in the form of a lambda function, function pointer, or function object. To wait for a task to finish without obtaining the result, call the `concurrency::task::wait` method. The `task::wait` method returns a `concurrency::task_status` value that describes whether the task was completed or canceled. To get the result of the task, call the `concurrency::task::get` method. This method calls `task::wait` to wait for the task to finish, and therefore blocks execution of the current thread until the result is available.

The following example shows how to create a task, wait for its result, and display its value. The examples in this documentation use lambda functions because they provide a more succinct syntax. However, you can also use function pointers and function objects when you use tasks.

```
// basic-task.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create a task.
    task<int> t([]()
    {
        return 42;
    });

    // In this example, you don't necessarily need to call wait() because
    // the call to get() also waits for the result.
    t.wait();

    // Print the result.
    wcout << t.get() << endl;
}

/* Output:
42
*/
```

When you use the `concurrency::create_task` function, you can use the `auto` keyword instead of declaring the type. For example, consider this code that creates and prints the identity matrix:

```

// create-task.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <string>
#include <iostream>
#include <array>

using namespace concurrency;
using namespace std;

int wmain()
{
    task<array<array<int, 10>, 10>> create_identity_matrix([]
    {
        array<array<int, 10>, 10> matrix;
        int row = 0;
        for_each(begin(matrix), end(matrix), [&row](array<int, 10>& matrixRow)
        {
            fill(begin(matrixRow), end(matrixRow), 0);
            matrixRow[row] = 1;
            row++;
        });
        return matrix;
    });

    auto print_matrix = create_identity_matrix.then([](array<array<int, 10>, 10> matrix)
    {
        for_each(begin(matrix), end(matrix), [](array<int, 10>& matrixRow)
        {
            wstring comma;
            for_each(begin(matrixRow), end(matrixRow), [&comma](int n)
            {
                wcout << comma << n;
                comma = L", ";
            });
            wcout << endl;
        });
    });

    print_matrix.wait();
}
/* Output:
1, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 1, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 1, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 1, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 1, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 1, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 1, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 1, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 1, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 1
*/

```

You can use the `create_task` function to create the equivalent operation.

```

auto create_identity_matrix = create_task([]
{
    array<array<int, 10>, 10> matrix;
    int row = 0;
    for_each(begin(matrix), end(matrix), [&row](array<int, 10>& matrixRow)
    {
        fill(begin(matrixRow), end(matrixRow), 0);
        matrixRow[row] = 1;
        row++;
    });
    return matrix;
});

```

If an exception is thrown during the execution of a task, the runtime marshals that exception in the subsequent call to `task::get` or `task::wait`, or to a task-based continuation. For more information about the task exception-handling mechanism, see [Exception Handling](#).

For an example that uses `task`, `concurrency::task_completion_event`, cancellation, see [Walkthrough: Connecting Using Tasks and XML HTTP Requests](#). (The `task_completion_event` class is described later in this document.)

#### TIP

To learn details that are specific to tasks in UWP apps, see [Asynchronous programming in C++](#) and [Creating Asynchronous Operations in C++ for UWP Apps](#).

## Continuation Tasks

In asynchronous programming, it is very common for one asynchronous operation, on completion, to invoke a second operation and pass data to it. Traditionally, this is done by using callback methods. In the Concurrency Runtime, the same functionality is provided by *continuation tasks*. A continuation task (also known just as a continuation) is an asynchronous task that is invoked by another task, which is known as the *antecedent*, when the antecedent completes. By using continuations, you can:

- Pass data from the antecedent to the continuation.
- Specify the precise conditions under which the continuation is invoked or not invoked.
- Cancel a continuation either before it starts or cooperatively while it is running.
- Provide hints about how the continuation should be scheduled. (This applies to Universal Windows Platform (UWP) apps only. For more information, see [Creating Asynchronous Operations in C++ for UWP Apps](#).)
- Invoke multiple continuations from the same antecedent.
- Invoke one continuation when all or any of multiple antecedents complete.
- Chain continuations one after another to any length.
- Use a continuation to handle exceptions that are thrown by the antecedent.

These features enable you to execute one or more tasks when the first task completes. For example, you can create a continuation that compresses a file after the first task reads it from disk.

The following example modifies the previous one to use the `concurrency::task::then` method to schedule a continuation that prints the value of the antecedent task when it is available.

```

// basic-continuation.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    auto t = create_task([]() -> int
    {
        return 42;
    });

    t.then([](int result)
    {
        wcout << result << endl;
    }).wait();

    // Alternatively, you can chain the tasks directly and
    // eliminate the local variable.
    /*create_task([]() -> int
    {
        return 42;
    }).then([](int result)
    {
        wcout << result << endl;
    }).wait();*/
}

/* Output:
42
*/

```

You can chain and nest tasks to any length. A task can also have multiple continuations. The following example illustrates a basic continuation chain that increments the value of the previous task three times.

```

// continuation-chain.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    auto t = create_task([]() -> int
    {
        return 0;
    });

    // Create a lambda that increments its input value.
    auto increment = [](int n) { return n + 1; };

    // Run a chain of continuations and print the result.
    int result = t.then(increment).then(increment).then(increment).get();
    wcout << result << endl;
}

/* Output:
3
*/

```

A continuation can also return another task. If there is no cancellation, then this task is executed before the subsequent continuation. This technique is known as *asynchronous unwrapping*. Asynchronous unwrapping is useful when you want to perform additional work in the background, but do not want the current task to block the current thread. (This is common in UWP apps, where continuations can run on the UI thread). The following example shows three tasks. The first task returns another task that is run before a continuation task.

```
// async-unwrapping.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    auto t = create_task([]()
    {
        wcout << L"Task A" << endl;

        // Create an inner task that runs before any continuation
        // of the outer task.
        return create_task([]()
        {
            wcout << L"Task B" << endl;
        });
    });

    // Run and wait for a continuation of the outer task.
    t.then([]()
    {
        wcout << L"Task C" << endl;
    }).wait();
}

/* Output:
Task A
Task B
Task C
*/
```

### IMPORTANT

When a continuation of a task returns a nested task of type `N`, the resulting task has the type `N`, not `task<N>`, and completes when the nested task completes. In other words, the continuation performs the unwrapping of the nested task.

## Value-Based Versus Task-Based Continuations

Given a `task` object whose return type is `T`, you can provide a value of type `T` or `task<T>` to its continuation tasks. A continuation that takes type `T` is known as a *value-based continuation*. A value-based continuation is scheduled for execution when the antecedent task completes without error and is not canceled. A continuation that takes type `task<T>` as its parameter is known as a *task-based continuation*. A task-based continuation is always scheduled for execution when the antecedent task finishes, even when the antecedent task is canceled or throws an exception. You can then call `task::get` to get the result of the antecedent task. If the antecedent task was canceled, `task::get` throws `concurrency::task_canceled`. If the antecedent task threw an exception, `task::get` rethrows that exception. A task-based continuation is not marked as canceled when its antecedent task is canceled.

# Composing Tasks

This section describes the `concurrency::when_all` and `concurrency::when_any` functions, which can help you compose multiple tasks to implement common patterns.

## The `when_all` Function

The `when_all` function produces a task that completes after a set of tasks complete. This function returns a `std::vector` object that contains the result of each task in the set. The following basic example uses `when_all` to create a task that represents the completion of three other tasks.

```
// join-tasks.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <array>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Start multiple tasks.
    array<task<void>, 3> tasks =
    {
        create_task([] { wcout << L"Hello from taskA." << endl; }),
        create_task([] { wcout << L"Hello from taskB." << endl; }),
        create_task([] { wcout << L"Hello from taskC." << endl; })
    };

    auto joinTask = when_all(begin(tasks), end(tasks));

    // Print a message from the joining thread.
    wcout << L"Hello from the joining thread." << endl;

    // Wait for the tasks to finish.
    joinTask.wait();
}

/* Sample output:
Hello from the joining thread.
Hello from taskA.
Hello from taskC.
Hello from taskB.
*/
```

### NOTE

The tasks that you pass to `when_all` must be uniform. In other words, they must all return the same type.

You can also use the `&&` syntax to produce a task that completes after a set of tasks complete, as shown in the following example.

```
auto t = t1 && t2; // same as when_all
```

It is common to use a continuation together with `when_all` to perform an action after a set of tasks finishes. The following example modifies the previous one to print the sum of three tasks that each produce an `int` result.

```

// Start multiple tasks.
array<task<int>, 3> tasks =
{
    create_task([]() -> int { return 88; }),
    create_task([]() -> int { return 42; }),
    create_task([]() -> int { return 99; })
};

auto joinTask = when_all(begin(tasks), end(tasks)).then([](vector<int> results)
{
    wcout << L"The sum is "
        << accumulate(begin(results), end(results), 0)
        << L'.' << endl;
});

// Print a message from the joining thread.
wcout << L"Hello from the joining thread." << endl;

// Wait for the tasks to finish.
joinTask.wait();

/* Output:
    Hello from the joining thread.
    The sum is 229.
*/

```

In this example, you can also specify `task<vector<int>>` to produce a task-based continuation.

If any task in a set of tasks is canceled or throws an exception, `when_all` immediately completes and does not wait for the remaining tasks to finish. If an exception is thrown, the runtime rethrows the exception when you call `task::get` or `task::wait` on the task object that `when_all` returns. If more than one task throws, the runtime chooses one of them. Therefore, ensure that you observe all exceptions after all tasks complete; an unhandled task exception causes the app to terminate.

Here's a utility function that you can use to ensure that your program observes all exceptions. For each task in the provided range, `observe_all_exceptions` triggers any exception that occurred to be rethrown and then swallows that exception.

```

// Observes all exceptions that occurred in all tasks in the given range.
template<class T, class InIt>
void observe_all_exceptions(InIt first, InIt last)
{
    std::for_each(first, last, [](concurrency::task<T> t)
    {
        t.then([](concurrency::task<T> previousTask)
        {
            try
            {
                previousTask.get();
            }
            // Although you could catch (...), this demonstrates how to catch specific exceptions. Your
app
            // might handle different exception types in different ways.
            catch (Platform::Exception^)
            {
                // Swallow the exception.
            }
            catch (const std::exception&)
            {
                // Swallow the exception.
            }
        });
    });
}

```

Consider a UWP app that uses C++ and XAML and writes a set of files to disk. The following example shows how to use `when_all` and `observe_all_exceptions` to ensure that the program observes all exceptions.

```

// Writes content to files in the provided storage folder.
// The first element in each pair is the file name. The second element holds the file contents.
task<void> MainPage::WriteFilesAsync(StorageFolder^ folder, const vector<pair<String^, String^>>&
fileContents)
{
    // For each file, create a task chain that creates the file and then writes content to it. Then add
the task chain to a vector of tasks.
    vector<task<void>> tasks;
    for (auto fileContent : fileContents)
    {
        auto fileName = fileContent.first;
        auto content = fileContent.second;

        // Create the file. The CreationCollisionOption::FailIfExists flag specifies to fail if the file
already exists.
        tasks.emplace_back(create_task(folder->CreateFileAsync(fileName,
CreationCollisionOption::FailIfExists)).then([content](StorageFile^ file)
        {
            // Write its contents.
            return create_task(FileIO::WriteTextAsync(file, content));
        }));
    }

    // When all tasks finish, create a continuation task that observes any exceptions that occurred.
return when_all(begin(tasks), end(tasks)).then([tasks](task<void> previousTask)
{
    task_status status = completed;
    try
    {
        status = previousTask.wait();
    }
    catch (COMException^ e)
    {
        // We'll handle the specific errors below.
    }
    // TODO: If other exception types might happen, add catch handlers here.

    // Ensure that we observe all exceptions.
    observe_all_exceptions<void>(begin(tasks), end(tasks));

    // Cancel any continuations that occur after this task if any previous task was canceled.
    // Although cancellation is not part of this example, we recommend this pattern for cases that
do.
    if (status == canceled)
    {
        cancel_current_task();
    }
});
}

```

To run this example

1. In MainPage.xaml, add a `Button` control.

```
<Button x:Name="Button1" Click="Button_Click">Write files</Button>
```

1. In MainPage.xaml.h, add these forward declarations to the `private` section of the `MainPage` class declaration.

```

void Button_Click(Platform::Object^ sender, Windows::UI::Xaml::RoutedEventArgs^ e);
concurrency::task<void> WriteFilesAsync(Windows::Storage::StorageFolder^ folder, const
std::vector<std::pair<Platform::String^, Platform::String^>>& fileContents);

```

1. In MainPage.xaml.cpp, implement the `Button_Click` event handler.

```
// A button click handler that demonstrates the scenario.
void MainPage::Button_Click(Object^ sender, RoutedEventArgs^ e)
{
    // In this example, the same file name is specified two times. WriteFilesAsync fails if one of the
    // files already exists.
    vector<pair<String^, String^>> fileContents;
    fileContents.emplace_back(make_pair(ref new String(L"file1.txt"), ref new String(L"Contents of file
1")));
    fileContents.emplace_back(make_pair(ref new String(L"file2.txt"), ref new String(L"Contents of file
2")));
    fileContents.emplace_back(make_pair(ref new String(L"file1.txt"), ref new String(L"Contents of file
3")));

    Button1->IsEnabled = false; // Disable the button during the operation.
    WriteFilesAsync(ApplicationData::Current->TemporaryFolder, fileContents).then([this](task<void>
previousTask)
    {
        try
        {
            previousTask.get();
        }
        // Although cancellation is not part of this example, we recommend this pattern for cases that
        // do.
        catch (const task_canceled&)
        {
            // Your app might show a message to the user, or handle the error in some other way.
        }

        Button1->IsEnabled = true; // Enable the button.
    });
}
```

1. In MainPage.xaml.cpp, implement `WriteFilesAsync` as shown in the example.

#### TIP

`when_all` is a non-blocking function that produces a `task` as its result. Unlike `task::wait`, it is safe to call this function in a UWP app on the ASTA (Application STA) thread.

### The `when_any` Function

The `when_any` function produces a task that completes when the first task in a set of tasks completes. This function returns a `std::pair` object that contains the result of the completed task and the index of that task in the set.

The `when_any` function is especially useful in the following scenarios:

- Redundant operations. Consider an algorithm or operation that can be performed in many ways. You can use the `when_any` function to select the operation that finishes first and then cancel the remaining operations.
- Interleaved operations. You can start multiple operations that all must finish and use the `when_any` function to process results as each operation finishes. After one operation finishes, you can start one or more additional tasks.
- Throttled operations. You can use the `when_any` function to extend the previous scenario by limiting the number of concurrent operations.
- Expired operations. You can use the `when_any` function to select between one or more tasks and a task

that finishes after a specific time.

As with `when_all`, it is common to use a continuation that has `when_any` to perform action when the first in a set of tasks finish. The following basic example uses `when_any` to create a task that completes when the first of three other tasks completes.

```
// select-task.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <array>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Start multiple tasks.
    array<task<int>, 3> tasks = {
        create_task([]() -> int { return 88; }),
        create_task([]() -> int { return 42; }),
        create_task([]() -> int { return 99; })
    };

    // Select the first to finish.
    when_any(begin(tasks), end(tasks)).then([](pair<int, size_t> result)
    {
        wcout << "First task to finish returns "
            << result.first
            << " and has index "
            << result.second
            << L'.' << endl;
    }).wait();
}

/* Sample output:
   First task to finish returns 42 and has index 1.
*/
```

In this example, you can also specify `task<pair<int, size_t>>` to produce a task-based continuation.

#### NOTE

As with `when_all`, the tasks that you pass to `when_any` must all return the same type.

You can also use the `||` syntax to produce a task that completes after the first task in a set of tasks completes, as shown in the following example.

```
auto t = t1 || t2; // same as when_any
```

#### TIP

As with `when_all`, `when_any` is non-blocking and is safe to call in a UWP app on the ASTA thread.

## Delayed Task Execution

It is sometimes necessary to delay the execution of a task until a condition is satisfied, or to start a task in response to an external event. For example, in asynchronous programming, you might have to start a task in response to an I/O completion event.

Two ways to accomplish this are to use a continuation or to start a task and wait on an event inside the task's work function. However, there are cases where it is not possible to use one of these techniques. For example, to create a continuation, you must have the antecedent task. However, if you do not have the antecedent task, you can create a *task completion event* and later chain that completion event to the antecedent task when it becomes available. In addition, because a waiting task also blocks a thread, you can use task completion events to perform work when an asynchronous operation completes, and thereby free a thread.

The `concurrency::task_completion_event` class helps simplify such composition of tasks. Like the `task` class, the type parameter `T` is the type of the result that is produced by the task. This type can be `void` if the task does not return a value. `T` cannot use the `const` modifier. Typically, a `task_completion_event` object is provided to a thread or task that will signal it when the value for it becomes available. At the same time, one or more tasks are set as listeners of that event. When the event is set, the listener tasks complete and their continuations are scheduled to run.

For an example that uses `task_completion_event` to implement a task that completes after a delay, see [How to: Create a Task that Completes After a Delay](#).

## Task Groups

A *task group* organizes a collection of tasks. Task groups push tasks on to a work-stealing queue. The scheduler removes tasks from this queue and executes them on available computing resources. After you add tasks to a task group, you can wait for all tasks to finish or cancel tasks that have not yet started.

The PPL uses the `concurrency::task_group` and `concurrency::structured_task_group` classes to represent task groups, and the `concurrency::task_handle` class to represent the tasks that run in these groups. The `task_handle` class encapsulates the code that performs work. Like the `task` class, the work function comes in the form of a lambda function, function pointer, or function object. You typically do not need to work with `task_handle` objects directly. Instead, you pass work functions to a task group, and the task group creates and manages the `task_handle` objects.

The PPL divides task groups into these two categories: *unstructured task groups* and *structured task groups*. The PPL uses the `task_group` class to represent unstructured task groups and the `structured_task_group` class to represent structured task groups.

### IMPORTANT

The PPL also defines the `concurrency::parallel_invoke` algorithm, which uses the `structured_task_group` class to execute a set of tasks in parallel. Because the `parallel_invoke` algorithm has a more succinct syntax, we recommend that you use it instead of the `structured_task_group` class when you can. The topic [Parallel Algorithms](#) describes `parallel_invoke` in greater detail.

Use `parallel_invoke` when you have several independent tasks that you want to execute at the same time, and you must wait for all tasks to finish before you continue. This technique is often referred to as *fork and join* parallelism. Use `task_group` when you have several independent tasks that you want to execute at the same time, but you want to wait for the tasks to finish at a later time. For example, you can add tasks to a `task_group` object and wait for the tasks to finish in another function or from another thread.

Task groups support the concept of cancellation. Cancellation enables you to signal to all active tasks that you want to cancel the overall operation. Cancellation also prevents tasks that have not yet started from starting. For more information about cancellation, see [Cancellation in the PPL](#).

The runtime also provides an exception-handling model that enables you to throw an exception from a task and handle that exception when you wait for the associated task group to finish. For more information about this exception-handling model, see [Exception Handling](#).

## Comparing task\_group to structured\_task\_group

Although we recommend that you use `task_group` or `parallel_invoke` instead of the `structured_task_group` class, there are cases where you want to use `structured_task_group`, for example, when you write a parallel algorithm that performs a variable number of tasks or requires support for cancellation. This section explains the differences between the `task_group` and `structured_task_group` classes.

The `task_group` class is thread-safe. Therefore you can add tasks to a `task_group` object from multiple threads and wait on or cancel a `task_group` object from multiple threads. The construction and destruction of a `structured_task_group` object must occur in the same lexical scope. In addition, all operations on a `structured_task_group` object must occur on the same thread. The exception to this rule is the `concurrency::structured_task_group::cancel` and `concurrency::structured_task_group::is_canceling` methods. A child task can call these methods to cancel the parent task group or check for cancellation at any time.

You can run additional tasks on a `task_group` object after you call the `concurrency::task_group::wait` or `concurrency::task_group::run_and_wait` method. Conversely, if you run additional tasks on a `structured_task_group` object after you call the `concurrency::structured_task_group::wait` or `concurrency::structured_task_group::run_and_wait` methods, then the behavior is undefined.

Because the `structured_task_group` class does not synchronize across threads, it has less execution overhead than the `task_group` class. Therefore, if your problem does not require that you schedule work from multiple threads and you cannot use the `parallel_invoke` algorithm, the `structured_task_group` class can help you write better performing code.

If you use one `structured_task_group` object inside another `structured_task_group` object, the inner object must finish and be destroyed before the outer object finishes. The `task_group` class does not require for nested task groups to finish before the outer group finishes.

Unstructured task groups and structured task groups work with task handles in different ways. You can pass work functions directly to a `task_group` object; the `task_group` object will create and manage the task handle for you. The `structured_task_group` class requires you to manage a `task_handle` object for each task. Every `task_handle` object must remain valid throughout the lifetime of its associated `structured_task_group` object. Use the `concurrency::make_task` function to create a `task_handle` object, as shown in the following basic example:

```
// make-task-structure.cpp
// compile with: /EHsc
#include <pp1.h>

using namespace concurrency;

int wmain()
{
    // Use the make_task function to define several tasks.
    auto task1 = make_task([] { /*TODO: Define the task body.*/ });
    auto task2 = make_task([] { /*TODO: Define the task body.*/ });
    auto task3 = make_task([] { /*TODO: Define the task body.*/ });

    // Create a structured task group and run the tasks concurrently.

    structured_task_group tasks;

    tasks.run(task1);
    tasks.run(task2);
    tasks.run_and_wait(task3);
}
```

To manage task handles for cases where you have a variable number of tasks, use a stack-allocation routine

such as `_alloca` or a container class, such as `std::vector`.

Both `task_group` and `structured_task_group` support cancellation. For more information about cancellation, see [Cancellation in the PPL](#).

## Example

The following basic example shows how to work with task groups. This example uses the `parallel_invoke` algorithm to perform two tasks concurrently. Each task adds sub-tasks to a `task_group` object. Note that the `task_group` class allows for multiple tasks to add tasks to it concurrently.

```
// using-task-groups.cpp
// compile with: /EHsc
#include <ppl.h>
#include <sstream>
#include <iostream>

using namespace concurrency;
using namespace std;

// Prints a message to the console.
template<typename T>
void print_message(T t)
{
    wstringstream ss;
    ss << L"Message from task: " << t << endl;
    wcout << ss.str();
}

int wmain()
{
    // A task_group object that can be used from multiple threads.
    task_group tasks;

    // Concurrently add several tasks to the task_group object.
    parallel_invoke(
        [&] {
            // Add a few tasks to the task_group object.
            tasks.run([] { print_message(L"Hello"); });
            tasks.run([] { print_message(42); });
        },
        [&] {
            // Add one additional task to the task_group object.
            tasks.run([] { print_message(3.14); });
        }
    );

    // Wait for all tasks to finish.
    tasks.wait();
}
```

The following is sample output for this example:

```
Message from task: Hello
Message from task: 3.14
Message from task: 42
```

Because the `parallel_invoke` algorithm runs tasks concurrently, the order of the output messages could vary.

For complete examples that show how to use the `parallel_invoke` algorithm, see [How to: Use parallel\\_invoke to Write a Parallel Sort Routine](#) and [How to: Use parallel\\_invoke to Execute Parallel](#)

[Operations](#). For a complete example that uses the `task_group` class to implement asynchronous futures, see [Walkthrough: Implementing Futures](#).

## Robust Programming

Make sure that you understand the role of cancellation and exception handling when you use tasks, task groups, and parallel algorithms. For example, in a tree of parallel work, a task that is canceled prevents child tasks from running. This can cause problems if one of the child tasks performs an operation that is important to your application, such as freeing a resource. In addition, if a child task throws an exception, that exception could propagate through an object destructor and cause undefined behavior in your application. For an example that illustrates these points, see the [Understand how Cancellation and Exception Handling Affect Object Destruction](#) section in the Best Practices in the Parallel Patterns Library document. For more information about the cancellation and exception-handling models in the PPL, see [Cancellation](#) and [Exception Handling](#).

## Related Topics

TITLE	DESCRIPTION
<a href="#">How to: Use <code>parallel_invoke</code> to Write a Parallel Sort Routine</a>	Shows how to use the <code>parallel_invoke</code> algorithm to improve the performance of the bitonic sort algorithm.
<a href="#">How to: Use <code>parallel_invoke</code> to Execute Parallel Operations</a>	Shows how to use the <code>parallel_invoke</code> algorithm to improve the performance of a program that performs multiple operations on a shared data source.
<a href="#">How to: Create a Task that Completes After a Delay</a>	Shows how to use the <code>task</code> , <code>cancellation_token_source</code> , <code>cancellation_token</code> , and <code>task_completion_event</code> classes to create a task that completes after a delay.
<a href="#">Walkthrough: Implementing Futures</a>	Shows how to combine existing functionality in the Concurrency Runtime into something that does more.
<a href="#">Parallel Patterns Library (PPL)</a>	Describes the PPL, which provides an imperative programming model for developing concurrent applications.

## Reference

[task Class \(Concurrency Runtime\)](#)

[task\\_completion\\_event Class](#)

[when\\_all Function](#)

[when\\_any Function](#)

[task\\_group Class](#)

[parallel\\_invoke Function](#)

[structured\\_task\\_group Class](#)

# How to: Use `parallel_invoke` to Write a Parallel Sort Routine

3/4/2019 • 8 minutes to read • [Edit Online](#)

This document describes how to use the `parallel_invoke` algorithm to improve the performance of the bitonic sort algorithm. The bitonic sort algorithm recursively divides the input sequence into smaller sorted partitions. The bitonic sort algorithm can run in parallel because each partition operation is independent of all other operations.

Although the bitonic sort is an example of a *sorting network* that sorts all combinations of input sequences, this example sorts sequences whose lengths are a power of two.

## NOTE

This example uses a parallel sort routine for illustration. You can also use the built-in sorting algorithms that the PPL provides: `concurrency::parallel_sort`, `concurrency::parallel_buffered_sort`, and `concurrency::parallel_radixsort`. For more information, see [Parallel Algorithms](#).

## Sections

This document describes the following tasks:

- [Performing Bitonic Sort Serially](#)
- [Using `parallel\_invoke` to Perform Bitonic Sort in Parallel](#)

## Performing Bitonic Sort Serially

The following example shows the serial version of the bitonic sort algorithm. The `bitonic_sort` function divides the sequence into two partitions, sorts those partitions in opposite directions, and then merges the results. This function calls itself two times recursively to sort each partition.

```

const bool INCREASING = true;
const bool DECREASING = false;

// Comparator function for the bitonic sort algorithm.
template <class T>
void compare(T* items, int i, int j, bool dir)
{
    if (dir == (items[i] > items[j]))
    {
        swap(items[i], items[j]);
    }
}

// Sorts a bitonic sequence in the specified order.
template <class T>
void bitonic_merge(T* items, int lo, int n, bool dir)
{
    if (n > 1)
    {
        int m = n / 2;
        for (int i = lo; i < lo + m; ++i)
        {
            compare(items, i, i + m, dir);
        }
        bitonic_merge(items, lo, m, dir);
        bitonic_merge(items, lo + m, m, dir);
    }
}

// Sorts the given sequence in the specified order.
template <class T>
void bitonic_sort(T* items, int lo, int n, bool dir)
{
    if (n > 1)
    {
        // Divide the array into two partitions and then sort
        // the partitions in different directions.
        int m = n / 2;
        bitonic_sort(items, lo, m, INCREASING);
        bitonic_sort(items, lo + m, m, DECREASING);

        // Merge the results.
        bitonic_merge(items, lo, n, dir);
    }
}

// Sorts the given sequence in increasing order.
template <class T>
void bitonic_sort(T* items, int size)
{
    bitonic_sort(items, 0, size, INCREASING);
}

```

[\[Top\]](#)

## Using `parallel_invoke` to Perform Bitonic Sort in Parallel

This section describes how to use the `parallel_invoke` algorithm to perform the bitonic sort algorithm in parallel.

### Procedures

To perform the bitonic sort algorithm in parallel

1. Add a `#include` directive for the header file `ppl.h`.

```
#include <ppl.h>
```

1. Add a `using` directive for the `concurrency` namespace.

```
using namespace concurrency;
```

1. Create a new function, called `parallel_bitonic_merge`, which uses the `parallel_invoke` algorithm to merge the sequences in parallel if there is sufficient amount of work to do. Otherwise, call `bitonic_merge` to merge the sequences serially.

```
// Sorts a bitonic sequence in the specified order.
template <class T>
void parallel_bitonic_merge(T* items, int lo, int n, bool dir)
{
    // Merge the sequences concurrently if there is sufficient work to do.
    if (n > 500)
    {
        int m = n / 2;
        for (int i = lo; i < lo + m; ++i)
        {
            compare(items, i, i + m, dir);
        }

        // Use the parallel_invoke algorithm to merge the sequences in parallel.
        parallel_invoke(
            [&items,lo,m,dir] { parallel_bitonic_merge(items, lo, m, dir); },
            [&items,lo,m,dir] { parallel_bitonic_merge(items, lo + m, m, dir); }
        );
    }
    // Otherwise, perform the work serially.
    else if (n > 1)
    {
        bitonic_merge(items, lo, n, dir);
    }
}
```

1. Perform a process that resembles the one in the previous step, but for the `bitonic_sort` function.

```
// Sorts the given sequence in the specified order.
template <class T>
void parallel_bitonic_sort(T* items, int lo, int n, bool dir)
{
    if (n > 1)
    {
        // Divide the array into two partitions and then sort
        // the partitions in different directions.
        int m = n / 2;

        // Sort the partitions in parallel.
        parallel_invoke(
            [&items,lo,m] { parallel_bitonic_sort(items, lo, m, INCREASING); },
            [&items,lo,m] { parallel_bitonic_sort(items, lo + m, m, DECREASING); }
        );

        // Merge the results.
        parallel_bitonic_merge(items, lo, n, dir);
    }
}
```

1. Create an overloaded version of the `parallel_bitonic_sort` function that sorts the array in increasing order.

```

// Sorts the given sequence in increasing order.
template <class T>
void parallel_bitonic_sort(T* items, int size)
{
    parallel_bitonic_sort(items, 0, size, INCREASING);
}

```

The `parallel_invoke` algorithm reduces overhead by performing the last of the series of tasks on the calling context. For example, in the `parallel_bitonic_sort` function, the first task runs on a separate context, and the second task runs on the calling context.

```

// Sort the partitions in parallel.
parallel_invoke(
    [&items,lo,m] { parallel_bitonic_sort(items, lo, m, INCREASING); },
    [&items,lo,m] { parallel_bitonic_sort(items, lo + m, m, DECREASING); }
);

```

The following complete example performs both the serial and the parallel versions of the bitonic sort algorithm. The example also prints to the console the time that is required to perform each computation.

```

// parallel-bitonic-sort.cpp
// compile with: /EHsc
#include <windows.h>
#include <algorithm>
#include <iostream>
#include <random>
#include <ppl.h>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}

const bool INCREASING = true;
const bool DECREASING = false;

// Comparator function for the bitonic sort algorithm.
template <class T>
void compare(T* items, int i, int j, bool dir)
{
    if (dir == (items[i] > items[j]))
    {
        swap(items[i], items[j]);
    }
}

// Sorts a bitonic sequence in the specified order.
template <class T>
void bitonic_merge(T* items, int lo, int n, bool dir)
{
    if (n > 1)
    {
        int m = n / 2;
        for (int i = lo; i < lo + m; ++i)
        {

```

```

        compare(items, i, i + m, dir);
    }
    bitonic_merge(items, lo, m, dir);
    bitonic_merge(items, lo + m, m, dir);
}
}

// Sorts the given sequence in the specified order.
template <class T>
void bitonic_sort(T* items, int lo, int n, bool dir)
{
    if (n > 1)
    {
        // Divide the array into two partitions and then sort
        // the partitions in different directions.
        int m = n / 2;
        bitonic_sort(items, lo, m, INCREASING);
        bitonic_sort(items, lo + m, m, DECREASING);

        // Merge the results.
        bitonic_merge(items, lo, n, dir);
    }
}

// Sorts the given sequence in increasing order.
template <class T>
void bitonic_sort(T* items, int size)
{
    bitonic_sort(items, 0, size, INCREASING);
}

// Sorts a bitonic sequence in the specified order.
template <class T>
void parallel_bitonic_merge(T* items, int lo, int n, bool dir)
{
    // Merge the sequences concurrently if there is sufficient work to do.
    if (n > 500)
    {
        int m = n / 2;
        for (int i = lo; i < lo + m; ++i)
        {
            compare(items, i, i + m, dir);
        }

        // Use the parallel_invoke algorithm to merge the sequences in parallel.
        parallel_invoke(
            [&items, lo, m, dir] { parallel_bitonic_merge(items, lo, m, dir); },
            [&items, lo, m, dir] { parallel_bitonic_merge(items, lo + m, m, dir); }
        );
    }
    // Otherwise, perform the work serially.
    else if (n > 1)
    {
        bitonic_merge(items, lo, n, dir);
    }
}

// Sorts the given sequence in the specified order.
template <class T>
void parallel_bitonic_sort(T* items, int lo, int n, bool dir)
{
    if (n > 1)
    {
        // Divide the array into two partitions and then sort
        // the partitions in different directions.
        int m = n / 2;

        // Sort the partitions in parallel.
        parallel_invoke(

```

```

        [&items,lo,m] { parallel_bitonic_sort(items, lo, m, INCREASING); },
        [&items,lo,m] { parallel_bitonic_sort(items, lo + m, m, DECREASING); }
    );

    // Merge the results.
    parallel_bitonic_merge(items, lo, n, dir);
}
}

// Sorts the given sequence in increasing order.
template <class T>
void parallel_bitonic_sort(T* items, int size)
{
    parallel_bitonic_sort(items, 0, size, INCREASING);
}

int wmain()
{
    // For this example, the size must be a power of two.
    const int size = 0x200000;

    // Create two large arrays and fill them with random values.
    int* a1 = new int[size];
    int* a2 = new int[size];

    mt19937 gen(42);
    for(int i = 0; i < size; ++i)
    {
        a1[i] = a2[i] = gen();
    }

    __int64 elapsed;

    // Perform the serial version of the sort.
    elapsed = time_call([&] { bitonic_sort(a1, size); });
    wcout << L"serial time: " << elapsed << endl;

    // Now perform the parallel version of the sort.
    elapsed = time_call([&] { parallel_bitonic_sort(a2, size); });
    wcout << L"parallel time: " << elapsed << endl;

    delete[] a1;
    delete[] a2;
}

```

The following sample output is for a computer that has four processors.

```

serial time: 4353
parallel time: 1248

```

[\[Top\]](#)

## Compiling the Code

To compile the code, copy it and then paste it in a Visual Studio project, or paste it in a file that is named `parallel-bitonic-sort.cpp` and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc parallel-bitonic-sort.cpp**

## Robust Programming

This example uses the `parallel_invoke` algorithm instead of the `concurrency::task_group` class because the lifetime of each task group does not extend beyond a function. We recommend that you use `parallel_invoke`

when you can because it has less execution overhead than `task_group` objects, and therefore lets you write better performing code.

The parallel versions of some algorithms perform better only when there is sufficient work to do. For example, the `parallel_bitonic_merge` function calls the serial version, `bitonic_merge`, if there are 500 or fewer elements in the sequence. You can also plan your overall sorting strategy based on the amount of work. For example, it might be more efficient to use the serial version of the quick sort algorithm if the array contains fewer than 500 items, as shown in the following example:

```
template <class T>
void quick_sort(T* items, int lo, int n)
{
    // TODO: The function body is omitted for brevity.
}

template <class T>
void parallel_bitonic_sort(T* items, int lo, int n, bool dir)
{
    // Use the serial quick sort algorithm if there are relatively few
    // items to sort. The associated overhead for running few tasks in
    // parallel may not overcome the benefits of parallel processing.
    if (n - lo + 1 <= 500)
    {
        quick_sort(items, lo, n);
    }
    else if (n > 1)
    {
        // Divide the array into two partitions and then sort
        // the partitions in different directions.
        int m = n / 2;

        // Sort the partitions in parallel.
        parallel_invoke(
            [&items,lo,m] { parallel_bitonic_sort(items, lo, m, INCREASING); },
            [&items,lo,m] { parallel_bitonic_sort(items, lo + m, m, DECREASING); }
        );

        // Merge the results.
        parallel_bitonic_merge(items, lo, n, dir);
    }
}
```

As with any parallel algorithm, we recommend that you profile and tune your code as appropriate.

## See also

[Task Parallelism](#)

[parallel\\_invoke Function](#)

# How to: Use `parallel_invoke` to Execute Parallel Operations

3/4/2019 • 7 minutes to read • [Edit Online](#)

This example shows how to use the `concurrency::parallel_invoke` algorithm to improve the performance of a program that performs multiple operations on a shared data source. Because no operations modify the source, they can be executed in parallel in a straightforward manner.

## Example

Consider the following code example that creates a variable of type `MyDataType`, calls a function to initialize that variable, and then performs multiple lengthy operations on that data.

```
MyDataType data;
initialize_data(data);

lengthy_operation1(data);
lengthy_operation2(data);
lengthy_operation3(data);
```

If the `lengthy_operation1`, `lengthy_operation2`, and `lengthy_operation3` functions do not modify the `MyDataType` variable, these functions can be executed in parallel without additional modifications.

## Example

The following example modifies the previous example to run in parallel. The `parallel_invoke` algorithm executes each task in parallel and returns after all tasks are finished.

```
MyDataType data;
initialize_data(data);

concurrency::parallel_invoke(
    [&data] { lengthy_operation1(data); },
    [&data] { lengthy_operation2(data); },
    [&data] { lengthy_operation3(data); }
);
```

## Example

The following example downloads *The Iliad* by Homer from [gutenberg.org](http://gutenberg.org) and performs multiple operations on that file. The example first performs these operations serially and then performs the same operations in parallel.

```
// parallel-word-mining.cpp
// compile with: /EHsc /MD /DUNICODE /D_AFXDLL
#define _WIN32_WINNT 0x0501
#include <afxinet.h>
#include <ppl.h>
#include <string>
#include <iostream>
#include <vector>
#include <map>
#include <algorithm>
```

```

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}

// Downloads the file at the given URL.
CString get_http_file(CInternetSession& session, const CString& url);

// Adds each word in the provided string to the provided vector of strings.
void make_word_list(const wstring& text, vector<wstring>& words);

// Finds the most common words whose length are greater than or equal to the
// provided minimum.
vector<pair<wstring, size_t>> find_common_words(const vector<wstring>& words,
    size_t min_length, size_t count);

// Finds the longest sequence of words that have the same first letter.
vector<wstring> find_longest_sequence(const vector<wstring>& words);

// Finds all pairs of palindromes that appear in the provided collection
// of words.
vector<pair<wstring, wstring>> find_palindromes(const vector<wstring>& words,
    size_t min_length);

int wmain()
{
    // Manages the network connection.
    CInternetSession session(L"Microsoft Internet Browser");

    // Download 'The Iliad' from gutenber.org.
    wcout << L"Downloading 'The Iliad'..." << endl;
    wstring file = get_http_file(session, L"http://www.gutenberg.org/files/6130/6130-0.txt");
    wcout << endl;

    // Convert the text to a list of individual words.
    vector<wstring> words;
    make_word_list(file, words);

    // Compare the time that it takes to perform several operations on the data
    // serially and in parallel.
    __int64 elapsed;

    vector<pair<wstring, size_t>> common_words;
    vector<wstring> longest_sequence;
    vector<pair<wstring, wstring>> palindromes;

    wcout << L"Running serial version...";
    elapsed = time_call([&] {
        common_words = find_common_words(words, 5, 9);
        longest_sequence = find_longest_sequence(words);
        palindromes = find_palindromes(words, 5);
    });
    wcout << L" took " << elapsed << L" ms." << endl;

    wcout << L"Running parallel version...";
    elapsed = time_call([&] {
        parallel_invoke(
            [&] { common_words = find_common_words(words, 5, 9); },
            [&] { longest_sequence = find_longest_sequence(words); },
            [&] { palindromes = find_palindromes(words, 5); }
        );
    });
}

```

```

    );
});
wcout << L" took " << elapsed << L" ms." << endl;
wcout << endl;

// Print results.

wcout << L"The most common words that have five or more letters are:"
    << endl;
for_each(begin(common_words), end(common_words),
    [](const pair<wstring, size_t>& p) {
        wcout << L"    " << p.first << L" (" << p.second << L")" << endl;
    });

wcout << L"The longest sequence of words that have the same first letter is:"
    << endl << L"    ";
for_each(begin(longest_sequence), end(longest_sequence),
    [](const wstring& s) {
        wcout << s << L' ';
    });
wcout << endl;

wcout << L"The following palindromes appear in the text:" << endl;
for_each(begin(palindromes), end(palindromes),
    [](const pair<wstring, wstring>& p) {
        wcout << L"    " << p.first << L" " << p.second << endl;
    });
}

// Downloads the file at the given URL.
CString get_http_file(CInternetSession& session, const CString& url)
{
    CString result;

    // Reads data from an HTTP server.
    CHttpFile* http_file = NULL;

    try
    {
        // Open URL.
        http_file = reinterpret_cast<CHttpFile*>(session.OpenURL(url, 1));

        // Read the file.
        if(http_file != NULL)
        {
            UINT bytes_read;
            do
            {
                char buffer[10000];
                bytes_read = http_file->Read(buffer, sizeof(buffer));
                result += buffer;
            }
            while (bytes_read > 0);
        }
    }
    catch (CInternetException)
    {
        // TODO: Handle exception
    }

    // Clean up and return.
    delete http_file;

    return result;
}

// Adds each word in the provided string to the provided vector of strings.
void make_word_list(const wstring& text, vector<wstring>& words)
{

```

```

// Add continuous sequences of alphanumeric characters to the
// string vector.
wstring current_word;
for_each(begin(text), end(text), [&](wchar_t ch) {
    if (!iswalnum(ch))
    {
        if (current_word.length() > 0)
        {
            words.push_back(current_word);
            current_word.clear();
        }
    }
    else
    {
        current_word += ch;
    }
});
}

// Finds the most common words whose length are greater than or equal to the
// provided minimum.
vector<pair<wstring, size_t>> find_common_words(const vector<wstring>& words,
size_t min_length, size_t count)
{
    typedef pair<wstring, size_t> pair;

    // Counts the occurrences of each word.
    map<wstring, size_t> counts;

    for_each(begin(words), end(words), [&](const wstring& word) {
        // Increment the count of words that are at least the minimum length.
        if (word.length() >= min_length)
        {
            auto find = counts.find(word);
            if (find != end(counts))
                find->second++;
            else
                counts.insert(make_pair(word, 1));
        }
    });

    // Copy the contents of the map to a vector and sort the vector by
    // the number of occurrences of each word.
    vector<pair> wordvector;
    copy(begin(counts), end(counts), back_inserter(wordvector));

    sort(begin(wordvector), end(wordvector), [](const pair& x, const pair& y) {
        return x.second > y.second;
    });

    size_t size = min(wordvector.size(), count);
    wordvector.erase(begin(wordvector) + size, end(wordvector));

    return wordvector;
}

// Finds the longest sequence of words that have the same first letter.
vector<wstring> find_longest_sequence(const vector<wstring>& words)
{
    // The current sequence of words that have the same first letter.
    vector<wstring> candidate_list;
    // The longest sequence of words that have the same first letter.
    vector<wstring> longest_run;

    for_each(begin(words), end(words), [&](const wstring& word) {
        // Initialize the candidate list if it is empty.
        if (candidate_list.size() == 0)
        {
            candidate_list.push_back(word);

```

```

        candidate_list.push_back(word);
    }
    // Add the word to the candidate sequence if the first letter
    // of the word is the same as each word in the sequence.
    else if (word[0] == candidate_list[0][0])
    {
        candidate_list.push_back(word);
    }
    // The initial letter has changed; reset the candidate list.
    else
    {
        // Update the longest sequence if needed.
        if (candidate_list.size() > longest_run.size())
            longest_run = candidate_list;

        candidate_list.clear();
        candidate_list.push_back(word);
    }
});

return longest_run;
}

// Finds all pairs of palindromes that appear in the provided collection
// of words.
vector<pair<wstring, wstring>> find_palindromes(const vector<wstring>& words,
size_t min_length)
{
    typedef pair<wstring, wstring> pair;
    vector<pair> result;

    // Copy the words to a new vector object and sort that vector.
    vector<wstring> wordvector;
    copy(begin(words), end(words), back_inserter(wordvector));
    sort(begin(wordvector), end(wordvector));

    // Add each word in the original collection to the result whose palindrome
    // also exists in the collection.
    for_each(begin(words), end(words), [&](const wstring& word) {
        if (word.length() >= min_length)
        {
            wstring rev = word;
            reverse(begin(rev), end(rev));

            if (rev != word && binary_search(begin(wordvector), end(wordvector), rev))
            {
                auto candidate1 = make_pair(word, rev);
                auto candidate2 = make_pair(rev, word);
                if (find(begin(result), end(result), candidate1) == end(result) &&
                    find(begin(result), end(result), candidate2) == end(result))
                    result.push_back(candidate1);
            }
        }
    });

    return result;
}

```

This example produces the following sample output.

```
Downloading 'The Iliad'...

Running serial version... took 953 ms.
Running parallel version... took 656 ms.

The most common words that have five or more letters are:
  their (953)
  shall (444)
  which (431)
  great (398)
  Hector (349)
  Achilles (309)
  through (301)
  these (268)
  chief (259)

The longest sequence of words that have the same first letter is:
  through the tempest to the tented

The following palindromes appear in the text:
  spots stops
  speed deeps
  keels sleek
```

This example uses the `parallel_invoke` algorithm to call multiple functions that act on the same data source. You can use the `parallel_invoke` algorithm to call any set of functions in parallel, not only those that act on the same data.

Because the `parallel_invoke` algorithm calls each work function in parallel, its performance is bounded by the function that takes the longest time to finish (that is, if the runtime processes each function on a separate processor). If this example performs more tasks in parallel than the number of available processors, multiple tasks can run on each processor. In this case, performance is bounded by the group of tasks that takes the longest time to finish.

Because this example performs three tasks in parallel, you should not expect performance to scale on computers that have more than three processors. To improve performance more, you can break the longest-running tasks into smaller tasks and run those tasks in parallel.

You can use the `parallel_invoke` algorithm instead of the `concurrency::task_group` and `concurrency::structured_task_group` classes if you do not require support for cancellation. For an example that compares the usage of the `parallel_invoke` algorithm versus task groups, see [How to: Use parallel\\_invoke to Write a Parallel Sort Routine](#).

## Compiling the Code

To compile the code, copy it and then paste it in a Visual Studio project, or paste it in a file that is named `parallel-word-mining.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc /MD /DUNICODE /D_AFXDLL parallel-word-mining.cpp
```

## See also

[Parallel Algorithms](#)  
[parallel\\_invoke Function](#)

# How to: Create a Task that Completes After a Delay

3/4/2019 • 8 minutes to read • [Edit Online](#)

This example shows how to use the `concurrency::task`, `concurrency::cancellation_token_source`, `concurrency::cancellation_token`, `concurrency::task_completion_event`, `concurrency::timer`, and `concurrency::call` classes to create a task that completes after a delay. You can use this method to build loops that occasionally poll for data, introduce timeouts, delay handling of user input for a predetermined time, and so on.

## Example

The following example shows the `complete_after` and `cancel_after_timeout` functions. The `complete_after` function creates a `task` object that completes after the specified delay. It uses a `timer` object and a `call` object to set a `task_completion_event` object after the specified delay. By using the `task_completion_event` class, you can define a task that completes after a thread or another task signals that a value is available. When the event is set, listener tasks complete and their continuations are scheduled to run.

### TIP

For more information about the `timer` and `call` classes, which are part of the Asynchronous Agents Library, see [Asynchronous Message Blocks](#).

The `cancel_after_timeout` function builds on the `complete_after` function to cancel a task if that task does not complete before a given timeout. The `cancel_after_timeout` function creates two tasks. The first task indicates success and completes after the provided task completes; the second task indicates failure and completes after the specified timeout. The `cancel_after_timeout` function creates a continuation task that runs when the success or failure task completes. If the failure task completes first, the continuation cancels the token source to cancel the overall task.

```

// Creates a task that completes after the specified delay.
task<void> complete_after(unsigned int timeout)
{
    // A task completion event that is set when a timer fires.
    task_completion_event<void> tce;

    // Create a non-repeating timer.
    auto fire_once = new timer<int>(timeout, 0, nullptr, false);
    // Create a call object that sets the completion event after the timer fires.
    auto callback = new call<int>([tce](int)
    {
        tce.set();
    });

    // Connect the timer to the callback and start the timer.
    fire_once->link_target(callback);
    fire_once->start();

    // Create a task that completes after the completion event is set.
    task<void> event_set(tce);

    // Create a continuation task that cleans up resources and
    // and return that continuation task.
    return event_set.then([callback, fire_once]()
    {
        delete callback;
        delete fire_once;
    });
}

// Cancels the provided task after the specified delay, if the task
// did not complete.
template<typename T>
task<T> cancel_after_timeout(task<T> t, cancellation_token_source cts, unsigned int timeout)
{
    // Create a task that returns true after the specified task completes.
    task<bool> success_task = t.then([](T)
    {
        return true;
    });
    // Create a task that returns false after the specified timeout.
    task<bool> failure_task = complete_after(timeout).then([]
    {
        return false;
    });

    // Create a continuation task that cancels the overall task
    // if the timeout task finishes first.
    return (failure_task || success_task).then([t, cts](bool success)
    {
        if(!success)
        {
            // Set the cancellation token. The task that is passed as the
            // t parameter should respond to the cancellation and stop
            // as soon as it can.
            cts.cancel();
        }

        // Return the original task.
        return t;
    });
}

```

## Example

The following example computes the count of prime numbers in the range [0, 100000] multiple times. The operation fails if it does not complete in a given time limit. The `count_primes` function demonstrates how to use the `cancel_after_timeout` function. It counts the number of primes in the given range and fails if the task does not complete in the provided time. The `wmain` function calls the `count_primes` function multiple times. Each time, it halves the time limit. The program finishes after the operation does not complete in the current time limit.

```
// Determines whether the input value is prime.
bool is_prime(int n)
{
    if (n < 2)
        return false;
    for (int i = 2; i < n; ++i)
    {
        if ((n % i) == 0)
            return false;
    }
    return true;
}

// Counts the number of primes in the range [0, max_value].
// The operation fails if it exceeds the specified timeout.
bool count_primes(unsigned int max_value, unsigned int timeout)
{
    cancellation_token_source cts;

    // Create a task that computes the count of prime numbers.
    // The task is canceled after the specified timeout.
    auto t = cancel_after_timeout(task<size_t>([max_value, timeout]
    {
        combinable<size_t> counts;
        parallel_for<unsigned int>(0, max_value + 1, [&counts](unsigned int n)
        {
            // Respond if the overall task is cancelled by canceling
            // the current task.
            if (cts.get_token().is_canceled())
            {
                cancel_current_task();
            }
            // NOTE: You can replace the calls to is_canceled
            // and cancel_current_task with a call to interruption_point.
            // interruption_point();

            // Increment the local counter if the value is prime.
            if (is_prime(n))
            {
                counts.local()++;
            }
        });
        // Return the sum of counts across all threads.
        return counts.combine(plus<size_t>());
    }, cts.get_token(), cts, timeout);

    // Print the result.
    try
    {
        auto primes = t.get();
        wcout << L"Found " << primes << L" prime numbers within "
                << timeout << L" ms." << endl;
        return true;
    }
    catch (const task_canceled& e)
    {
        wcout << L"The task timed out." << endl;
        return false;
    }
}
```

```

int wmain()
{
    // Compute the count of prime numbers in the range [0, 100000]
    // until the operation fails.
    // Each time the test succeeds, the time limit is halved.

    unsigned int max = 100000;
    unsigned int timeout = 5000;

    bool success = true;
    do
    {
        success = count_primes(max, timeout);
        timeout /= 2;
    } while (success);
}
/* Sample output:
   Found 9592 prime numbers within 5000 ms.
   Found 9592 prime numbers within 2500 ms.
   Found 9592 prime numbers within 1250 ms.
   Found 9592 prime numbers within 625 ms.
   The task timed out.
*/

```

When you use this technique to cancel tasks after a delay, any unstarted tasks will not start after the overall task is canceled. However, it is important for any long-running tasks to respond to cancellation in a timely manner. For more information about task cancellation, see [Cancellation in the PPL](#).

## Example

Here is the complete code for this example:

```

// task-delay.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Creates a task that completes after the specified delay.
task<void> complete_after(unsigned int timeout)
{
    // A task completion event that is set when a timer fires.
    task_completion_event<void> tce;

    // Create a non-repeating timer.
    auto fire_once = new timer<int>(timeout, 0, nullptr, false);
    // Create a call object that sets the completion event after the timer fires.
    auto callback = new call<int>([tce](int)
    {
        tce.set();
    });

    // Connect the timer to the callback and start the timer.
    fire_once->link_target(callback);
    fire_once->start();

    // Create a task that completes after the completion event is set.
    task<void> event_set(tce);

    // Create a continuation task that cleans up resources and
    // and return that continuation task.
    return event_set.then([callback, fire_once]()

```

```

    {
        delete callback;
        delete fire_once;
    });
}

// Cancels the provided task after the specified delay, if the task
// did not complete.
template<typename T>
task<T> cancel_after_timeout(task<T> t, cancellation_token_source cts, unsigned int timeout)
{
    // Create a task that returns true after the specified task completes.
    task<bool> success_task = t.then([](T)
    {
        return true;
    });
    // Create a task that returns false after the specified timeout.
    task<bool> failure_task = complete_after(timeout).then([]
    {
        return false;
    });

    // Create a continuation task that cancels the overall task
    // if the timeout task finishes first.
    return (failure_task || success_task).then([t, cts](bool success)
    {
        if(!success)
        {
            // Set the cancellation token. The task that is passed as the
            // t parameter should respond to the cancellation and stop
            // as soon as it can.
            cts.cancel();
        }

        // Return the original task.
        return t;
    });
}

// Determines whether the input value is prime.
bool is_prime(int n)
{
    if (n < 2)
        return false;
    for (int i = 2; i < n; ++i)
    {
        if ((n % i) == 0)
            return false;
    }
    return true;
}

// Counts the number of primes in the range [0, max_value].
// The operation fails if it exceeds the specified timeout.
bool count_primes(unsigned int max_value, unsigned int timeout)
{
    cancellation_token_source cts;

    // Create a task that computes the count of prime numbers.
    // The task is canceled after the specified timeout.
    auto t = cancel_after_timeout(task<size_t>([max_value, timeout]
    {
        combinable<size_t> counts;
        parallel_for<unsigned int>(0, max_value + 1, [&counts](unsigned int n)
        {
            // Respond if the overall task is cancelled by canceling
            // the current task.
            if (cts.get_token().is_canceled())
                r

```

```

        cancel_current_task();
    }
    // NOTE: You can replace the calls to is_canceled
    // and cancel_current_task with a call to interruption_point.
    // interruption_point();

    // Increment the local counter if the value is prime.
    if (is_prime(n))
    {
        counts.local()++;
    }
});
// Return the sum of counts across all threads.
return counts.combine(plus<size_t>());
}, cts.get_token(), cts, timeout);

// Print the result.
try
{
    auto primes = t.get();
    wcout << L"Found " << primes << L" prime numbers within "
        << timeout << L" ms." << endl;
    return true;
}
catch (const task_canceled& e)
{
    wcout << L"The task timed out." << endl;
    return false;
}
}

int wmain()
{
    // Compute the count of prime numbers in the range [0, 100000]
    // until the operation fails.
    // Each time the test succeeds, the time limit is halved.

    unsigned int max = 100000;
    unsigned int timeout = 5000;

    bool success = true;
    do
    {
        success = count_primes(max, timeout);
        timeout /= 2;
    } while (success);
}
/* Sample output:
   Found 9592 prime numbers within 5000 ms.
   Found 9592 prime numbers within 2500 ms.
   Found 9592 prime numbers within 1250 ms.
   Found 9592 prime numbers within 625 ms.
   The task timed out.
*/

```

## Compiling the Code

To compile the code, copy it and then paste it in a Visual Studio project, or paste it in a file that is named `task-delay.cpp` and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc task-delay.cpp**

## See also

[Task Parallelism](#)

task Class (Concurrency Runtime)

cancellation\_token\_source Class

cancellation\_token Class

task\_completion\_event Class

timer Class

call Class

Asynchronous Message Blocks

Cancellation in the PPL

# Parallel Algorithms

3/4/2019 • 26 minutes to read • [Edit Online](#)

The Parallel Patterns Library (PPL) provides algorithms that concurrently perform work on collections of data. These algorithms resemble those provided by the C++ Standard Library.

The parallel algorithms are composed from existing functionality in the Concurrency Runtime. For example, the `concurrency::parallel_for` algorithm uses a `concurrency::structured_task_group` object to perform the parallel loop iterations. The `parallel_for` algorithm partitions work in an optimal way given the available number of computing resources.

## Sections

- [The parallel\\_for Algorithm](#)
- [The parallel\\_for\\_each Algorithm](#)
- [The parallel\\_invoke Algorithm](#)
- [The parallel\\_transform and parallel\\_reduce Algorithms](#)
  - [The parallel\\_transform Algorithm](#)
  - [The parallel\\_reduce Algorithm](#)
  - [Example: Performing Map and Reduce in Parallel](#)
- [Partitioning Work](#)
- [Parallel Sorting](#)
  - [Choosing a Sorting Algorithm](#)

## The parallel\_for Algorithm

The `concurrency::parallel_for` algorithm repeatedly performs the same task in parallel. Each of these tasks is parameterized by an iteration value. This algorithm is useful when you have a loop body that does not share resources among iterations of that loop.

The `parallel_for` algorithm partitions tasks in an optimum way for parallel execution. It uses a work-stealing algorithm and range stealing to balance these partitions when workloads are unbalanced. When one loop iteration blocks cooperatively, the runtime redistributes the range of iterations that is assigned to the current thread to other threads or processors. Similarly, when a thread completes a range of iterations, the runtime redistributes work from other threads to that thread. The `parallel_for` algorithm also supports *nested parallelism*. When one parallel loop contains another parallel loop, the runtime coordinates processing resources between the loop bodies in an efficient way for parallel execution.

The `parallel_for` algorithm has several overloaded versions. The first version takes a start value, an end value, and a work function (a lambda expression, function object, or function pointer). The second version takes a start value, an end value, a value by which to step, and a work function. The first version of this function uses 1 as the step value. The remaining versions take partitioner objects, which enable you to specify how `parallel_for` should partition ranges among threads. Partitioners are explained in greater detail in the section [Partitioning Work](#) in this document.

You can convert many `for` loops to use `parallel_for`. However, the `parallel_for` algorithm differs from the `for` statement in the following ways:

- The `parallel_for` algorithm `parallel_for` does not execute the tasks in a pre-determined order.
- The `parallel_for` algorithm does not support arbitrary termination conditions. The `parallel_for` algorithm stops when the current value of the iteration variable is one less than `last`.
- The `_Index_type` type parameter must be an integral type. This integral type can be signed or unsigned.
- The loop iteration must be forward. The `parallel_for` algorithm throws an exception of type `std::invalid_argument` if the `_Step` parameter is less than 1.
- The exception-handling mechanism for the `parallel_for` algorithm differs from that of a `for` loop. If multiple exceptions occur simultaneously in a parallel loop body, the runtime propagates only one of the exceptions to the thread that called `parallel_for`. In addition, when one loop iteration throws an exception, the runtime does not immediately stop the overall loop. Instead, the loop is placed in the cancelled state and the runtime discards any tasks that have not yet started. For more information about exception-handling and parallel algorithms, see [Exception Handling](#).

Although the `parallel_for` algorithm does not support arbitrary termination conditions, you can use cancellation to stop all tasks. For more information about cancellation, see [Cancellation in the PPL](#).

#### NOTE

The scheduling cost that results from load balancing and support for features such as cancellation might not overcome the benefits of executing the loop body in parallel, especially when the loop body is relatively small. You can minimize this overhead by using a partitioner in your parallel loop. For more information, see [Partitioning Work](#) later in this document.

#### Example

The following example shows the basic structure of the `parallel_for` algorithm. This example prints to the console each value in the range [1, 5] in parallel.

```
// parallel-for-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <array>
#include <sstream>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Print each value from 1 to 5 in parallel.
    parallel_for(1, 6, [](int value) {
        wstringstream ss;
        ss << value << L' ';
        wcout << ss.str();
    });
}
```

This example produces the following sample output:

```
1 2 4 3 5
```

Because the `parallel_for` algorithm acts on each item in parallel, the order in which the values are printed to the console will vary.

For a complete example that uses the `parallel_for` algorithm, see [How to: Write a parallel\\_for Loop](#).

[\[Top\]](#)

## The `parallel_for_each` Algorithm

The `concurrency::parallel_for_each` algorithm performs tasks on an iterative container, such as those provided by the C++ Standard Library, in parallel. It uses the same partitioning logic that the `parallel_for` algorithm uses.

The `parallel_for_each` algorithm resembles the C++ Standard Library `std::for_each` algorithm, except that the `parallel_for_each` algorithm executes the tasks concurrently. Like other parallel algorithms, `parallel_for_each` does not execute the tasks in a specific order.

Although the `parallel_for_each` algorithm works on both forward iterators and random access iterators, it performs better with random access iterators.

### Example

The following example shows the basic structure of the `parallel_for_each` algorithm. This example prints to the console each value in a `std::array` object in parallel.

```
// parallel-for-each-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <array>
#include <sstream>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create an array of integer values.
    array<int, 5> values = { 1, 2, 3, 4, 5 };

    // Print each value in the array in parallel.
    parallel_for_each(begin(values), end(values), [](int value) {
        wstringstream ss;
        ss << value << L' ';
        wcout << ss.str();
    });
}
/* Sample output:
5 4 3 1 2
*/
```

This example produces the following sample output:

```
4 5 1 2 3
```

Because the `parallel_for_each` algorithm acts on each item in parallel, the order in which the values are printed to the console will vary.

For a complete example that uses the `parallel_for_each` algorithm, see [How to: Write a parallel\\_for\\_each Loop](#).

[\[Top\]](#)

## The `parallel_invoke` Algorithm

The `concurrency::parallel_invoke` algorithm executes a set of tasks in parallel. It does not return until each task finishes. This algorithm is useful when you have several independent tasks that you want to execute at the same time.

The `parallel_invoke` algorithm takes as its parameters a series of work functions (lambda functions, function objects, or function pointers). The `parallel_invoke` algorithm is overloaded to take between two and ten parameters. Every function that you pass to `parallel_invoke` must take zero parameters.

Like other parallel algorithms, `parallel_invoke` does not execute the tasks in a specific order. The topic [Task Parallelism](#) explains how the `parallel_invoke` algorithm relates to tasks and task groups.

### Example

The following example shows the basic structure of the `parallel_invoke` algorithm. This example concurrently calls the `twice` function on three local variables and prints the result to the console.

```
// parallel_invoke-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <string>
#include <iostream>

using namespace concurrency;
using namespace std;

// Returns the result of adding a value to itself.
template <typename T>
T twice(const T& t) {
    return t + t;
}

int wmain()
{
    // Define several values.
    int n = 54;
    double d = 5.6;
    wstring s = L"Hello";

    // Call the twice function on each value concurrently.
    parallel_invoke(
        [&n] { n = twice(n); },
        [&d] { d = twice(d); },
        [&s] { s = twice(s); }
    );

    // Print the values to the console.
    wcout << n << L' ' << d << L' ' << s << endl;
}
```

This example produces the following output:

```
108 11.2 HelloHello
```

For complete examples that use the `parallel_invoke` algorithm, see [How to: Use parallel\\_invoke to Write a](#)

[\[Top\]](#)

## The parallel\_transform and parallel\_reduce Algorithms

The `concurrency::parallel_transform` and `concurrency::parallel_reduce` algorithms are parallel versions of the C++ Standard Library algorithms `std::transform` and `std::accumulate`, respectively. The Concurrency Runtime versions behave like the C++ Standard Library versions except that the operation order is not determined because they execute in parallel. Use these algorithms when you work with a set that is large enough to get performance and scalability benefits from being processed in parallel.

### IMPORTANT

The `parallel_transform` and `parallel_reduce` algorithms support only random access, bi-directional, and forward iterators because these iterators produce stable memory addresses. Also, these iterators must produce non-`const` l-values.

### The parallel\_transform Algorithm

You can use the `parallel_transform` algorithm to perform many data parallelization operations. For example, you can:

- Adjust the brightness of an image, and perform other image processing operations.
- Sum or take the dot product between two vectors, and perform other numeric calculations on vectors.
- Perform 3-D ray tracing, where each iteration refers to one pixel that must be rendered.

The following example shows the basic structure that is used to call the `parallel_transform` algorithm. This example negates each element of a `std::vector` object in two ways. The first way uses a lambda expression. The second way uses `std::negate`, which derives from `std::unary_function`.

```

// basic-parallel-transform.cpp
// compile with: /EHsc
#include <ppl.h>
#include <random>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create a large vector that contains random integer data.
    vector<int> values(1250000);
    generate(begin(values), end(values), mt19937(42));

    // Create a vector to hold the results.
    // Depending on your requirements, you can also transform the
    // vector in-place.
    vector<int> results(values.size());

    // Negate each element in parallel.
    parallel_transform(begin(values), end(values), begin(results), [](int n) {
        return -n;
    });

    // Alternatively, use the negate class to perform the operation.
    parallel_transform(begin(values), end(values), begin(values), negate<int>());
}

```

#### WARNING

This example demonstrates the basic use of `parallel_transform`. Because the work function does not perform a significant amount of work, a significant increase in performance is not expected in this example.

The `parallel_transform` algorithm has two overloads. The first overload takes one input range and a unary function. The unary function can be a lambda expression that takes one argument, a function object, or a type that derives from `unary_function`. The second overload takes two input ranges and a binary function. The binary function can be a lambda expression that takes two arguments, a function object, or a type that derives from `std::binary_function`. The following example illustrates these differences.

```

//
// Demonstrate use of parallel_transform together with a unary function.

// This example uses a lambda expression.
parallel_transform(begin(values), end(values),
    begin(results), [](int n) {
        return -n;
    });

// Alternatively, use the negate class:
parallel_transform(begin(values), end(values),
    begin(results), negate<int>());

//
// Demonstrate use of parallel_transform together with a binary function.

// This example uses a lambda expression.
parallel_transform(begin(values), end(values), begin(results),
    begin(results), [](int n, int m) {
        return n * m;
    });

// Alternatively, use the multiplies class:
parallel_transform(begin(values), end(values), begin(results),
    begin(results), multiplies<int>());

```

### IMPORTANT

The iterator that you supply for the output of `parallel_transform` must completely overlap the input iterator or not overlap at all. The behavior of this algorithm is unspecified if the input and output iterators partially overlap.

### The parallel\_reduce Algorithm

The `parallel_reduce` algorithm is useful when you have a sequence of operations that satisfy the associative property. (This algorithm does not require the commutative property.) Here are some of the operations that you can perform with `parallel_reduce`:

- Multiply sequences of matrices to produce a matrix.
- Multiply a vector by a sequence of matrices to produce a vector.
- Compute the length of a sequence of strings.
- Combine a list of elements, such as strings, into one element.

The following basic example shows how to use the `parallel_reduce` algorithm to combine a sequence of strings into one string. As with the examples for `parallel_transform`, performance gains are not expected in this basic example.

```

// basic-parallel-reduce.cpp
// compile with: /EHsc
#include <ppl.h>
#include <iostream>
#include <string>
#include <vector>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create a vector of strings.
    vector<wstring> words{
        L"Lorem ",
        L"ipsum ",
        L"dolor ",
        L"sit ",
        L"amet, ",
        L"consectetur ",
        L"adipiscing ",
        L"elit."};

    // Reduce the vector to one string in parallel.
    wcout << parallel_reduce(begin(words), end(words), wstring()) << endl;
}

/* Output:
   Lorem ipsum dolor sit amet, consectetur adipiscing elit.
*/

```

In many cases, you can think of `parallel_reduce` as shorthand for the use of the `parallel_for_each` algorithm together with the `concurrency::combinable` class.

### Example: Performing Map and Reduce in Parallel

A *map* operation applies a function to each value in a sequence. A *reduce* operation combines the elements of a sequence into one value. You can use the C++ Standard Library `std::transform` and `std::accumulate` functions to perform map and reduce operations. However, for many problems, you can use the `parallel_transform` algorithm to perform the map operation in parallel and the `parallel_reduce` algorithm perform the reduce operation in parallel.

The following example compares the time that it takes to compute the sum of prime numbers serially and in parallel. The map phase transforms non-prime values to 0 and the reduce phase sums the values.

```

// parallel-map-reduce-sum-of-primes.cpp
// compile with: /EHsc
#include <windows.h>
#include <ppl.h>
#include <array>
#include <numeric>
#include <iostream>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}

```

```

}

// Determines whether the input value is prime.
bool is_prime(int n)
{
    if (n < 2)
        return false;
    for (int i = 2; i < n; ++i)
    {
        if ((n % i) == 0)
            return false;
    }
    return true;
}

int wmain()
{
    // Create an array object that contains 200000 integers.
    array<int, 200000> a;

    // Initialize the array such that a[i] == i.
    iota(begin(a), end(a), 0);

    int prime_sum;
    __int64 elapsed;

    // Compute the sum of the numbers in the array that are prime.
    elapsed = time_call([&] {
        transform(begin(a), end(a), begin(a), [](int i) {
            return is_prime(i) ? i : 0;
        });
        prime_sum = accumulate(begin(a), end(a), 0);
    });
    wcout << prime_sum << endl;
    wcout << L"serial time: " << elapsed << L" ms" << endl << endl;

    // Now perform the same task in parallel.
    elapsed = time_call([&] {
        parallel_transform(begin(a), end(a), begin(a), [](int i) {
            return is_prime(i) ? i : 0;
        });
        prime_sum = parallel_reduce(begin(a), end(a), 0);
    });
    wcout << prime_sum << endl;
    wcout << L"parallel time: " << elapsed << L" ms" << endl << endl;
}

/* Sample output:
1709600813
serial time: 7406 ms

1709600813
parallel time: 1969 ms
*/

```

For another example that performs a map and reduce operation in parallel, see [How to: Perform Map and Reduce Operations in Parallel](#).

[\[Top\]](#)

## Partitioning Work

To parallelize an operation on a data source, an essential step is to *partition* the source into multiple sections that can be accessed concurrently by multiple threads. A partitioner specifies how a parallel algorithm should partition ranges among threads. As explained previously in this document, the PPL uses a default partitioning mechanism that creates an initial workload and then uses a work-stealing algorithm and range

stealing to balance these partitions when workloads are unbalanced. For example, when one loop iteration completes a range of iterations, the runtime redistributes work from other threads to that thread. However, for some scenarios, you might want to specify a different partitioning mechanism that is better suited to your problem.

The `parallel_for`, `parallel_for_each`, and `parallel_transform` algorithms provide overloaded versions that take an additional parameter, `_Partitioner`. This parameter defines the partitioner type that divides work. Here are the kinds of partitioners that the PPL defines:

#### `concurrency::affinity_partitioner`

Divides work into a fixed number of ranges (typically the number of worker threads that are available to work on the loop). This partitioner type resembles `static_partitioner`, but improves cache affinity by the way it maps ranges to worker threads. This partitioner type can improve performance when a loop is executed over the same data set multiple times (such as a loop within a loop) and the data fits in cache. This partitioner does not fully participate in cancellation. It also does not use cooperative blocking semantics and therefore cannot be used with parallel loops that have a forward dependency.

#### `concurrency::auto_partitioner`

Divides work into an initial number of ranges (typically the number of worker threads that are available to work on the loop). The runtime uses this type by default when you do not call an overloaded parallel algorithm that takes a `_Partitioner` parameter. Each range can be divided into sub-ranges, and thereby enables load balancing to occur. When a range of work completes, the runtime redistributes sub-ranges of work from other threads to that thread. Use this partitioner if your workload does not fall under one of the other categories or you require full support for cancellation or cooperative blocking.

#### `concurrency::simple_partitioner`

Divides work into ranges such that each range has at least the number of iterations that are specified by the given chunk size. This partitioner type participates in load balancing; however, the runtime does not divide ranges into sub-ranges. For each worker, the runtime checks for cancellation and performs load-balancing after `_Chunk_size` iterations complete.

#### `concurrency::static_partitioner`

Divides work into a fixed number of ranges (typically the number of worker threads that are available to work on the loop). This partitioner type can improve performance because it does not use work-stealing and therefore has less overhead. Use this partitioner type when each iteration of a parallel loop performs a fixed and uniform amount of work and you do not require support for cancellation or forward cooperative blocking.

#### **WARNING**

The `parallel_for_each` and `parallel_transform` algorithms support only containers that use random access iterators (such as `std::vector`) for the static, simple, and affinity partitioners. The use of containers that use bidirectional and forward iterators produces a compile-time error. The default partitioner, `auto_partitioner`, supports all three of these iterator types.

Typically, these partitioners are used in the same way, except for `affinity_partitioner`. Most partitioner types do not maintain state and are not modified by the runtime. Therefore you can create these partitioner objects at the call site, as shown in the following example.

```

// static-partitioner.cpp
// compile with: /EHsc
#include <ppl.h>

using namespace concurrency;

void DoWork(int n)
{
    // TODO: Perform a fixed amount of work...
}

int wmain()
{
    // Use a static partitioner to perform a fixed amount of parallel work.
    parallel_for(0, 100000, [](int n) {
        DoWork(n);
    }, static_partitioner());
}

```

However, you must pass an `affinity_partitioner` object as a non-`const`, l-value reference so that the algorithm can store state for future loops to reuse. The following example shows a basic application that performs the same operation on a data set in parallel multiple times. The use of `affinity_partitioner` can improve performance because the array is likely to fit in cache.

```

// affinity-partitioner.cpp
// compile with: /EHsc
#include <ppl.h>
#include <array>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create an array and fill it with zeroes.
    array<unsigned char, 8 * 1024> data;
    data.fill(0);

    // Use an affinity partitioner to perform parallel work on data
    // that is likely to remain in cache.
    // We use the same affinity partitioner throughout so that the
    // runtime can schedule work to occur at the same location for each
    // iteration of the outer loop.

    affinity_partitioner ap;
    for (int i = 0; i < 100000; i++)
    {
        parallel_for_each(begin(data), end(data), [](unsigned char& c)
        {
            c++;
        }, ap);
    }
}

```

#### Caution

Use caution when you modify existing code that relies on cooperative blocking semantics to use `static_partitioner` or `affinity_partitioner`. These partitioner types do not use load balancing or range stealing, and therefore can alter the behavior of your application.

The best way to determine whether to use a partitioner in any given scenario is to experiment and measure how long it takes operations to complete under representative loads and computer configurations. For example, static partitioning might provide significant speedup on a multi-core computer that has only a few

cores, but it might result in slowdowns on computers that have relatively many cores.

[\[Top\]](#)

## Parallel Sorting

The PPL provides three sorting algorithms: `concurrency::parallel_sort`, `concurrency::parallel_buffered_sort`, and `concurrency::parallel_radixsort`. These sorting algorithms are useful when you have a data set that can benefit from being sorted in parallel. In particular, sorting in parallel is useful when you have a large dataset or when you use a computationally-expensive compare operation to sort your data. Each of these algorithms sorts elements in place.

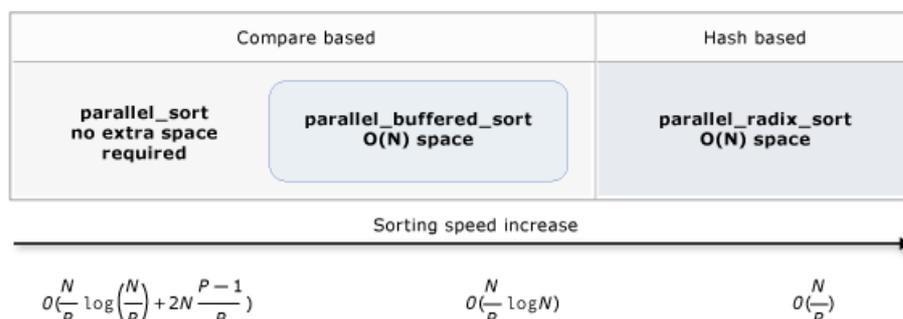
The `parallel_sort` and `parallel_buffered_sort` algorithms are both compare-based algorithms. That is, they compare elements by value. The `parallel_sort` algorithm has no additional memory requirements, and is suitable for general-purpose sorting. The `parallel_buffered_sort` algorithm can perform better than `parallel_sort`, but it requires  $O(N)$  space.

The `parallel_radixsort` algorithm is hash-based. That is, it uses integer keys to sort elements. By using keys, this algorithm can directly compute the destination of an element instead of using comparisons. Like `parallel_buffered_sort`, this algorithm requires  $O(N)$  space.

The following table summarizes the important properties of the three parallel sorting algorithms.

ALGORITHM	DESCRIPTION	SORTING MECHANISM	SORT STABILITY	MEMORY REQUIREMENTS	TIME COMPLEXITY	ITERATOR ACCESS
<code>parallel_sort</code>	General-purpose compare-based sort.	Compare-based (ascending)	Unstable	None	$O((N/P)\log(N/P) + 2N((P-1)/P))$	Random
<code>parallel_buffered_sort</code>	Fast general-purpose compare-based sort that requires $O(N)$ space.	Compare-based (ascending)	Unstable	Requires additional $O(N)$ space	$O((N/P)\log(N))$	Random
<code>parallel_radixsort</code>	Integer key-based sort that requires $O(N)$ space.	Hash-based	Stable	Requires additional $O(N)$ space	$O(N/P)$	Random

The following illustration shows the important properties of the three parallel sorting algorithms more graphically.



These parallel sorting algorithms follow the rules of cancellation and exception handling. For more

information about cancellation and exception handling in the Concurrency Runtime, see [Canceling Parallel Algorithms](#) and [Exception Handling](#).

#### TIP

These parallel sorting algorithms support move semantics. You can define a move assignment operator to enable swap operations to occur more efficiently. For more information about move semantics and the move assignment operator, see [Rvalue Reference Declarator: &&](#), and [Move Constructors and Move Assignment Operators \(C++\)](#). If you do not provide a move assignment operator or swap function, the sorting algorithms use the copy constructor.

The following basic example shows how to use `parallel_sort` to sort a `vector` of `int` values. By default, `parallel_sort` uses `std::less` to compare values.

```
// basic-parallel-sort.cpp
// compile with: /EHsc
#include <ppl.h>
#include <random>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create and sort a large vector of random values.
    vector<int> values(25000000);
    generate(begin(values), end(values), mt19937(42));
    parallel_sort(begin(values), end(values));

    // Print a few values.
    wcout << "V(0)          = " << values[0] << endl;
    wcout << "V(12500000) = " << values[12500000] << endl;
    wcout << "V(24999999) = " << values[24999999] << endl;
}
/* Output:
V(0)          = -2147483129
V(12500000) = -427327
V(24999999) = 2147483311
*/
```

This example shows how to provide a custom compare function. It uses the `std::complex::real` method to sort `std::complex<double>` values in ascending order.

```

// For this example, ensure that you add the following #include directive:
// #include <complex>

// Create and sort a large vector of random values.
vector<complex<double>> values(25000000);
generate(begin(values), end(values), mt19937(42));
parallel_sort(begin(values), end(values),
    [](const complex<double>& left, const complex<double>& right) {
        return left.real() < right.real();
    });

// Print a few values.
wcout << "V(0)          = " << values[0] << endl;
wcout << "V(12500000) = " << values[12500000] << endl;
wcout << "V(24999999) = " << values[24999999] << endl;
/* Output:
V(0)          = (383,0)
V(12500000) = (2.1479e+009,0)
V(24999999) = (4.29497e+009,0)
*/

```

This example shows how to provide a hash function to the `parallel_radixsort` algorithm. This example sorts 3-D points. The points are sorted based on their distance from a reference point.

```

// parallel-sort-points.cpp
// compile with: /EHsc
#include <ppl.h>
#include <random>
#include <iostream>

using namespace concurrency;
using namespace std;

// Defines a 3-D point.
struct Point
{
    int X;
    int Y;
    int Z;
};

// Computes the Euclidean distance between two points.
size_t euclidean_distance(const Point& p1, const Point& p2)
{
    int dx = p1.X - p2.X;
    int dy = p1.Y - p2.Y;
    int dz = p1.Z - p2.Z;
    return static_cast<size_t>(sqrt((dx*dx) + (dy*dy) + (dz*dz)));
}

int wmain()
{
    // The central point of reference.
    const Point center = { 3, 2, 7 };

    // Create a few random Point values.
    vector<Point> values(7);
    mt19937 random(42);
    generate(begin(values), end(values), [&random] {
        Point p = { random()%10, random()%10, random()%10 };
        return p;
    });

    // Print the values before sorting them.
    wcout << "Before sorting:" << endl;
    for each(begin(values), end(values), [center](const Point& p) {

```

```

for_each(begin(values), end(values), [center](const Point& p) {
    wcout << L'(' << p.X << L", " << p.Y << L", " << p.Z
        << L") D = " << euclidean_distance(p, center) << endl;
});
wcout << endl;

// Sort the values based on their distances from the reference point.
parallel_radixsort(begin(values), end(values),
    [center](const Point& p) -> size_t {
        return euclidean_distance(p, center);
    });

// Print the values after sorting them.
wcout << "After sorting:" << endl;
for_each(begin(values), end(values), [center](const Point& p) {
    wcout << L'(' << p.X << L", " << p.Y << L", " << p.Z
        << L") D = " << euclidean_distance(p, center) << endl;
});
wcout << endl;
}
/* Output:
Before sorting:
(2,7,6) D = 5
(4,6,5) D = 4
(0,4,0) D = 7
(3,8,4) D = 6
(0,4,1) D = 7
(2,5,5) D = 3
(7,6,9) D = 6

After sorting:
(2,5,5) D = 3
(4,6,5) D = 4
(2,7,6) D = 5
(3,8,4) D = 6
(7,6,9) D = 6
(0,4,0) D = 7
(0,4,1) D = 7
*/

```

For illustration, this example uses a relatively small data set. You can increase the initial size of the vector to experiment with performance improvements over larger sets of data.

This example uses a lambda expression as the hash function. You can also use one of the built-in implementations of the `std::hash class` or define your own specialization. You can also use a custom hash function object, as shown in this example:

```

// Functor class for computing the distance between points.
class hash_distance
{
public:
    hash_distance(const Point& reference)
        : m_reference(reference)
    {
    }

    size_t operator()(const Point& pt) const {
        return euclidean_distance(pt, m_reference);
    }

private:
    Point m_reference;
};

```

```
// Use hash_distance to compute the distance between points.
parallel_radixsort(begin(values), end(values), hash_distance(center));
```

The hash function must return an integral type (`std::is_integral::value` must be **true**). This integral type must be convertible to type `size_t`.

### Choosing a Sorting Algorithm

In many cases, `parallel_sort` provides the best balance of speed and memory performance. However, as you increase the size of your data set, the number of available processors, or the complexity of your compare function, `parallel_buffered_sort` or `parallel_radixsort` can perform better. The best way to determine which sorting algorithm to use in any given scenario is to experiment and measure how long it takes to sort typical data under representative computer configurations. Keep the following guidelines in mind when you choose a sorting strategy.

- The size of your data set. In this document, a *small* dataset contains fewer than 1,000 elements, a *medium* dataset contains between 10,000 and 100,000 elements, and a *large* dataset contains more than 100,000 elements.
- The amount of work that your compare function or hash function performs.
- The amount of available computing resources.
- The characteristics of your data set. For example, one algorithm might perform well for data that is already nearly sorted, but not as well for data that is completely unsorted.
- The chunk size. The optional `_Chunk_size` argument specifies when the algorithm switches from a parallel to a serial sort implementation as it subdivides the overall sort into smaller units of work. For example, if you provide 512, the algorithm switches to serial implementation when a unit of work contains 512 or fewer elements. A serial implementation can improve overall performance because it eliminates the overhead that is required to process data in parallel.

It might not be worthwhile to sort a small dataset in parallel, even when you have a large number of available computing resources or your compare function or hash function performs a relatively large amount of work. You can use `std::sort` function to sort small datasets. (`parallel_sort` and `parallel_buffered_sort` call `sort` when you specify a chunk size that is larger than the dataset; however, `parallel_buffered_sort` would have to allocate  $O(N)$  space, which could take additional time due to lock contention or memory allocation.)

If you must conserve memory or your memory allocator is subject to lock contention, use `parallel_sort` to sort a medium-sized dataset. `parallel_sort` requires no additional space; the other algorithms require  $O(N)$  space.

Use `parallel_buffered_sort` to sort medium-sized datasets and when your application meets the additional  $O(N)$  space requirement. `parallel_buffered_sort` can be especially useful when you have a large number of computing resources or an expensive compare function or hash function.

Use `parallel_radixsort` to sort large datasets and when your application meets the additional  $O(N)$  space requirement. `parallel_radixsort` can be especially useful when the equivalent compare operation is more expensive or when both operations are expensive.

#### Caution

Implementing a good hash function requires that you know the dataset range and how each element in the dataset is transformed to a corresponding unsigned value. Because the hash operation works on unsigned values, consider a different sorting strategy if unsigned hash values cannot be produced.

The following example compares the performance of `sort`, `parallel_sort`, `parallel_buffered_sort`, and

parallel\_radixsort against the same large set of random data.

```
// choosing-parallel-sort.cpp
// compile with: /EHsc
#include <ppl.h>
#include <random>
#include <iostream>
#include <windows.h>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}

const size_t DATASET_SIZE = 10000000;

// Create
// Creates the dataset for this example. Each call
// produces the same predefined sequence of random data.
vector<size_t> GetData()
{
    vector<size_t> data(DATASET_SIZE);
    generate(begin(data), end(data), mt19937(42));
    return data;
}

int wmain()
{
    // Use std::sort to sort the data.
    auto data = GetData();
    wcout << L"Testing std::sort...";
    auto elapsed = time_call([&data] { sort(begin(data), end(data)); });
    wcout << L" took " << elapsed << L" ms." <<endl;

    // Use concurrency::parallel_sort to sort the data.
    data = GetData();
    wcout << L"Testing concurrency::parallel_sort...";
    elapsed = time_call([&data] { parallel_sort(begin(data), end(data)); });
    wcout << L" took " << elapsed << L" ms." <<endl;

    // Use concurrency::parallel_buffered_sort to sort the data.
    data = GetData();
    wcout << L"Testing concurrency::parallel_buffered_sort...";
    elapsed = time_call([&data] { parallel_buffered_sort(begin(data), end(data)); });
    wcout << L" took " << elapsed << L" ms." <<endl;

    // Use concurrency::parallel_radixsort to sort the data.
    data = GetData();
    wcout << L"Testing concurrency::parallel_radixsort...";
    elapsed = time_call([&data] { parallel_radixsort(begin(data), end(data)); });
    wcout << L" took " << elapsed << L" ms." <<endl;
}
/* Sample output (on a computer that has four cores):
   Testing std::sort... took 2906 ms.
   Testing concurrency::parallel_sort... took 2234 ms.
   Testing concurrency::parallel_buffered_sort... took 1782 ms.
   Testing concurrency::parallel_radixsort... took 907 ms.
*/
```

In this example, which assumes that it is acceptable to allocate  $O(N)$  space during the sort, `parallel_radixsort` performs the best on this dataset on this computer configuration.

[\[Top\]](#)

## Related Topics

TITLE	DESCRIPTION
<a href="#">How to: Write a <code>parallel_for</code> Loop</a>	Shows how to use the <code>parallel_for</code> algorithm to perform matrix multiplication.
<a href="#">How to: Write a <code>parallel_for_each</code> Loop</a>	Shows how to use the <code>parallel_for_each</code> algorithm to compute the count of prime numbers in a <code>std::array</code> object in parallel.
<a href="#">How to: Use <code>parallel_invoke</code> to Write a Parallel Sort Routine</a>	Shows how to use the <code>parallel_invoke</code> algorithm to improve the performance of the bitonic sort algorithm.
<a href="#">How to: Use <code>parallel_invoke</code> to Execute Parallel Operations</a>	Shows how to use the <code>parallel_invoke</code> algorithm to improve the performance of a program that performs multiple operations on a shared data source.
<a href="#">How to: Perform Map and Reduce Operations in Parallel</a>	Shows how to use the <code>parallel_transform</code> and <code>parallel_reduce</code> algorithms to perform a map and reduce operation that counts the occurrences of words in files.
<a href="#">Parallel Patterns Library (PPL)</a>	Describes the PPL, which provides an imperative programming model that promotes scalability and ease-of-use for developing concurrent applications.
<a href="#">Cancellation in the PPL</a>	Explains the role of cancellation in the PPL, how to cancel parallel work, and how to determine when a task group is canceled.
<a href="#">Exception Handling</a>	Explains the role of exception handling in the Concurrency Runtime.

## Reference

[parallel\\_for Function](#)

[parallel\\_for\\_each Function](#)

[parallel\\_invoke Function](#)

[affinity\\_partitioner Class](#)

[auto\\_partitioner Class](#)

[simple\\_partitioner Class](#)

[static\\_partitioner Class](#)

[parallel\\_sort Function](#)

[parallel\\_buffered\\_sort Function](#)

parallel\_radixsort Function

# How to: Write a `parallel_for` Loop

3/4/2019 • 4 minutes to read • [Edit Online](#)

This example demonstrates how to use `concurrency::parallel_for` to compute the product of two matrices.

## Example

The following example shows the `matrix_multiply` function, which computes the product of two square matrices.

```
// Computes the product of two square matrices.
void matrix_multiply(double** m1, double** m2, double** result, size_t size)
{
    for (size_t i = 0; i < size; i++)
    {
        for (size_t j = 0; j < size; j++)
        {
            double temp = 0;
            for (int k = 0; k < size; k++)
            {
                temp += m1[i][k] * m2[k][j];
            }
            result[i][j] = temp;
        }
    }
}
```

## Example

The following example shows the `parallel_matrix_multiply` function, which uses the `parallel_for` algorithm to perform the outer loop in parallel.

```
// Computes the product of two square matrices in parallel.
void parallel_matrix_multiply(double** m1, double** m2, double** result, size_t size)
{
    parallel_for (size_t(0), size, [&](size_t i)
    {
        for (size_t j = 0; j < size; j++)
        {
            double temp = 0;
            for (int k = 0; k < size; k++)
            {
                temp += m1[i][k] * m2[k][j];
            }
            result[i][j] = temp;
        }
    });
}
```

This example parallelizes the outer loop only because it performs enough work to benefit from the overhead for parallel processing. If you parallelize the inner loop, you will not receive a gain in performance because the small amount of work that the inner loop performs does not overcome the overhead for parallel processing. Therefore, parallelizing the outer loop only is the best way to maximize the benefits of concurrency on most systems.

## Example

The following more complete example compares the performance of the `matrix_multiply` function versus the `parallel_matrix_multiply` function.

```
// parallel-matrix-multiply.cpp
// compile with: /EHsc
#include <windows.h>
#include <ppl.h>
#include <iostream>
#include <random>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}

// Creates a square matrix with the given number of rows and columns.
double** create_matrix(size_t size);

// Frees the memory that was allocated for the given square matrix.
void destroy_matrix(double** m, size_t size);

// Initializes the given square matrix with values that are generated
// by the given generator function.
template <class Generator>
double** initialize_matrix(double** m, size_t size, Generator& gen);

// Computes the product of two square matrices.
void matrix_multiply(double** m1, double** m2, double** result, size_t size)
{
    for (size_t i = 0; i < size; i++)
    {
        for (size_t j = 0; j < size; j++)
        {
            double temp = 0;
            for (int k = 0; k < size; k++)
            {
                temp += m1[i][k] * m2[k][j];
            }
            result[i][j] = temp;
        }
    }
}

// Computes the product of two square matrices in parallel.
void parallel_matrix_multiply(double** m1, double** m2, double** result, size_t size)
{
    parallel_for (size_t(0), size, [&](size_t i)
    {
        for (size_t j = 0; j < size; j++)
        {
            double temp = 0;
            for (int k = 0; k < size; k++)
            {
                temp += m1[i][k] * m2[k][j];
            }
            result[i][j] = temp;
        }
    });
}
```

```

int wmain()
{
    // The number of rows and columns in each matrix.
    // TODO: Change this value to experiment with serial
    // versus parallel performance.
    const size_t size = 750;

    // Create a random number generator.
    mt19937 gen(42);

    // Create and initialize the input matrices and the matrix that
    // holds the result.
    double** m1 = initialize_matrix(create_matrix(size), size, gen);
    double** m2 = initialize_matrix(create_matrix(size), size, gen);
    double** result = create_matrix(size);

    // Print to the console the time it takes to multiply the
    // matrices serially.
    wcout << L"serial: " << time_call([&] {
        matrix_multiply(m1, m2, result, size);
    }) << endl;

    // Print to the console the time it takes to multiply the
    // matrices in parallel.
    wcout << L"parallel: " << time_call([&] {
        parallel_matrix_multiply(m1, m2, result, size);
    }) << endl;

    // Free the memory that was allocated for the matrices.
    destroy_matrix(m1, size);
    destroy_matrix(m2, size);
    destroy_matrix(result, size);
}

// Creates a square matrix with the given number of rows and columns.
double** create_matrix(size_t size)
{
    double** m = new double*[size];
    for (size_t i = 0; i < size; ++i)
    {
        m[i] = new double[size];
    }
    return m;
}

// Frees the memory that was allocated for the given square matrix.
void destroy_matrix(double** m, size_t size)
{
    for (size_t i = 0; i < size; ++i)
    {
        delete[] m[i];
    }
    delete m;
}

// Initializes the given square matrix with values that are generated
// by the given generator function.
template <class Generator>
double** initialize_matrix(double** m, size_t size, Generator& gen)
{
    for (size_t i = 0; i < size; ++i)
    {
        for (size_t j = 0; j < size; ++j)
        {
            m[i][j] = static_cast<double>(gen());
        }
    }
    return m;
}

```

```
}
```

The following sample output is for a computer that has four processors.

```
serial: 3853  
parallel: 1311
```

## Compiling the Code

To compile the code, copy it and then paste it in a Visual Studio project, or paste it in a file that is named `parallel-matrix-multiply.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc parallel-matrix-multiply.cpp
```

## See also

[Parallel Algorithms](#)

[parallel\\_for Function](#)

# How to: Write a parallel\_for\_each Loop

3/4/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use the `concurrency::parallel_for_each` algorithm to compute the count of prime numbers in a `std::array` object in parallel.

## Example

The following example computes the count of prime numbers in an array two times. The example first uses the `std::for_each` algorithm to compute the count serially. The example then uses the `parallel_for_each` algorithm to perform the same task in parallel. The example also prints to the console the time that is required to perform both computations.

```
// parallel-count-primes.cpp
// compile with: /EHsc
#include <windows.h>
#include <ppl.h>
#include <iostream>
#include <algorithm>
#include <array>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}

// Determines whether the input value is prime.
bool is_prime(int n)
{
    if (n < 2)
        return false;
    for (int i = 2; i < n; ++i)
    {
        if ((n % i) == 0)
            return false;
    }
    return true;
}

int wmain()
{
    // Create an array object that contains 200000 integers.
    array<int, 200000> a;

    // Initialize the array such that a[i] == i.
    int n = 0;
    generate(begin(a), end(a), [&] {
        return n++;
    });

    LONG prime_count;
    __int64 elapsed;
```

```

// Use the for_each algorithm to count the number of prime numbers
// in the array serially.
prime_count = 0L;
elapsed = time_call([&] {
    for_each (begin(a), end(a), [&](int n ) {
        if (is_prime(n))
            ++prime_count;
    });
});
wcout << L"serial version: " << endl
    << L"found " << prime_count << L" prime numbers" << endl
    << L"took " << elapsed << L" ms" << endl << endl;

// Use the parallel_for_each algorithm to count the number of prime numbers
// in the array in parallel.
prime_count = 0L;
elapsed = time_call([&] {
    parallel_for_each (begin(a), end(a), [&](int n ) {
        if (is_prime(n))
            InterlockedIncrement(&prime_count);
    });
});
wcout << L"parallel version: " << endl
    << L"found " << prime_count << L" prime numbers" << endl
    << L"took " << elapsed << L" ms" << endl << endl;
}

```

The following sample output is for a computer that has four processors.

```

serial version:
found 17984 prime numbers
took 6115 ms

parallel version:
found 17984 prime numbers
took 1653 ms

```

## Compiling the Code

To compile the code, copy it and then paste it in a Visual Studio project, or paste it in a file that is named `parallel-count-primes.cpp` and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc parallel-count-primes.cpp**

## Robust Programming

The lambda expression that the example passes to the `parallel_for_each` algorithm uses the `InterlockedIncrement` function to enable parallel iterations of the loop to increment the counter simultaneously. If you use functions such as `InterlockedIncrement` to synchronize access to shared resources, you can present performance bottlenecks in your code. You can use a lock-free synchronization mechanism, for example, the [concurrency::combinable](#) class, to eliminate simultaneous access to shared resources. For an example that uses the `combinable` class in this manner, see [How to: Use combinable to Improve Performance](#).

## See also

[Parallel Algorithms](#)  
[parallel\\_for\\_each Function](#)

# How to: Perform Map and Reduce Operations in Parallel

3/4/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use the `concurrency::parallel_transform` and `concurrency::parallel_reduce` algorithms and the `concurrency::concurrent_unordered_map` class to count the occurrences of words in files.

A *map* operation applies a function to each value in a sequence. A *reduce* operation combines the elements of a sequence into one value. You can use the C++ Standard Library `std::transform` and `std::accumulate` functions to perform map and reduce operations. However, to improve performance for many problems, you can use the `parallel_transform` algorithm to perform the map operation in parallel and the `parallel_reduce` algorithm to perform the reduce operation in parallel. In some cases, you can use `concurrent_unordered_map` to perform the map and the reduce in one operation.

## Example

The following example counts the occurrences of words in files. It uses `std::vector` to represent the contents of two files. The map operation computes the occurrences of each word in each vector. The reduce operation accumulates the word counts across both vectors.

```
// parallel-map-reduce.cpp
// compile with: /EHsc
#include <ppl.h>
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <numeric>
#include <unordered_map>
#include <windows.h>

using namespace concurrency;
using namespace std;

class MapFunc
{
public:
    unordered_map<wstring, size_t> operator()(vector<wstring>& elements) const
    {
        unordered_map<wstring, size_t> m;
        for_each(begin(elements), end(elements), [&m](const wstring& elem)
        {
            m[elem]++;
        });
        return m;
    }
};

struct ReduceFunc : binary_function<unordered_map<wstring, size_t>,
    unordered_map<wstring, size_t>, unordered_map<wstring, size_t>>
{
    unordered_map<wstring, size_t> operator() (
        const unordered_map<wstring, size_t>& x,
        const unordered_map<wstring, size_t>& y) const
    {
        unordered_map<wstring, size_t> ret(x);
        for_each(begin(y), end(y), [&ret](const pair<wstring, size_t>& pr) {
```

```

        auto key = pr.first;
        auto val = pr.second;
        ret[key] += val;
    });
    return ret;
}
};

int wmain()
{
    // File 1
    vector<wstring> v1 {
        L"word1", // 1
        L"word1", // 1
        L"word2",
        L"word3",
        L"word4"
    };

    // File 2
    vector<wstring> v2 {
        L"word5",
        L"word6",
        L"word7",
        L"word8",
        L"word1" // 3
    };

    vector<vector<wstring>> v { v1, v2 };

    vector<unordered_map<wstring, size_t>> map(v.size());

    // The Map operation
    parallel_transform(begin(v), end(v), begin(map), MapFunc());

    // The Reduce operation
    unordered_map<wstring, size_t> result = parallel_reduce(
        begin(map), end(map), unordered_map<wstring, size_t>(), ReduceFunc());

    wcout << L"\word1\" occurs " << result.at(L"word1") << L" times. " << endl;
}
/* Output:
"word1" occurs 3 times.
*/

```

## Compiling the Code

To compile the code, copy it and then paste it in a Visual Studio project, or paste it in a file that is named `parallel-map-reduce.cpp` and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc parallel-map-reduce.cpp**

## Robust Programming

In this example, you can use the `concurrent_unordered_map` class—which is defined in `concurrent_unordered_map.h`—to perform the map and reduce in one operation.

```

// File 1
vector<wstring> v1 {
    L"word1", // 1
    L"word1", // 2
    L"word2",
    L"word3",
    L"word4",
};

// File 2
vector<wstring> v2 {
    L"word5",
    L"word6",
    L"word7",
    L"word8",
    L"word1", // 3
};

vector<vector<wstring>> v { v1, v2 };

concurrent_unordered_map<wstring, size_t> result;
for_each(begin(v), end(v), [&result](const vector<wstring>& words) {
    parallel_for_each(begin(words), end(words), [&result](const wstring& word) {
        InterlockedIncrement(&result[word]);
    });
});

wcout << L"\\"word1\\" occurs " << result.at(L"word1") << L" times. " << endl;

/* Output:
   "word1" occurs 3 times.
*/

```

Typically, you parallelize only the outer or the inner loop. Parallelize the inner loop if you have relatively few files and each file contains many words. Parallelize the outer loop if you have relatively many files and each file contains few words.

## See also

[Parallel Algorithms](#)

[parallel\\_transform Function](#)

[parallel\\_reduce Function](#)

[concurrent\\_unordered\\_map Class](#)

# Parallel Containers and Objects

3/28/2019 • 13 minutes to read • [Edit Online](#)

The Parallel Patterns Library (PPL) includes several containers and objects that provide thread-safe access to their elements.

A *concurrent container* provides concurrency-safe access to the most important operations. The functionality of these containers resembles those that are provided by the C++ Standard Library. For example, the `concurrency::concurrent_vector` class resembles the `std::vector` class, except that the `concurrent_vector` class lets you append elements in parallel. Use concurrent containers when you have parallel code that requires both read and write access to the same container.

A *concurrent object* is shared concurrently among components. A process that computes the state of a concurrent object in parallel produces the same result as another process that computes the same state serially. The `concurrency::combinable` class is one example of a concurrent object type. The `combinable` class lets you perform computations in parallel, and then combine those computations into a final result. Use concurrent objects when you would otherwise use a synchronization mechanism, for example, a mutex, to synchronize access to a shared variable or resource.

## Sections

This topic describes the following parallel containers and objects in detail.

Concurrent containers:

- [concurrent\\_vector Class](#)
  - [Differences Between concurrent\\_vector and vector](#)
  - [Concurrency-Safe Operations](#)
  - [Exception Safety](#)
- [concurrent\\_queue Class](#)
  - [Differences Between concurrent\\_queue and queue](#)
  - [Concurrency-Safe Operations](#)
  - [Iterator Support](#)
- [concurrent\\_unordered\\_map Class](#)
  - [Differences Between concurrent\\_unordered\\_map and unordered\\_map](#)
  - [Concurrency-Safe Operations](#)
- [concurrent\\_unordered\\_multimap Class](#)
- [concurrent\\_unordered\\_set Class](#)
- [concurrent\\_unordered\\_multiset Class](#)

Concurrent objects:

- [combinable Class](#)

- [Methods and Features](#)
- [Examples](#)

## concurrent\_vector Class

The `concurrency::concurrent_vector` class is a sequence container class that, just like the `std::vector` class, lets you randomly access its elements. The `concurrent_vector` class enables concurrency-safe append and element access operations. Append operations do not invalidate existing pointers or iterators. Iterator access and traversal operations are also concurrency-safe.

### Differences Between `concurrent_vector` and `vector`

The `concurrent_vector` class closely resembles the `vector` class. The complexity of append, element access, and iterator access operations on a `concurrent_vector` object are the same as for a `vector` object. The following points illustrate where `concurrent_vector` differs from `vector`:

- Append, element access, iterator access, and iterator traversal operations on a `concurrent_vector` object are concurrency-safe.
- You can add elements only to the end of a `concurrent_vector` object. The `concurrent_vector` class does not provide the `insert` method.
- A `concurrent_vector` object does not use [move semantics](#) when you append to it.
- The `concurrent_vector` class does not provide the `erase` or `pop_back` methods. As with `vector`, use the `clear` method to remove all elements from a `concurrent_vector` object.
- The `concurrent_vector` class does not store its elements contiguously in memory. Therefore, you cannot use the `concurrent_vector` class in all the ways that you can use an array. For example, for a variable named `v` of type `concurrent_vector`, the expression `&v[0]+2` produces undefined behavior.
- The `concurrent_vector` class defines the `grow_by` and `grow_to_at_least` methods. These methods resemble the `resize` method, except that they are concurrency-safe.
- A `concurrent_vector` object does not relocate its elements when you append to it or resize it. This enables existing pointers and iterators to remain valid during concurrent operations.
- The runtime does not define a specialized version of `concurrent_vector` for type `bool`.

### Concurrency-Safe Operations

All methods that append to or increase the size of a `concurrent_vector` object, or access an element in a `concurrent_vector` object, are concurrency-safe. The exception to this rule is the `resize` method.

The following table shows the common `concurrent_vector` methods and operators that are concurrency-safe.

<code>at</code>	<code>end</code>	<code>operator[]</code>
<code>begin</code>	<code>front</code>	<code>push_back</code>
<code>back</code>	<code>grow_by</code>	<code>rbegin</code>
<code>capacity</code>	<code>grow_to_at_least</code>	<code>rend</code>
<code>empty</code>	<code>max_size</code>	<code>size</code>

Operations that the runtime provides for compatibility with the C++ Standard Library, for example, `reserve`, are not concurrency-safe. The following table shows the common methods and operators that are not concurrency-safe.

<code>assign</code>	<code>reserve</code>
<code>clear</code>	<code>resize</code>
<code>operator=</code>	<code>shrink_to_fit</code>

Operations that modify the value of existing elements are not concurrency-safe. Use a synchronization object such as a `reader_writer_lock` object to synchronize concurrent read and write operations to the same data element. For more information about synchronization objects, see [Synchronization Data Structures](#).

When you convert existing code that uses `vector` to use `concurrent_vector`, concurrent operations can cause the behavior of your application to change. For example, consider the following program that concurrently performs two tasks on a `concurrent_vector` object. The first task appends additional elements to a `concurrent_vector` object. The second task computes the sum of all elements in the same object.

```
// parallel-vector-sum.cpp
// compile with: /EHsc
#include <ppl.h>
#include <concurrent_vector.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create a concurrent_vector object that contains a few
    // initial elements.
    concurrent_vector<int> v;
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);

    // Perform two tasks in parallel.
    // The first task appends additional elements to the concurrent_vector object.
    // The second task computes the sum of all elements in the same object.

    parallel_invoke(
        [&v] {
            for(int i = 0; i < 10000; ++i)
            {
                v.push_back(i);
            }
        },
        [&v] {
            combinable<int> sums;
            for(auto i = begin(v); i != end(v); ++i)
            {
                sums.local() += *i;
            }
            wcout << L"sum = " << sums.combine(plus<int>()) << endl;
        }
    );
}
```

Although the `end` method is concurrency-safe, a concurrent call to the `push_back` method causes the value

that is returned by `end` to change. The number of elements that the iterator traverses is indeterminate. Therefore, this program can produce a different result each time that you run it.

### Exception Safety

If a growth or assignment operation throws an exception, the state of the `concurrent_vector` object becomes invalid. The behavior of a `concurrent_vector` object that is in an invalid state is undefined unless stated otherwise. However, the destructor always frees the memory that the object allocates, even if the object is in an invalid state.

The data type of the vector elements, `T`, must meet the following requirements. Otherwise, the behavior of the `concurrent_vector` class is undefined.

- The destructor must not throw.
- If the default or copy constructor throws, the destructor must not be declared by using the `virtual` keyword and it must work correctly with zero-initialized memory.

[\[Top\]](#)

## concurrent\_queue Class

The `concurrency::concurrent_queue` class, just like the `std::queue` class, lets you access its front and back elements. The `concurrent_queue` class enables concurrency-safe enqueue and dequeue operations. The `concurrent_queue` class also provides iterator support that is not concurrency-safe.

### Differences Between `concurrent_queue` and `queue`

The `concurrent_queue` class closely resembles the `queue` class. The following points illustrate where `concurrent_queue` differs from `queue`:

- Enqueue and dequeue operations on a `concurrent_queue` object are concurrency-safe.
- The `concurrent_queue` class provides iterator support that is not concurrency-safe.
- The `concurrent_queue` class does not provide the `front` or `pop` methods. The `concurrent_queue` class replaces these methods by defining the `try_pop` method.
- The `concurrent_queue` class does not provide the `back` method. Therefore, you cannot reference the end of the queue.
- The `concurrent_queue` class provides the `unsafe_size` method instead of the `size` method. The `unsafe_size` method is not concurrency-safe.

### Concurrency-Safe Operations

All methods that enqueue to or dequeue from a `concurrent_queue` object are concurrency-safe.

The following table shows the common `concurrent_queue` methods and operators that are concurrency-safe.

<code>empty</code>	<code>push</code>
<code>get_allocator</code>	<code>try_pop</code>

Although the `empty` method is concurrency-safe, a concurrent operation may cause the queue to grow or shrink before the `empty` method returns.

The following table shows the common methods and operators that are not concurrency-safe.

<code>clear</code>	<code>unsafe_end</code>
<code>unsafe_begin</code>	<code>unsafe_size</code>

## Iterator Support

The `concurrent_queue` provides iterators that are not concurrency-safe. We recommend that you use these iterators for debugging only.

A `concurrent_queue` iterator traverses elements in the forward direction only. The following table shows the operators that each iterator supports.

OPERATOR	DESCRIPTION
<code>operator++</code>	Advances to next item in the queue. This operator is overloaded to provide both pre-increment and post-increment semantics.
<code>operator*</code>	Retrieves a reference to the current item.
<code>operator-&gt;</code>	Retrieves a pointer to the current item.

[\[Top\]](#)

## concurrent\_unordered\_map Class

The `concurrency::concurrent_unordered_map` class is an associative container class that, just like the `std::unordered_map` class, controls a varying-length sequence of elements of type `std::pair<const Key, Ty>`. Think of an unordered map as a dictionary that you can add a key and value pair to or look up a value by key. This class is useful when you have multiple threads or tasks that have to concurrently access a shared container, insert into it, or update it.

The following example shows the basic structure for using `concurrent_unordered_map`. This example inserts character keys in the range ['a', 'i']. Because the order of operations is undetermined, the final value for each key is also undetermined. However, it is safe to perform the insertions in parallel.

```

// unordered-map-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <concurrent_unordered_map.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    //
    // Insert a number of items into the map in parallel.

    concurrent_unordered_map<char, int> map;

    parallel_for(0, 1000, [&map](int i) {
        char key = 'a' + (i%9); // Generate a key in the range [a,i].
        int value = i;         // Set the value to i.
        map.insert(make_pair(key, value));
    });

    // Print the elements in the map.
    for_each(begin(map), end(map), [](const pair<char, int>& pr) {
        wcout << L "[" << pr.first << L", " << pr.second << L "]" ";
    });
}
/* Sample output:
[e, 751] [i, 755] [a, 756] [c, 758] [g, 753] [f, 752] [b, 757] [d, 750] [h, 754]
*/

```

For an example that uses `concurrent_unordered_map` to perform a map and reduce operation in parallel, see [How to: Perform Map and Reduce Operations in Parallel](#).

### Differences Between `concurrent_unordered_map` and `unordered_map`

The `concurrent_unordered_map` class closely resembles the `unordered_map` class. The following points illustrate where `concurrent_unordered_map` differs from `unordered_map`:

- The `erase`, `bucket`, `bucket_count`, and `bucket_size` methods are named `unsafe_erase`, `unsafe_bucket`, `unsafe_bucket_count`, and `unsafe_bucket_size`, respectively. The `unsafe_` naming convention indicates that these methods are not concurrency-safe. For more information about concurrency safety, see [Concurrency-Safe Operations](#).
- Insert operations do not invalidate existing pointers or iterators, nor do they change the order of items that already exist in the map. Insert and traverse operations can occur concurrently.
- `concurrent_unordered_map` supports forward iteration only.
- Insertion does not invalidate or update the iterators that are returned by `equal_range`. Insertion can append unequal items to the end of the range. The begin iterator points to an equal item.

To help avoid deadlock, no method of `concurrent_unordered_map` holds a lock when it calls the memory allocator, hash functions, or other user-defined code. Also, you must ensure that the hash function always evaluates equal keys to the same value. The best hash functions distribute keys uniformly across the hash code space.

### Concurrency-Safe Operations

The `concurrent_unordered_map` class enables concurrency-safe insert and element-access operations. Insert operations do not invalidate existing pointers or iterators. Iterator access and traversal operations are also concurrency-safe. The following table shows the commonly used `concurrent_unordered_map` methods and

operators that are concurrency-safe.

<code>at</code>	<code>count</code>	<code>find</code>	<code>key_eq</code>
<code>begin</code>	<code>empty</code>	<code>get_allocator</code>	<code>max_size</code>
<code>cbegin</code>	<code>end</code>	<code>hash_function</code>	<code>operator[]</code>
<code>cend</code>	<code>equal_range</code>	<code>insert</code>	<code>size</code>

Although the `count` method can be called safely from concurrently running threads, different threads can receive different results if a new value is simultaneously inserted into the container.

The following table shows the commonly used methods and operators that are not concurrency-safe.

<code>clear</code>	<code>max_load_factor</code>	<code>rehash</code>
<code>load_factor</code>	<code>operator=</code>	

In addition to these methods, any method that begins with `unsafe_` is also not concurrency-safe.

[\[Top\]](#)

## concurrent\_unordered\_multimap Class

The `concurrency::concurrent_unordered_multimap` class closely resembles the `concurrent_unordered_map` class except that it allows for multiple values to map to the same key. It also differs from `concurrent_unordered_map` in the following ways:

- The `concurrent_unordered_multimap::insert` method returns an iterator instead of `std::pair<iterator, bool>`.
- The `concurrent_unordered_multimap` class does not provide `operator[]` nor the `at` method.

The following example shows the basic structure for using `concurrent_unordered_multimap`. This example inserts character keys in the range ['a', 'i']. `concurrent_unordered_multimap` enables a key to have multiple values.

```

// unordered-multimap-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <concurrent_unordered_map.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    //
    // Insert a number of items into the map in parallel.

    concurrent_unordered_multimap<char, int> map;

    parallel_for(0, 10, [&map](int i) {
        char key = 'a' + (i%9); // Generate a key in the range [a,i].
        int value = i;         // Set the value to i.
        map.insert(make_pair(key, value));
    });

    // Print the elements in the map.
    for_each(begin(map), end(map), [](const pair<char, int>& pr) {
        wcout << L "[" << pr.first << L", " << pr.second << L "]" << " ";
    });
}
/* Sample output:
[e, 4] [i, 8] [a, 9] [a, 0] [c, 2] [g, 6] [f, 5] [b, 1] [d, 3] [h, 7]
*/

```

[\[Top\]](#)

## concurrent\_unordered\_set Class

The `concurrency::concurrent_unordered_set` class closely resembles the `concurrent_unordered_map` class except that it manages values instead of key and value pairs. The `concurrent_unordered_set` class does not provide `operator[]` nor the `at` method.

The following example shows the basic structure for using `concurrent_unordered_set`. This example inserts character values in the range ['a', 'i']. It is safe to perform the insertions in parallel.

```

// unordered-set-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <concurrent_unordered_set.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    //
    // Insert a number of items into the set in parallel.

    concurrent_unordered_set<char> set;

    parallel_for(0, 10000, [&set](int i) {
        set.insert('a' + (i%9)); // Generate a value in the range [a,i].
    });

    // Print the elements in the set.
    for_each(begin(set), end(set), [](char c) {
        wcout << L "[" << c << L " ] ";
    });
}
/* Sample output:
[e] [i] [a] [c] [g] [f] [b] [d] [h]
*/

```

[\[Top\]](#)

## concurrent\_unordered\_multiset Class

The `concurrency::concurrent_unordered_multiset` class closely resembles the `concurrent_unordered_set` class except that it allows for duplicate values. It also differs from `concurrent_unordered_set` in the following ways:

- The `concurrent_unordered_multiset::insert` method returns an iterator instead of `std::pair<iterator, bool>`.
- The `concurrent_unordered_multiset` class does not provide `operator[]` nor the `at` method.

The following example shows the basic structure for using `concurrent_unordered_multiset`. This example inserts character values in the range ['a', 'i']. `concurrent_unordered_multiset` enables a value to occur multiple times.

```

// unordered-set-structure.cpp
// compile with: /EHsc
#include <ppl.h>
#include <concurrent_unordered_set.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    //
    // Insert a number of items into the set in parallel.

    concurrent_unordered_multiset<char> set;

    parallel_for(0, 40, [&set](int i) {
        set.insert('a' + (i%9)); // Generate a value in the range [a,i].
    });

    // Print the elements in the set.
    for_each(begin(set), end(set), [](char c) {
        wcout << L "[" << c << L " ] ";
    });
}
/* Sample output:
[e] [e] [e] [e] [i] [i] [i] [i] [a] [a] [a] [a] [a] [c] [c] [c] [c] [c] [g] [g]
[g] [g] [f] [f] [f] [f] [b] [b] [b] [b] [b] [d] [d] [d] [d] [h] [h] [h] [h]
*/

```

[\[Top\]](#)

## combinable Class

The `concurrency::combinable` class provides reusable, thread-local storage that lets you perform fine-grained computations and then merge those computations into a final result. You can think of a `combinable` object as a reduction variable.

The `combinable` class is useful when you have a resource that is shared among several threads or tasks. The `combinable` class helps you eliminate shared state by providing access to shared resources in a lock-free manner. Therefore, this class provides an alternative to using a synchronization mechanism, for example, a mutex, to synchronize access to shared data from multiple threads.

### Methods and Features

The following table shows some of the important methods of the `combinable` class. For more information about all the `combinable` class methods, see [combinable Class](#).

METHOD	DESCRIPTION
<a href="#">local</a>	Retrieves a reference to the local variable that is associated with the current thread context.
<a href="#">clear</a>	Removes all thread-local variables from the <code>combinable</code> object.
<a href="#">combine</a> <a href="#">combine_each</a>	Uses the provided combine function to generate a final value from the set of all thread-local computations.

The `combinable` class is a template class that is parameterized on the final merged result. If you call the default constructor, the `T` template parameter type must have a default constructor and a copy constructor. If the `T` template parameter type does not have a default constructor, call the overloaded version of the constructor that takes an initialization function as its parameter.

You can store additional data in a `combinable` object after you call the `combine` or `combine_each` methods. You can also call the `combine` and `combine_each` methods multiple times. If no local value in a `combinable` object changes, the `combine` and `combine_each` methods produce the same result every time that they are called.

## Examples

For examples about how to use the `combinable` class, see the following topics:

- [How to: Use combinable to Improve Performance](#)
- [How to: Use combinable to Combine Sets](#)

[\[Top\]](#)

## Related Topics

[How to: Use Parallel Containers to Increase Efficiency](#)

Shows how to use parallel containers to efficiently store and access data in parallel.

[How to: Use combinable to Improve Performance](#)

Shows how to use the `combinable` class to eliminate shared state, and thereby improve performance.

[How to: Use combinable to Combine Sets](#)

Shows how to use a `combine` function to merge thread-local sets of data.

[Parallel Patterns Library \(PPL\)](#)

Describes the PPL, which provides an imperative programming model that promotes scalability and ease-of-use for developing concurrent applications.

## Reference

[concurrent\\_vector Class](#)

[concurrent\\_queue Class](#)

[concurrent\\_unordered\\_map Class](#)

[concurrent\\_unordered\\_multimap Class](#)

[concurrent\\_unordered\\_set Class](#)

[concurrent\\_unordered\\_multiset Class](#)

[combinable Class](#)

# How to: Use Parallel Containers to Increase Efficiency

3/4/2019 • 6 minutes to read • [Edit Online](#)

This topic shows how to use parallel containers to efficiently store and access data in parallel.

The example code computes the set of prime and Carmichael numbers in parallel. Then, for each Carmichael number, the code computes the prime factors of that number.

## Example

The following example shows the `is_prime` function, which determines whether an input value is a prime number, and the `is_carmichael` function, which determines whether the input value is a Carmichael number.

```
// Determines whether the input value is prime.
bool is_prime(int n)
{
    if (n < 2)
        return false;
    for (int i = 2; i < n; ++i)
    {
        if ((n % i) == 0)
            return false;
    }
    return true;
}

// Determines whether the input value is a Carmichael number.
bool is_carmichael(const int n)
{
    if (n < 2)
        return false;

    int k = n;
    for (int i = 2; i <= k / i; ++i)
    {
        if (k % i == 0)
        {
            if ((k / i) % i == 0)
                return false;
            if ((n - 1) % (i - 1) != 0)
                return false;
            k /= i;
            i = 1;
        }
    }
    return k != n && (n - 1) % (k - 1) == 0;
}
```

## Example

The following example uses the `is_prime` and `is_carmichael` functions to compute the sets of prime and Carmichael numbers. The example uses the `concurrency::parallel_invoke` and `concurrency::parallel_for` algorithms to compute each set in parallel. For more information about parallel algorithms, see [Parallel Algorithms](#).

This example uses a `concurrency::concurrent_queue` object to hold the set of Carmichael numbers because it will later use that object as a work queue. It uses a `concurrency::concurrent_vector` object to hold the set of prime numbers because it will later iterate through this set to find prime factors.

```

// The maximum number to test.
const int max = 10000000;

// Holds the Carmichael numbers that are in the range [0, max).
concurrent_queue<int> carmichaels;

// Holds the prime numbers that are in the range [0, sqrt(max)).
concurrent_vector<int> primes;

// Generate the set of Carmichael numbers and the set of prime numbers
// in parallel.
parallel_invoke(
    [&] {
        parallel_for(0, max, [&](int i) {
            if (is_carmichael(i)) {
                carmichaels.push(i);
            }
        });
    },
    [&] {
        parallel_for(0, int(sqrt(static_cast<double>(max))), [&](int i) {
            if (is_prime(i)) {
                primes.push_back(i);
            }
        });
    });
};

```

## Example

The following example shows the `prime_factors_of` function, which uses trial division to find all prime factors of the given value.

This function uses the `concurrency::parallel_for_each` algorithm to iterate through the collection of prime numbers. The `concurrent_vector` object enables the parallel loop to concurrently add prime factors to the result.

```

// Finds all prime factors of the given value.
concurrent_vector<int> prime_factors_of(int n,
    const concurrent_vector<int>& primes)
{
    // Holds the prime factors of n.
    concurrent_vector<int> prime_factors;

    // Use trial division to find the prime factors of n.
    // Every prime number that divides evenly into n is a prime factor of n.
    const int max = sqrt(static_cast<double>(n));
    parallel_for_each(begin(primes), end(primes), [&](int prime)
    {
        if (prime <= max)
        {
            if ((n % prime) == 0)
                prime_factors.push_back(prime);
        }
    });

    return prime_factors;
}

```

## Example

This example processes each element in the queue of Carmichael numbers by calling the `prime_factors_of` function to compute its prime factors. It uses a task group to perform this work in parallel. For more information

about task groups, see [Task Parallelism](#).

This example prints the prime factors for each Carmichael number if that number has more than four prime factors.

```
// Use a task group to compute the prime factors of each
// Carmichael number in parallel.
task_group tasks;

int carmichael;
while (carmichaels.try_pop(carmichael))
{
    tasks.run([carmichael,&primes]
    {
        // Compute the prime factors.
        auto prime_factors = prime_factors_of(carmichael, primes);

        // For brevity, print the prime factors for the current number only
        // if there are more than 4.
        if (prime_factors.size() > 4)
        {
            // Sort and then print the prime factors.
            sort(begin(prime_factors), end(prime_factors));

            wstringstream ss;
            ss << L"Prime factors of " << carmichael << L" are:";

            for_each (begin(prime_factors), end(prime_factors),
                [&](int prime_factor) { ss << L' ' << prime_factor; });
            ss << L'.' << endl;

            wcout << ss.str();
        }
    });
}

// Wait for the task group to finish.
tasks.wait();
```

## Example

The following code shows the complete example, which uses parallel containers to compute the prime factors of the Carmichael numbers.

```
// carmichael-primes.cpp
// compile with: /EHsc
#include <ppl.h>
#include <concurrent_queue.h>
#include <concurrent_vector.h>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

// Determines whether the input value is prime.
bool is_prime(int n)
{
    if (n < 2)
        return false;
    for (int i = 2; i < n; ++i)
    {
        if ((n % i) == 0)
            return false;
    }
}
```

```

    }
    return true;
}

// Determines whether the input value is a Carmichael number.
bool is_carmichael(const int n)
{
    if (n < 2)
        return false;

    int k = n;
    for (int i = 2; i <= k / i; ++i)
    {
        if (k % i == 0)
        {
            if ((k / i) % i == 0)
                return false;
            if ((n - 1) % (i - 1) != 0)
                return false;
            k /= i;
            i = 1;
        }
    }
    return k != n && (n - 1) % (k - 1) == 0;
}

// Finds all prime factors of the given value.
concurrent_vector<int> prime_factors_of(int n,
    const concurrent_vector<int>& primes)
{
    // Holds the prime factors of n.
    concurrent_vector<int> prime_factors;

    // Use trial division to find the prime factors of n.
    // Every prime number that divides evenly into n is a prime factor of n.
    const int max = sqrt(static_cast<double>(n));
    parallel_for_each(begin(primes), end(primes), [&](int prime)
    {
        if (prime <= max)
        {
            if ((n % prime) == 0)
                prime_factors.push_back(prime);
        }
    });

    return prime_factors;
}

int wmain()
{
    // The maximum number to test.
    const int max = 10000000;

    // Holds the Carmichael numbers that are in the range [0, max).
    concurrent_queue<int> carmichaels;

    // Holds the prime numbers that are in the range [0, sqrt(max)).
    concurrent_vector<int> primes;

    // Generate the set of Carmichael numbers and the set of prime numbers
    // in parallel.
    parallel_invoke(
        [&] {
            parallel_for(0, max, [&](int i) {
                if (is_carmichael(i)) {
                    carmichaels.push(i);
                }
            });
        },

```

```

[&] {
    parallel_for(0, int(sqrt(static_cast<double>(max))), [&](int i) {
        if (is_prime(i)) {
            primes.push_back(i);
        }
    });
});

// Use a task group to compute the prime factors of each
// Carmichael number in parallel.
task_group tasks;

int carmichael;
while (carmichaels.try_pop(carmichael))
{
    tasks.run([carmichael,&primes]
    {
        // Compute the prime factors.
        auto prime_factors = prime_factors_of(carmichael, primes);

        // For brevity, print the prime factors for the current number only
        // if there are more than 4.
        if (prime_factors.size() > 4)
        {
            // Sort and then print the prime factors.
            sort(begin(prime_factors), end(prime_factors));

            wstringstream ss;
            ss << L"Prime factors of " << carmichael << L" are:";

            for_each (begin(prime_factors), end(prime_factors),
                [&](int prime_factor) { ss << L' ' << prime_factor; });
            ss << L'.' << endl;

            wcout << ss.str();
        }
    });
}

// Wait for the task group to finish.
tasks.wait();
}

```

This example produces the following sample output.

```

Prime factors of 9890881 are: 7 11 13 41 241.
Prime factors of 825265 are: 5 7 17 19 73.
Prime factors of 1050985 are: 5 13 19 23 37.

```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `carmichael-primes.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc carmichael-primes.cpp
```

## See also

[Parallel Containers and Objects](#)

[Task Parallelism](#)

[concurrent\\_vector Class](#)

[concurrent\\_queue Class](#)

[parallel\\_invoke](#) Function

[parallel\\_for](#) Function

[task\\_group](#) Class

# How to: Use combinable to Improve Performance

3/4/2019 • 3 minutes to read • [Edit Online](#)

This example shows how to use the `concurrency::combinable` class to compute the sum of the numbers in a `std::array` object that are prime. The `combinable` class improves performance by eliminating shared state.

## TIP

In some cases, parallel map (`concurrency::parallel_transform`) and reduce (`concurrency::parallel_reduce`) can provide performance improvements over `combinable`. For an example that uses map and reduce operations to produce the same results as this example, see [Parallel Algorithms](#).

## Example

The following example uses the `std::accumulate` function to compute the sum of the elements in an array that are prime. In this example, `a` is an `array` object and the `is_prime` function determines whether its input value is prime.

```
prime_sum = accumulate(begin(a), end(a), 0, [&](int acc, int i) {
    return acc + (is_prime(i) ? i : 0);
});
```

## Example

The following example shows a naïve way to parallelize the previous example. This example uses the `concurrency::parallel_for_each` algorithm to process the array in parallel and a `concurrency::critical_section` object to synchronize access to the `prime_sum` variable. This example does not scale because each thread must wait for the shared resource to become available.

```
critical_section cs;
prime_sum = 0;
parallel_for_each(begin(a), end(a), [&](int i) {
    cs.lock();
    prime_sum += (is_prime(i) ? i : 0);
    cs.unlock();
});
```

## Example

The following example uses a `combinable` object to improve the performance of the previous example. This example eliminates the need for synchronization objects; it scales because the `combinable` object enables each thread to perform its task independently.

A `combinable` object is typically used in two steps. First, produce a series of fine-grained computations by performing work in parallel. Next, combine (or reduce) the computations into a final result. This example uses the `concurrency::combinable::local` method to obtain a reference to the local sum. It then uses the `concurrency::combinable::combine` method and a `std::plus` object to combine the local computations into the final result.

```
combinable<int> sum;
parallel_for_each(begin(a), end(a), [&](int i) {
    sum.local() += (is_prime(i) ? i : 0);
});
prime_sum = sum.combine(plus<int>());
```

## Example

The following complete example computes the sum of prime numbers both serially and in parallel. The example prints to the console the time that is required to perform both computations.

```

// parallel-sum-of-primes.cpp
// compile with: /EHsc
#include <windows.h>
#include <ppl.h>
#include <array>
#include <numeric>
#include <iostream>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}

// Determines whether the input value is prime.
bool is_prime(int n)
{
    if (n < 2)
        return false;
    for (int i = 2; i < n; ++i)
    {
        if ((n % i) == 0)
            return false;
    }
    return true;
}

int wmain()
{
    // Create an array object that contains 200000 integers.
    array<int, 200000> a;

    // Initialize the array such that a[i] == i.
    iota(begin(a), end(a), 0);

    int prime_sum;
    __int64 elapsed;

    // Compute the sum of the numbers in the array that are prime.
    elapsed = time_call([&] {
        prime_sum = accumulate(begin(a), end(a), 0, [&](int acc, int i) {
            return acc + (is_prime(i) ? i : 0);
        });
    });
    wcout << prime_sum << endl;
    wcout << L"serial time: " << elapsed << L" ms" << endl << endl;

    // Now perform the same task in parallel.
    elapsed = time_call([&] {
        combinable<int> sum;
        parallel_for_each(begin(a), end(a), [&](int i) {
            sum.local() += (is_prime(i) ? i : 0);
        });
        prime_sum = sum.combine(plus<int>());
    });
    wcout << prime_sum << endl;
    wcout << L"parallel time: " << elapsed << L" ms" << endl << endl;
}

```

The following sample output is for a computer that has four processors.

```
1709600813
serial time: 6178 ms

1709600813
parallel time: 1638 ms
```

## Compiling the Code

To compile the code, copy it and then paste it in a Visual Studio project, or paste it in a file that is named `parallel-sum-of-primes.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc parallel-sum-of-primes.cpp
```

## Robust Programming

For an example that uses map and reduce operations to produce the same results, see [Parallel Algorithms](#).

## See also

[Parallel Containers and Objects](#)

[combinable Class](#)

[critical\\_section Class](#)

# How to: Use combinable to Combine Sets

3/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to use the `concurrency::combinable` class to compute the set of prime numbers.

## Example

The following example computes the set of prime numbers two times. Each computation stores the result in a `std::bitset` object. The example first computes the set serially and then computes the set in parallel. The example also prints to the console the time that is required to perform both computations.

This example uses the `concurrency::parallel_for` algorithm and a `combinable` object to generate thread-local sets. It then uses the `concurrency::combinable::combine_each` method to combine the thread-local sets into the final set.

```
// parallel-combine-primes.cpp
// compile with: /EHsc
#include <windows.h>
#include <ppl.h>
#include <bitset>
#include <iostream>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}

// Determines whether the input value is prime.
bool is_prime(int n)
{
    if (n < 2)
        return false;
    for (int i = 2; i < n; ++i)
    {
        if ((n % i) == 0)
            return false;
    }
    return true;
}

const int limit = 40000;

int wmain()
{
    // A set of prime numbers that is computed serially.
    bitset<limit> primes1;

    // A set of prime numbers that is computed in parallel.
    bitset<limit> primes2;

    __int64 elapsed;

    // Compute the set of prime numbers in a serial loop.
```

```

elapsed = time_call([&
{
    for(int i = 0; i < limit; ++i) {
        if (is_prime(i))
            primes1.set(i);
    }
});
wcout << L"serial time: " << elapsed << L" ms" << endl << endl;

// Compute the same set of numbers in parallel.
elapsed = time_call([&
{
    // Use a parallel_for loop and a combinable object to compute
    // the set in parallel.
    // You do not need to synchronize access to the set because the
    // combinable object provides a separate bitset object to each thread.
    combinable<bitset<limit>> working;
    parallel_for(0, limit, [&](int i) {
        if (is_prime(i))
            working.local().set(i);
    });

    // Merge each thread-local computation into the final result.
    working.combine_each([&](bitset<limit>& local) {
        primes2 |= local;
    });
});
wcout << L"parallel time: " << elapsed << L" ms" << endl << endl;
}

```

The following sample output is for a computer that has four processors.

```

serial time: 312 ms

parallel time: 78 ms

```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named

```
parallel-combine-primes.cpp
```

and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc parallel-combine-primes.cpp**

## See also

[Parallel Containers and Objects](#)

[combinable Class](#)

[combinable::combine\\_each Method](#)

# Cancellation in the PPL

3/4/2019 • 22 minutes to read • [Edit Online](#)

This document explains the role of cancellation in the Parallel Patterns Library (PPL), how to cancel parallel work, and how to determine when parallel work is canceled.

## NOTE

The runtime uses exception handling to implement cancellation. Do not catch or handle these exceptions in your code. In addition, we recommend that you write exception-safe code in the function bodies for your tasks. For instance, you can use the *Resource Acquisition Is Initialization* (RAII) pattern to ensure that resources are correctly handled when an exception is thrown in the body of a task. For a complete example that uses the RAII pattern to clean up a resource in a cancelable task, see [Walkthrough: Removing Work from a User-Interface Thread](#).

## Key Points

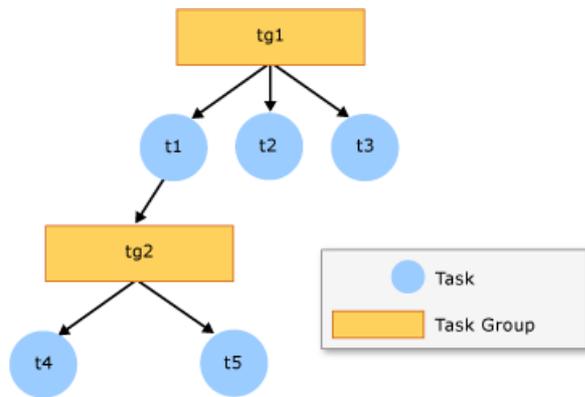
- Cancellation is cooperative and involves coordination between the code that requests cancellation and the task that responds to cancellation.
- When possible, use cancellation tokens to cancel work. The `concurrency::cancellation_token` class defines a cancellation token.
- When you use cancellation tokens, use the `concurrency::cancellation_token_source::cancel` method to initiate cancellation and the `concurrency::cancel_current_task` function to respond to cancellation. Use the `concurrency::cancellation_token::is_canceled` method to check whether any other task has requested cancellation.
- Cancellation does not occur immediately. Although new work is not started if a task or task group is cancelled, active work must check for and respond to cancellation.
- A value-based continuation inherits the cancellation token of its antecedent task. A task-based continuation never inherits the token of its antecedent task.
- Use the `concurrency::cancellation_token::none` method when you call a constructor or function that takes a `cancellation_token` object but you do not want the operation to be cancellable. Also, if you do not pass a cancellation token to the `concurrency::task` constructor or the `concurrency::create_task` function, that task is not cancellable.

## In this Document

- [Parallel Work Trees](#)
- [Canceling Parallel Tasks](#)
  - [Using a Cancellation Token to Cancel Parallel Work](#)
  - [Using the cancel Method to Cancel Parallel Work](#)
  - [Using Exceptions to Cancel Parallel Work](#)
- [Canceling Parallel Algorithms](#)
- [When Not to Use Cancellation](#)

# Parallel Work Trees

The PPL uses tasks and task groups to manage fine-grained tasks and computations. You can nest task groups to form *trees* of parallel work. The following illustration shows a parallel work tree. In this illustration, `tg1` and `tg2` represent task groups; `t1`, `t2`, `t3`, `t4`, and `t5` represent the work that the task groups perform.



The following example shows the code that is required to create the tree in the illustration. In this example, `tg1` and `tg2` are `concurrency::structured_task_group` objects; `t1`, `t2`, `t3`, `t4`, and `t5` are `concurrency::task_handle` objects.

```

// task-tree.cpp
// compile with: /c /EHsc
#include <ppl.h>
#include <sstream>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

void create_task_tree()
{
    // Create a task group that serves as the root of the tree.
    structured_task_group tg1;

    // Create a task that contains a nested task group.
    auto t1 = make_task([&] {
        structured_task_group tg2;

        // Create a child task.
        auto t4 = make_task([&] {
            // TODO: Perform work here.
        });

        // Create a child task.
        auto t5 = make_task([&] {
            // TODO: Perform work here.
        });

        // Run the child tasks and wait for them to finish.
        tg2.run(t4);
        tg2.run(t5);
        tg2.wait();
    });

    // Create a child task.
    auto t2 = make_task([&] {
        // TODO: Perform work here.
    });

    // Create a child task.
    auto t3 = make_task([&] {
        // TODO: Perform work here.
    });

    // Run the child tasks and wait for them to finish.
    tg1.run(t1);
    tg1.run(t2);
    tg1.run(t3);
    tg1.wait();
}

```

You can also use the [concurrency::task\\_group](#) class to create a similar work tree. The [concurrency::task](#) class also supports the notion of a tree of work. However, a `task` tree is a dependency tree. In a `task` tree, future works completes after current work. In a task group tree, internal work completes before outer work. For more information about the differences between tasks and task groups, see [Task Parallelism](#).

[\[Top\]](#)

## Canceling Parallel Tasks

There are multiple ways to cancel parallel work. The preferred way is to use a cancellation token. Task groups also support the [concurrency::task\\_group::cancel](#) method and the [concurrency::structured\\_task\\_group::cancel](#) method. The final way is to throw an exception in the body of a

task work function. No matter which method you choose, understand that cancellation does not occur immediately. Although new work is not started if a task or task group is cancelled, active work must check for and respond to cancellation.

For more examples that cancel parallel tasks, see [Walkthrough: Connecting Using Tasks and XML HTTP Requests](#), [How to: Use Cancellation to Break from a Parallel Loop](#), and [How to: Use Exception Handling to Break from a Parallel Loop](#).

### Using a Cancellation Token to Cancel Parallel Work

The `task`, `task_group`, and `structured_task_group` classes support cancellation through the use of cancellation tokens. The PPL defines the `concurrency::cancellation_token_source` and `concurrency::cancellation_token` classes for this purpose. When you use a cancellation token to cancel work, the runtime does not start new work that subscribes to that token. Work that is already active can use the `is_canceled` member function to monitor the cancellation token and stop when it can.

To initiate cancellation, call the `concurrency::cancellation_token_source::cancel` method. You respond to cancellation in these ways:

- For `task` objects, use the `concurrency::cancel_current_task` function. `cancel_current_task` cancels the current task and any of its value-based continuations. (It does not cancel the cancellation *token* that is associated with the task or its continuations.)
- For task groups and parallel algorithms, use the `concurrency::is_current_task_group_canceled` function to detect cancellation and return as soon as possible from the task body when this function returns **true**. (Do not call `cancel_current_task` from a task group.)

The following example shows the first basic pattern for task cancellation. The task body occasionally checks for cancellation inside a loop.

```
// task-basic-cancellation.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <concrct.h>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

bool do_work()
{
    // Simulate work.
    wcout << L"Performing work..." << endl;
    wait(250);
    return true;
}

int wmain()
{
    cancellation_token_source cts;
    auto token = cts.get_token();

    wcout << L"Creating task..." << endl;

    // Create a task that performs work until it is canceled.
    auto t = create_task([&]
    {
        bool moreToDo = true;
        while (moreToDo)
        {
            // Check for cancellation.
            if (token.is_canceled())
```

```

    {
        // TODO: Perform any necessary cleanup here...

        // Cancel the current task.
        cancel_current_task();
    }
    else
    {
        // Perform work.
        moreToDo = do_work();
    }
}
}, token);

// Wait for one second and then cancel the task.
wait(1000);

wcout << L"Canceling task..." << endl;
cts.cancel();

// Wait for the task to cancel.
wcout << L"Waiting for task to complete..." << endl;
t.wait();

wcout << L"Done." << endl;
}

/* Sample output:
Creating task...
Performing work...
Performing work...
Performing work...
Performing work...
Canceling task...
Waiting for task to complete...
Done.
*/

```

The `cancel_current_task` function throws; therefore, you do not need to explicitly return from the current loop or function.

#### TIP

Alternatively, you can call the `concurrency::interruption_point` function instead of `cancel_current_task`.

It is important to call `cancel_current_task` when you respond to cancellation because it transitions the task to the canceled state. If you return early instead of calling `cancel_current_task`, the operation transitions to the completed state and any value-based continuations are run.

#### Caution

Never throw `task_canceled` from your code. Call `cancel_current_task` instead.

When a task ends in the canceled state, the `concurrency::task::get` method throws `concurrency::task_canceled`. (Conversely, `concurrency::task::wait` returns `task_status::canceled` and does not throw.) The following example illustrates this behavior for a task-based continuation. A task-based continuation is always called, even when the antecedent task is canceled.

```

// task-canceled.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    auto t1 = create_task([]() -> int
    {
        // Cancel the task.
        cancel_current_task();
    });

    // Create a continuation that retrieves the value from the previous.
    auto t2 = t1.then([](task<int> t)
    {
        try
        {
            int n = t.get();
            wcout << L"The previous task returned " << n << L'.' << endl;
        }
        catch (const task_canceled& e)
        {
            wcout << L"The previous task was canceled." << endl;
        }
    });

    // Wait for all tasks to complete.
    t2.wait();
}
/* Output:
   The previous task was canceled.
*/

```

Because value-based continuations inherit the token of their antecedent task unless they were created with an explicit token, the continuations immediately enter the canceled state even when the antecedent task is still executing. Therefore, any exception that is thrown by the antecedent task after cancellation is not propagated to the continuation tasks. Cancellation always overrides the state of the antecedent task. The following example resembles the previous, but illustrates the behavior for a value-based continuation.

```

auto t1 = create_task([]() -> int
{
    // Cancel the task.
    cancel_current_task();
});

// Create a continuation that retrieves the value from the previous.
auto t2 = t1.then([](int n)
{
    wcout << L"The previous task returned " << n << L'.' << endl;
});

try
{
    // Wait for all tasks to complete.
    t2.get();
}
catch (const task_canceled& e)
{
    wcout << L"The task was canceled." << endl;
}
/* Output:
    The task was canceled.
*/

```

#### Caution

If you do not pass a cancellation token to the `task` constructor or the `concurrency::create_task` function, that task is not cancellable. In addition, you must pass the same cancellation token to the constructor of any nested tasks (that is, tasks that are created in the body of another task) to cancel all tasks simultaneously.

You might want to run arbitrary code when a cancellation token is canceled. For example, if your user chooses a **Cancel** button on the user interface to cancel the operation, you could disable that button until the user starts another operation. The following example shows how to use the `concurrency::cancellation_token::register_callback` method to register a callback function that runs when a cancellation token is canceled.

```

// task-cancellation-callback.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    cancellation_token_source cts;
    auto token = cts.get_token();

    // An event that is set in the cancellation callback.
    event e;

    cancellation_token_registration cookie;
    cookie = token.register_callback([&e, token, &cookie]()
    {
        wcout << L"In cancellation callback..." << endl;
        e.set();

        // Although not required, demonstrate how to unregister
        // the callback.
        token.deregister_callback(cookie);
    });

    wcout << L"Creating task..." << endl;

    // Create a task that waits to be canceled.
    auto t = create_task([&e]
    {
        e.wait();
    }, token);

    // Cancel the task.
    wcout << L"Canceling task..." << endl;
    cts.cancel();

    // Wait for the task to cancel.
    t.wait();

    wcout << L"Done." << endl;
}
/* Sample output:
   Creating task...
   Canceling task...
   In cancellation callback...
   Done.
*/

```

The document [Task Parallelism](#) explains the difference between value-based and task-based continuations. If you do not provide a `cancellation_token` object to a continuation task, the continuation inherits the cancellation token from the antecedent task in the following ways:

- A value-based continuation always inherits the cancellation token of the antecedent task.
- A task-based continuation never inherits the cancellation token of the antecedent task. The only way to make a task-based continuation cancelable is to explicitly pass a cancellation token.

These behaviors are not affected by a faulted task (that is, one that throws an exception). In this case, a value-based continuation is cancelled; a task-based continuation is not cancelled.

**Caution**

A task that is created in another task (in other words, a nested task) does not inherit the cancellation token of

the parent task. Only a value-based continuation inherits the cancellation token of its antecedent task.

#### TIP

Use the `concurrency::cancellation_token::none` method when you call a constructor or function that takes a `cancellation_token` object and you do not want the operation to be cancellable.

You can also provide a cancellation token to the constructor of a `task_group` or `structured_task_group` object. An important aspect of this is that child task groups inherit this cancellation token. For an example that demonstrates this concept by using the `concurrency::run_with_cancellation_token` function to run to call `parallel_for`, see [Canceling Parallel Algorithms](#) later in this document.

[\[Top\]](#)

#### Cancellation Tokens and Task Composition

The `concurrency::when_all` and `concurrency::when_any` functions can help you compose multiple tasks to implement common patterns. This section describes how these functions work with cancellation tokens.

When you provide a cancellation token to either the `when_all` and `when_any` function, that function cancels only when that cancellation token is cancelled or when one of the participant tasks ends in a canceled state or throws an exception.

The `when_all` function inherits the cancellation token from each task that composes the overall operation when you do not provide a cancellation token to it. The task that is returned from `when_all` is canceled when any of these tokens are cancelled and at least one of the participant tasks has not yet started or is running. A similar behavior occurs when one of the tasks throws an exception - the task that is returned from `when_all` is immediately canceled with that exception.

The runtime chooses the cancellation token for the task that is returned from `when_any` function when that task completes. If none of the participant tasks finish in a completed state and one or more of the tasks throws an exception, one of the tasks that threw is chosen to complete the `when_any` and its token is chosen as the token for the final task. If more than one task finishes in the completed state, the task that is returned from `when_any` task ends in a completed state. The runtime tries to pick a completed task whose token is not canceled at the time of completion so that the task that is returned from `when_any` is not immediately canceled even though other executing tasks might complete at a later point.

[\[Top\]](#)

#### Using the cancel Method to Cancel Parallel Work

The `concurrency::task_group::cancel` and `concurrency::structured_task_group::cancel` methods set a task group to the canceled state. After you call `cancel`, the task group does not start future tasks. The `cancel` methods can be called by multiple child tasks. A canceled task causes the `concurrency::task_group::wait` and `concurrency::structured_task_group::wait` methods to return `concurrency::canceled`.

If a task group is canceled, calls from each child task into the runtime can trigger an *interruption point*, which causes the runtime to throw and catch an internal exception type to cancel active tasks. The Concurrency Runtime does not define specific interruption points; they can occur in any call to the runtime. The runtime must handle the exceptions that it throws in order to perform cancellation. Therefore, do not handle unknown exceptions in the body of a task.

If a child task performs a time-consuming operation and does not call into the runtime, it must periodically check for cancellation and exit in a timely manner. The following example shows one way to determine when work is canceled. Task `t4` cancels the parent task group when it encounters an error. Task `t5` occasionally calls the `structured_task_group::is_canceling` method to check for cancellation. If the parent task group is canceled, task `t5` prints a message and exits.

```

structured_task_group tg2;

// Create a child task.
auto t4 = make_task([&] {
    // Perform work in a loop.
    for (int i = 0; i < 1000; ++i)
    {
        // Call a function to perform work.
        // If the work function fails, cancel the parent task
        // and break from the loop.
        bool succeeded = work(i);
        if (!succeeded)
        {
            tg2.cancel();
            break;
        }
    }
});

// Create a child task.
auto t5 = make_task([&] {
    // Perform work in a loop.
    for (int i = 0; i < 1000; ++i)
    {
        // To reduce overhead, occasionally check for
        // cancelation.
        if ((i%100) == 0)
        {
            if (tg2.is_canceling())
            {
                wcout << L"The task was canceled." << endl;
                break;
            }
        }

        // TODO: Perform work here.
    }
});

// Run the child tasks and wait for them to finish.
tg2.run(t4);
tg2.run(t5);
tg2.wait();

```

This example checks for cancellation on every 100<sup>th</sup> iteration of the task loop. The frequency with which you check for cancellation depends on the amount of work your task performs and how quickly you need for tasks to respond to cancellation.

If you do not have access to the parent task group object, call the [concurrency::is\\_current\\_task\\_group\\_canceling](#) function to determine whether the parent task group is canceled.

The `cancel` method only affects child tasks. For example, if you cancel the task group `tg1` in the illustration of the parallel work tree, all tasks in the tree (`t1`, `t2`, `t3`, `t4`, and `t5`) are affected. If you cancel the nested task group, `tg2`, only tasks `t4` and `t5` are affected.

When you call the `cancel` method, all child task groups are also canceled. However, cancellation does not affect any parents of the task group in a parallel work tree. The following examples show this by building on the parallel work tree illustration.

The first of these examples creates a work function for the task `t4`, which is a child of the task group `tg2`. The work function calls the function `work` in a loop. If any call to `work` fails, the task cancels its parent task group. This causes task group `tg2` to enter the canceled state, but it does not cancel task group `tg1`.

```

auto t4 = make_task([&] {
    // Perform work in a loop.
    for (int i = 0; i < 1000; ++i)
    {
        // Call a function to perform work.
        // If the work function fails, cancel the parent task
        // and break from the loop.
        bool succeeded = work(i);
        if (!succeeded)
        {
            tg2.cancel();
            break;
        }
    }
});

```

This second example resembles the first one, except that the task cancels task group `tg1`. This affects all tasks in the tree (`t1`, `t2`, `t3`, `t4`, and `t5`).

```

auto t4 = make_task([&] {
    // Perform work in a loop.
    for (int i = 0; i < 1000; ++i)
    {
        // Call a function to perform work.
        // If the work function fails, cancel all tasks in the tree.
        bool succeeded = work(i);
        if (!succeeded)
        {
            tg1.cancel();
            break;
        }
    }
});

```

The `structured_task_group` class is not thread-safe. Therefore, a child task that calls a method of its parent `structured_task_group` object produces unspecified behavior. The exceptions to this rule are the `structured_task_group::cancel` and `concurrency::structured_task_group::is_canceling` methods. A child task can call these methods to cancel the parent task group and check for cancellation.

#### Caution

Although you can use a cancellation token to cancel work that is performed by a task group that runs as a child of a `task` object, you cannot use the `task_group::cancel` or `structured_task_group::cancel` methods to cancel `task` objects that run in a task group.

[\[Top\]](#)

## Using Exceptions to Cancel Parallel Work

The use of cancellation tokens and the `cancel` method are more efficient than exception handling at canceling a parallel work tree. Cancellation tokens and the `cancel` method cancel a task and any child tasks in a top-down manner. Conversely, exception handling works in a bottom-up manner and must cancel each child task group independently as the exception propagates upward. The topic [Exception Handling](#) explains how the Concurrency Runtime uses exceptions to communicate errors. However, not all exceptions indicate an error. For example, a search algorithm might cancel its associated task when it finds the result. However, as mentioned previously, exception handling is less efficient than using the `cancel` method to cancel parallel work.

#### Caution

We recommend that you use exceptions to cancel parallel work only when necessary. Cancellation tokens and the task group `cancel` methods are more efficient and less prone to error.

When you throw an exception in the body of a work function that you pass to a task group, the runtime stores that exception and marshals the exception to the context that waits for the task group to finish. As with the `cancel` method, the runtime discards any tasks that have not yet started, and does not accept new tasks.

This third example resembles the second one, except that task `t4` throws an exception to cancel the task group `tg2`. This example uses a `try - catch` block to check for cancellation when the task group `tg2` waits for its child tasks to finish. Like the first example, this causes the task group `tg2` to enter the canceled state, but it does not cancel task group `tg1`.

```
structured_task_group tg2;

// Create a child task.
auto t4 = make_task([&] {
    // Perform work in a loop.
    for (int i = 0; i < 1000; ++i)
    {
        // Call a function to perform work.
        // If the work function fails, throw an exception to
        // cancel the parent task.
        bool succeeded = work(i);
        if (!succeeded)
        {
            throw exception("The task failed");
        }
    }
});

// Create a child task.
auto t5 = make_task([&] {
    // TODO: Perform work here.
});

// Run the child tasks.
tg2.run(t4);
tg2.run(t5);

// Wait for the tasks to finish. The runtime marshals any exception
// that occurs to the call to wait.
try
{
    tg2.wait();
}
catch (const exception& e)
{
    wcout << e.what() << endl;
}
```

This fourth example uses exception handling to cancel the whole work tree. The example catches the exception when task group `tg1` waits for its child tasks to finish instead of when task group `tg2` waits for its child tasks. Like the second example, this causes both tasks groups in the tree, `tg1` and `tg2`, to enter the canceled state.

```
// Run the child tasks.
tg1.run(t1);
tg1.run(t2);
tg1.run(t3);

// Wait for the tasks to finish. The runtime marshals any exception
// that occurs to the call to wait.
try
{
    tg1.wait();
}
catch (const exception& e)
{
    wcout << e.what() << endl;
}
```

Because the `task_group::wait` and `structured_task_group::wait` methods throw when a child task throws an exception, you do not receive a return value from them.

[\[Top\]](#)

## Canceling Parallel Algorithms

Parallel algorithms in the PPL, for example, `parallel_for`, build on task groups. Therefore, you can use many of the same techniques to cancel a parallel algorithm.

The following examples illustrate several ways to cancel a parallel algorithm.

The following example uses the `run_with_cancellation_token` function to call the `parallel_for` algorithm. The `run_with_cancellation_token` function takes a cancellation token as an argument and calls the provided work function synchronously. Because parallel algorithms are built upon tasks, they inherit the cancellation token of the parent task. Therefore, `parallel_for` can respond to cancellation.

```

// cancel-parallel-for.cpp
// compile with: /EHsc
#include <ppltasks.h>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Call parallel_for in the context of a cancellation token.
    cancellation_token_source cts;
    run_with_cancellation_token([&cts]()
    {
        // Print values to the console in parallel.
        parallel_for(0, 20, [&cts](int n)
        {
            // For demonstration, cancel the overall operation
            // when n equals 11.
            if (n == 11)
            {
                cts.cancel();
            }
            // Otherwise, print the value.
            else
            {
                wstringstream ss;
                ss << n << endl;
                wcout << ss.str();
            }
        });
    }, cts.get_token());
}
/* Sample output:
15
16
17
10
0
18
5
*/

```

The following example uses the `concurrency::structured_task_group::run_and_wait` method to call the `parallel_for` algorithm. The `structured_task_group::run_and_wait` method waits for the provided task to finish. The `structured_task_group` object enables the work function to cancel the task.

```

// To enable cancelation, call parallel_for in a task group.
structured_task_group tg;

task_group_status status = tg.run_and_wait([&] {
    parallel_for(0, 100, [&](int i) {
        // Cancel the task when i is 50.
        if (i == 50)
        {
            tg.cancel();
        }
        else
        {
            // TODO: Perform work here.
        }
    });
});

// Print the task group status.
wcout << L"The task group status is: ";
switch (status)
{
case not_complete:
    wcout << L"not complete." << endl;
    break;
case completed:
    wcout << L"completed." << endl;
    break;
case canceled:
    wcout << L"canceled." << endl;
    break;
default:
    wcout << L"unknown." << endl;
    break;
}

```

This example produces the following output.

```
The task group status is: canceled.
```

The following example uses exception handling to cancel a `parallel_for` loop. The runtime marshals the exception to the calling context.

```

try
{
    parallel_for(0, 100, [&](int i) {
        // Throw an exception to cancel the task when i is 50.
        if (i == 50)
        {
            throw i;
        }
        else
        {
            // TODO: Perform work here.
        }
    });
}
catch (int n)
{
    wcout << L"Caught " << n << endl;
}

```

This example produces the following output.

The following example uses a Boolean flag to coordinate cancellation in a `parallel_for` loop. Every task runs because this example does not use the `cancel` method or exception handling to cancel the overall set of tasks. Therefore, this technique can have more computational overhead than a cancellation mechanism.

```
// Create a Boolean flag to coordinate cancellation.
bool canceled = false;

parallel_for(0, 100, [&](int i) {
    // For illustration, set the flag to cancel the task when i is 50.
    if (i == 50)
    {
        canceled = true;
    }

    // Perform work if the task is not canceled.
    if (!canceled)
    {
        // TODO: Perform work here.
    }
});
```

Each cancellation method has advantages over the others. Choose the method that fits your specific needs.

[\[Top\]](#)

## When Not to Use Cancellation

The use of cancellation is appropriate when each member of a group of related tasks can exit in a timely manner. However, there are some scenarios where cancellation may not be appropriate for your application. For example, because task cancellation is cooperative, the overall set of tasks will not cancel if any individual task is blocked. For example, if one task has not yet started, but it unblocks another active task, it will not start if the task group is canceled. This can cause deadlock to occur in your application. A second example of where the use of cancellation may not be appropriate is when a task is canceled, but its child task performs an important operation, such as freeing a resource. Because the overall set of tasks is canceled when the parent task is canceled, that operation will not execute. For an example that illustrates this point, see the [Understand how Cancellation and Exception Handling Affect Object Destruction](#) section in the Best Practices in the Parallel Patterns Library topic.

[\[Top\]](#)

## Related Topics

TITLE	DESCRIPTION
<a href="#">How to: Use Cancellation to Break from a Parallel Loop</a>	Shows how to use cancellation to implement a parallel search algorithm.
<a href="#">How to: Use Exception Handling to Break from a Parallel Loop</a>	Shows how to use the <code>task_group</code> class to write a search algorithm for a basic tree structure.
<a href="#">Exception Handling</a>	Describes how the runtime handles exceptions that are thrown by task groups, lightweight tasks, and asynchronous agents, and how to respond to exceptions in your applications.

TITLE	DESCRIPTION
<a href="#">Task Parallelism</a>	Describes how tasks relate to task groups and how you can use unstructured and structured tasks in your applications.
<a href="#">Parallel Algorithms</a>	Describes the parallel algorithms, which concurrently perform work on collections of data
<a href="#">Parallel Patterns Library (PPL)</a>	Provides an overview of the Parallel Patterns Library.

## Reference

[task Class \(Concurrency Runtime\)](#)

[cancellation\\_token\\_source Class](#)

[cancellation\\_token Class](#)

[task\\_group Class](#)

[structured\\_task\\_group Class](#)

[parallel\\_for Function](#)

# How to: Use Cancellation to Break from a Parallel Loop

3/4/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use cancellation to implement a basic parallel search algorithm.

## Example

The following example uses cancellation to search for an element in an array. The `parallel_find_any` function uses the `concurrency::parallel_for` algorithm and the `concurrency::run_with_cancellation_token` function to search for the position that contains the given value. When the parallel loop finds the value, it calls the `concurrency::cancellation_token_source::cancel` method to cancel future work.

```

// parallel-array-search.cpp
// compile with: /EHsc
#include <ppl.h>
#include <iostream>
#include <random>

using namespace concurrency;
using namespace std;

// Returns the position in the provided array that contains the given value,
// or -1 if the value is not in the array.
template<typename T>
int parallel_find_any(const T a[], size_t count, const T& what)
{
    // The position of the element in the array.
    // The default value, -1, indicates that the element is not in the array.
    int position = -1;

    // Call parallel_for in the context of a cancellation token to search for the element.
    cancellation_token_source cts;
    run_with_cancellation_token([count, what, &a, &position, &cts]()
    {
        parallel_for(std::size_t(0), count, [what, &a, &position, &cts](int n) {
            if (a[n] == what)
            {
                // Set the return value and cancel the remaining tasks.
                position = n;
                cts.cancel();
            }
        });
    }, cts.get_token());

    return position;
}

int wmain()
{
    const size_t count = 10000;
    int values[count];

    // Fill the array with random values.
    mt19937 gen(34);
    for (size_t i = 0; i < count; ++i)
    {
        values[i] = gen()%10000;
    }

    // Search for any position in the array that contains value 3123.
    const int what = 3123;
    int position = parallel_find_any(values, count, what);
    if (position >= 0)
    {
        wcout << what << L" is at position " << position << L'.' << endl;
    }
    else
    {
        wcout << what << L" is not in the array." << endl;
    }
}
/* Sample output:
   3123 is at position 7835.
*/

```

The `concurrency::parallel_for` algorithm acts concurrently. Therefore, it does not perform the operations in a pre-determined order. If the array contains multiple instances of the value, the result can be any one of its positions.

# Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named

`parallel-array-search.cpp`

and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc parallel-array-search.cpp**

## See also

[Cancellation in the PPL](#)

[Parallel Algorithms](#)

[parallel\\_for Function](#)

[cancellation\\_token\\_source Class](#)

# How to: Use Exception Handling to Break from a Parallel Loop

3/4/2019 • 6 minutes to read • [Edit Online](#)

This topic shows how to write a search algorithm for a basic tree structure.

The topic [Cancellation](#) explains the role of cancellation in the Parallel Patterns Library. The use of exception handling is a less efficient way to cancel parallel work than the use of the `concurrency::task_group::cancel` and `concurrency::structured_task_group::cancel` methods. However, one scenario where the use of exception handling to cancel work is appropriate is when you call into a third-party library that uses tasks or parallel algorithms but does not provide a `task_group` or `structured_task_group` object to cancel.

## Example

The following example shows a basic `tree` type that contains a data element and a list of child nodes. The following section shows the body of the `for_all` method, which recursively performs a work function on each child node.

```
// A simple tree structure that has multiple child nodes.
template <typename T>
class tree
{
public:
    explicit tree(T data)
        : _data(data)
    {
    }

    // Retrieves the data element for the node.
    T get_data() const
    {
        return _data;
    }

    // Adds a child node to the tree.
    void add_child(tree& child)
    {
        _children.push_back(child);
    }

    // Performs the given work function on the data element of the tree and
    // on each child.
    template<class Function>
    void for_all(Function& action);

private:
    // The data for this node.
    T _data;
    // The child nodes.
    list<tree> _children;
};
```

## Example

The following example shows the `for_all` method. It uses the `concurrency::parallel_for_each` algorithm to

perform a work function on each node of the tree in parallel.

```
// Performs the given work function on the data element of the tree and
// on each child.
template<class Function>
void for_all(Function& action)
{
    // Perform the action on each child.
    parallel_for_each(begin(_children), end(_children), [&](tree& child) {
        child.for_all(action);
    });

    // Perform the action on this node.
    action(*this);
}
```

## Example

The following example shows the `search_for_value` function, which searches for a value in the provided `tree` object. This function passes to the `for_all` method a work function that throws when it finds a tree node that contains the provided value.

Assume that the `tree` class is provided by a third-party library, and that you cannot modify it. In this case, the use of exception handling is appropriate because the `for_all` method does not provide a `task_group` or `structured_task_group` object to the caller. Therefore, the work function is unable to directly cancel its parent task group.

When the work function that you provide to a task group throws an exception, the runtime stops all tasks that are in the task group (including any child task groups) and discards any tasks that have not yet started. The `search_for_value` function uses a `try - catch` block to capture the exception and print the result to the console.

```
// Searches for a value in the provided tree object.
template <typename T>
void search_for_value(tree<T>& t, int value)
{
    try
    {
        // Call the for_all method to search for a value. The work function
        // throws an exception when it finds the value.
        t.for_all([value](const tree<T>& node) {
            if (node.get_data() == value)
            {
                throw &node;
            }
        });
    }
    catch (const tree<T>* node)
    {
        // A matching node was found. Print a message to the console.
        wstringstream ss;
        ss << L"Found a node with value " << value << L'.' << endl;
        wcout << ss.str();
        return;
    }

    // A matching node was not found. Print a message to the console.
    wstringstream ss;
    ss << L"Did not find node with value " << value << L'.' << endl;
    wcout << ss.str();
}
```

## Example

The following example creates a `tree` object and searches it for several values in parallel. The `build_tree` function is shown later in this topic.

```
int wmain()
{
    // Build a tree that is four levels deep with the initial level
    // having three children. The value of each node is a random number.
    mt19937 gen(38);
    tree<int> t = build_tree<int>(4, 3, [&gen]{ return gen()%100000; });

    // Search for a few values in the tree in parallel.
    parallel_invoke(
        [&t] { search_for_value(t, 86131); },
        [&t] { search_for_value(t, 17522); },
        [&t] { search_for_value(t, 32614); }
    );
}
```

This example uses the `concurrency::parallel_invoke` algorithm to search for values in parallel. For more information about this algorithm, see [Parallel Algorithms](#).

## Example

The following complete example uses exception handling to search for values in a basic tree structure.

```
// task-tree-search.cpp
// compile with: /EHsc
#include <ppl.h>
#include <list>
#include <iostream>
#include <algorithm>
#include <sstream>
#include <random>

using namespace concurrency;
using namespace std;

// A simple tree structure that has multiple child nodes.
template <typename T>
class tree
{
public:
    explicit tree(T data)
        : _data(data)
    {
    }

    // Retrieves the data element for the node.
    T get_data() const
    {
        return _data;
    }

    // Adds a child node to the tree.
    void add_child(tree& child)
    {
        _children.push_back(child);
    }

    // Performs the given work function on the data element of the tree and
    // on each child.
    template<class Function>
```

```

void for_all(Function& action)
{
    // Perform the action on each child.
    parallel_for_each(begin(_children), end(_children), [&](tree& child) {
        child.for_all(action);
    });

    // Perform the action on this node.
    action(*this);
}

private:
    // The data for this node.
    T _data;
    // The child nodes.
    list<tree> _children;
};

// Builds a tree with the given depth.
// Each node of the tree is initialized with the provided generator function.
// Each level of the tree has one more child than the previous level.
template <typename T, class Generator>
tree<T> build_tree(int depth, int child_count, Generator& g)
{
    // Create the tree node.
    tree<T> t(g());

    // Add children.
    if (depth > 0)
    {
        for(int i = 0; i < child_count; ++i)
        {
            t.add_child(build_tree<T>(depth - 1, child_count + 1, g));
        }
    }

    return t;
}

// Searches for a value in the provided tree object.
template <typename T>
void search_for_value(tree<T>& t, int value)
{
    try
    {
        // Call the for_all method to search for a value. The work function
        // throws an exception when it finds the value.
        t.for_all([value](const tree<T>& node) {
            if (node.get_data() == value)
            {
                throw &node;
            }
        });
    }
    catch (const tree<T>* node)
    {
        // A matching node was found. Print a message to the console.
        wstringstream ss;
        ss << L"Found a node with value " << value << L'.' << endl;
        wcout << ss.str();
        return;
    }

    // A matching node was not found. Print a message to the console.
    wstringstream ss;
    ss << L"Did not find node with value " << value << L'.' << endl;
    wcout << ss.str();
}

```

```
int wmain()
{
    // Build a tree that is four levels deep with the initial level
    // having three children. The value of each node is a random number.
    mt19937 gen(38);
    tree<int> t = build_tree<int>(4, 3, [&gen]{ return gen()%100000; });

    // Search for a few values in the tree in parallel.
    parallel_invoke(
        [&t] { search_for_value(t, 86131); },
        [&t] { search_for_value(t, 17522); },
        [&t] { search_for_value(t, 32614); }
    );
}
```

This example produces the following sample output.

```
Found a node with value 32614.
Found a node with value 86131.
Did not find node with value 17522.
```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `task-tree-search.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc task-tree-search.cpp
```

## See also

- [Cancellation in the PPL](#)
- [Exception Handling](#)
- [Task Parallelism](#)
- [Parallel Algorithms](#)
- [task\\_group Class](#)
- [structured\\_task\\_group Class](#)
- [parallel\\_for\\_each Function](#)

# Asynchronous Agents Library

3/4/2019 • 5 minutes to read • [Edit Online](#)

The Asynchronous Agents Library (or just *Agents Library*) provides a programming model that lets you increase the robustness of concurrency-enabled application development. The Agents Library is a C++ template library that promotes an actor-based programming model and in-process message passing for coarse-grained dataflow and pipelining tasks. The Agents Library builds on the scheduling and resource management components of the Concurrency Runtime.

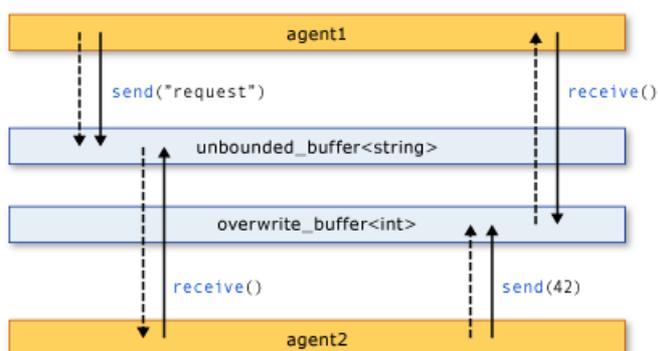
## Programming Model

The Agents Library provides alternatives to shared state by letting you connect isolated components through an asynchronous communication model that is based on dataflow instead of control flow. *Dataflow* refers to a programming model where computations are made when all required data is available; *control flow* refers to a programming model where computations are made in a predetermined order.

The dataflow programming model is related to the concept of *message passing*, where independent components of a program communicate with one another by sending messages.

The Agents Library is composed of three components: *asynchronous agents*, *asynchronous message blocks*, and *message-passing functions*. Agents maintain state, and use message blocks and message-passing functions to communicate with one another and with external components. Message-passing functions enable agents to send and receive messages to and from the external components. Asynchronous message blocks hold messages and enable agents to communicate in a synchronized manner.

The following illustration shows how two agents use message blocks and message-passing functions to communicate. In this illustration, `agent1` sends a message to `agent2` by using the `concurrency::send` function and a `concurrency::unbounded_buffer` object. `agent2` uses the `concurrency::receive` function to read the message. `agent2` uses the same method to send a message to `agent1`. Dashed arrows represent the flow of data between agents. Solid arrows connect the agents to the message blocks that they write to or read from.



### Legend

	Asynchronous agent
	Asynchronous message block
	Message-passing function

A code example that implements this illustration is shown later in this topic.

The agent programming model has several advantages over other concurrency and synchronization mechanisms, for example, events. One advantage is that by using message passing to transmit state changes between objects, you can isolate access to shared resources, and thereby improve scalability. An advantage to

message passing is that it ties synchronization to data instead of tying it to an external synchronization object. This simplifies data transmission among components and can eliminate programming errors in your applications.

## When to Use the Agents Library

Use the Agents library when you have multiple operations that must communicate with one another asynchronously. Message blocks and message-passing functions let you write parallel applications without requiring synchronization mechanisms such as locks. This lets you focus on application logic.

The agent programming model is often used to create *data pipelines* or *networks*. A data pipeline is a series of components, each of which performs a specific task that contributes to a larger goal. Every component in a dataflow pipeline performs work when it receives a message from another component. The result of that work is passed to other components in the pipeline or network. The components can use more fine-grained concurrency functionality from other libraries, for example, the [Parallel Patterns Library \(PPL\)](#).

## Example

The following example implements the illustration shown earlier in this topic.

```
// basic-agents.cpp
// compile with: /EHsc
#include <agents.h>
#include <string>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

// This agent writes a string to its target and reads an integer
// from its source.
class agent1 : public agent
{
public:
    explicit agent1(ISource<int>& source, ITarget<wstring>& target)
        : _source(source)
        , _target(target)
    {
    }

protected:
    void run()
    {
        // Send the request.
        wstringstream ss;
        ss << L"agent1: sending request..." << endl;
        wcout << ss.str();

        send(_target, wstring(L"request"));

        // Read the response.
        int response = receive(_source);

        ss = wstringstream();
        ss << L"agent1: received '" << response << L"'" << endl;
        wcout << ss.str();

        // Move the agent to the finished state.
        done();
    }

private:
```

```

    ISource<int>& _source;
    ITarget<wstring>& _target;
};

// This agent reads a string to its source and then writes an integer
// to its target.
class agent2 : public agent
{
public:
    explicit agent2(ISource<wstring>& source, ITarget<int>& target)
        : _source(source)
        , _target(target)
    {
    }

protected:
    void run()
    {
        // Read the request.
        wstring request = receive(_source);

        wstringstream ss;
        ss << L"agent2: received '" << request << L"'" << endl;
        wcout << ss.str();

        // Send the response.
        ss = wstringstream();
        ss << L"agent2: sending response..." << endl;
        wcout << ss.str();

        send(_target, 42);

        // Move the agent to the finished state.
        done();
    }

private:
    ISource<wstring>& _source;
    ITarget<int>& _target;
};

int wmain()
{
    // Step 1: Create two message buffers to serve as communication channels
    // between the agents.

    // The first agent writes messages to this buffer; the second
    // agents reads messages from this buffer.
    unbounded_buffer<wstring> buffer1;

    // The first agent reads messages from this buffer; the second
    // agents writes messages to this buffer.
    overwrite_buffer<int> buffer2;

    // Step 2: Create the agents.
    agent1 first_agent(buffer2, buffer1);
    agent2 second_agent(buffer1, buffer2);

    // Step 3: Start the agents. The runtime calls the run method on
    // each agent.
    first_agent.start();
    second_agent.start();

    // Step 4: Wait for both agents to finish.
    agent::wait(&first_agent);
    agent::wait(&second_agent);
}

```

This example produces the following output:

```
agent1: sending request...
agent2: received 'request'.
agent2: sending response...
agent1: received '42'.
```

The following topics describe the functionality used in this example.

## Related Topics

### [Asynchronous Agents](#)

Describes the role of asynchronous agents in solving larger computing tasks.

### [Asynchronous Message Blocks](#)

Describes the various message block types that are provided by the Agents Library.

### [Message Passing Functions](#)

Describes the various message passing routines that are provided by the Agents Library.

### [How to: Implement Various Producer-Consumer Patterns](#)

Describes how to implement the producer-consumer pattern in your application.

### [How to: Provide Work Functions to the call and transformer Classes](#)

Illustrates several ways to provide work functions to the `concurrency::call` and `concurrency::transformer` classes.

### [How to: Use transformer in a Data Pipeline](#)

Shows how to use the `concurrency::transformer` class in a data pipeline.

### [How to: Select Among Completed Tasks](#)

Shows how to use the `concurrency::choice` and `concurrency::join` classes to select the first task to complete a search algorithm.

### [How to: Send a Message at a Regular Interval](#)

Shows how to use the `concurrency::timer` class to send a message at a regular interval.

### [How to: Use a Message Block Filter](#)

Demonstrates how to use a filter to enable an asynchronous message block to accept or reject messages.

### [Parallel Patterns Library \(PPL\)](#)

Describes how to use various parallel patterns, such as parallel algorithms, in your applications.

### [Concurrency Runtime](#)

Describes the Concurrency Runtime, which simplifies parallel programming, and contains links to related topics.

# Asynchronous Agents

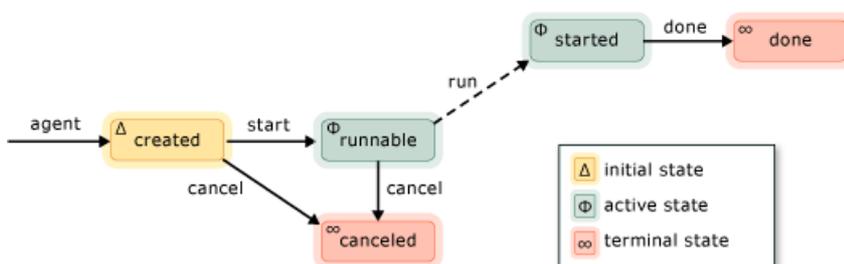
3/4/2019 • 2 minutes to read • [Edit Online](#)

An *asynchronous agent* (or just *agent*) is an application component that works asynchronously with other agents to solve larger computing tasks. Think of an agent as a task that has a set life cycle. For example, one agent might read data from an input/output device (such as the keyboard, a file on disk, or a network connection) and another agent might perform action on that data as it becomes available. The first agent uses message passing to inform the second agent that more data is available. The Concurrency Runtime task scheduler provides an efficient mechanism to enable agents to block and yield cooperatively without requiring less efficient preemption.

The Agents Library defines the `concurrency::agent` class to represent an asynchronous agent. `agent` is an abstract class that declares the virtual method `concurrency::agent::run`. The `run` method executes the task that is performed by the agent. Because `run` is abstract, you must implement this method in every class that you derive from `agent`.

## Agent Life Cycle

Agents have a set life cycle. The `concurrency::agent_status` enumeration defines the various states of an agent. The following illustration is a state diagram that shows how agents progress from one state to another. In this illustration, solid lines represent methods that you call from your application; dotted lines represent methods that are called from the runtime.



The following table describes each state in the `agent_status` enumeration.

AGENT STATE	DESCRIPTION
<code>agent_created</code>	The agent has not been scheduled for execution.
<code>agent_runnable</code>	The runtime is scheduling the agent for execution.
<code>agent_started</code>	The agent has started and is running.
<code>agent_done</code>	The agent finished.
<code>agent_canceled</code>	The agent was canceled before it entered the <code>started</code> state.

`agent_created` is the initial state of an agent, `agent_runnable` and `agent_started` are the active states, and `agent_done` and `agent_canceled` are the terminal states.

Use the `concurrency::agent::status` method to retrieve the current state of an `agent` object. Although the `status`

method is concurrency-safe, the state of the agent can change by the time the `status` method returns. For example, an agent could be in the `agent_started` state when you call the `status` method, but moved to the `agent_done` state just after the `status` method returns.

## Methods and Features

The following table shows some of the important methods that belong to the `agent` class. For more information about all of the `agent` class methods, see [agent Class](#).

METHOD	DESCRIPTION
<code>start</code>	Schedules the <code>agent</code> object for execution and sets it to the <code>agent_runnable</code> state.
<code>run</code>	Executes the task that is to be performed by the <code>agent</code> object.
<code>done</code>	Moves an agent to the <code>agent_done</code> state.
<code>cancel</code>	If the agent was not started, this method cancels execution of the agent and sets it to the <code>agent_canceled</code> state.
<code>status</code>	Retrieves the current state of the <code>agent</code> object.
<code>wait</code>	Waits for the <code>agent</code> object to enter the <code>agent_done</code> or <code>agent_canceled</code> state.
<code>wait_for_all</code>	Waits for all provided <code>agent</code> objects to enter the <code>agent_done</code> or <code>agent_canceled</code> state.
<code>wait_for_one</code>	Waits for at least one of the provided <code>agent</code> objects to enter the <code>agent_done</code> or <code>agent_canceled</code> state.

After you create an agent object, call the `concurrency::agent::start` method to schedule it for execution. The runtime calls the `run` method after it schedules the agent and sets it to the `agent_runnable` state.

The runtime does not manage exceptions that are thrown by asynchronous agents. For more information about exception handling and agents, see [Exception Handling](#).

## Example

For an example that shows how to create a basic agent-based application, see [Walkthrough: Creating an Agent-Based Application](#).

## See also

[Asynchronous Agents Library](#)

# Asynchronous Message Blocks

3/4/2019 • 23 minutes to read • [Edit Online](#)

The Agents Library provides several message-block types that enable you to propagate messages among application components in a thread-safe manner. These message-block types are often used with the various message-passing routines, such as [concurrency::send](#), [concurrency::asend](#), [concurrency::receive](#), and [concurrency::try\\_receive](#). For more information about the message passing routines that are defined by the Agents Library, see [Message Passing Functions](#).

## Sections

This topic contains the following sections:

- [Sources and Targets](#)
- [Message Propagation](#)
- [Overview of Message Block Types](#)
- [unbounded\\_buffer Class](#)
- [overwrite\\_buffer Class](#)
- [single\\_assignment Class](#)
- [call Class](#)
- [transformer Class](#)
- [choice Class](#)
- [join and multitype\\_join Classes](#)
- [timer Class](#)
- [Message Filtering](#)
- [Message Reservation](#)

## Sources and Targets

Sources and targets are two important participants in message passing. A *source* refers to an endpoint of communication that sends messages. A *target* refers to an endpoint of communication that receives messages. You can think of a source as an endpoint that you read from and a target as an endpoint that you write to. Applications connect sources and targets together to form *messaging networks*.

The Agents Library uses two abstract classes to represent sources and targets: [concurrency::ISource](#) and [concurrency::ITarget](#). Message block types that act as sources derive from `ISource`; message block types that act as targets derive from `ITarget`. Message block types that act as sources and targets derive from both `ISource` and `ITarget`.

[\[Top\]](#)

## Message Propagation

*Message propagation* is the act of sending a message from one component to another. When a message block is offered a message, it can accept, decline, or postpone that message. Every message block type stores and transmits messages in different ways. For example, the `unbounded_buffer` class stores an unlimited number of messages, the `overwrite_buffer` class stores a single message at a time, and the transformer class stores an altered version of each message. These message block types are described in more detail later in this document.

When a message block accepts a message, it can optionally perform work and, if the message block is a source, pass the resulting message to another member of the network. A message block can use a filter function to decline messages that it does not want to receive. Filters are described in more detail later in this topic, in the section [Message Filtering](#). A message block that postpones a message can reserve that message and consume it later. Message reservation is described in more detail later in this topic, in the section [Message Reservation](#).

The Agents Library enables message blocks to asynchronously or synchronously pass messages. When you pass a message to a message block synchronously, for example, by using the `send` function, the runtime blocks the current context until the target block either accepts or rejects the message. When you pass a message to a message block asynchronously, for example, by using the `asend` function, the runtime offers the message to the target, and if the target accepts the message, the runtime schedules an asynchronous task that propagates the message to the receiver. The runtime uses lightweight tasks to propagate messages in a cooperative manner. For more information about lightweight tasks, see [Task Scheduler](#).

Applications connect sources and targets together to form messaging networks. Typically, you link the network and call `send` or `asend` to pass data to the network. To connect a source message block to a target, call the `concurrency::ISource::link_target` method. To disconnect a source block from a target, call the `concurrency::ISource::unlink_target` method. To disconnect a source block from all of its targets, call the `concurrency::ISource::unlink_targets` method. When one of the predefined message block types leaves scope or is destroyed, it automatically disconnects itself from any target blocks. Some message block types restrict the maximum number of targets that they can write to. The following section describes the restrictions that apply to the predefined message block types.

[\[Top\]](#)

## Overview of Message Block Types

The following table briefly describes the role of the important message-block types.

### [unbounded\\_buffer](#)

Stores a queue of messages.

### [overwrite\\_buffer](#)

Stores one message that can be written to and read from multiple times.

### [single\\_assignment](#)

Stores one message that can be written to one time and read from multiple times.

### [call](#)

Performs work when it receives a message.

### [transformer](#)

Performs work when it receives data and sends the result of that work to another target block. The `transformer` class can act on different input and output types.

### [choice](#)

Selects the first available message from a set of sources.

### [join and multitype join](#)

Wait for all messages to be received from a set of sources and then combine the messages into one message for another message block.

## timer

Sends a message to a target block on a regular interval.

These message-block types have different characteristics that make them useful for different situations. These are some of the characteristics:

- *Propagation type*: Whether the message block acts as a source of data, a receiver of data, or both.
- *Message ordering*: Whether the message block maintains the original order in which messages are sent or received. Each predefined message block type maintains the original order in which it sends or receives messages.
- *Source count*: The maximum number of sources that the message block can read from.
- *Target count*: The maximum number of targets that the message block can write to.

The following table shows how these characteristics relate to the various message-block types.

MESSAGE BLOCK TYPE	PROPAGATION TYPE (SOURCE, TARGET, OR BOTH)	MESSAGE ORDERING (ORDERED OR UNORDERED)	SOURCE COUNT	TARGET COUNT
<code>unbounded_buffer</code>	Both	Ordered	Unbounded	Unbounded
<code>overwrite_buffer</code>	Both	Ordered	Unbounded	Unbounded
<code>single_assignment</code>	Both	Ordered	Unbounded	Unbounded
<code>call</code>	Target	Ordered	Unbounded	Not Applicable
<code>transformer</code>	Both	Ordered	Unbounded	1
<code>choice</code>	Both	Ordered	10	1
<code>join</code>	Both	Ordered	Unbounded	1
<code>multitype_join</code>	Both	Ordered	10	1
<code>timer</code>	Source	Not Applicable	Not Applicable	1

The following sections describe the message-block types in more detail.

[\[Top\]](#)

## unbounded\_buffer Class

The `concurrency::unbounded_buffer` class represents a general-purpose asynchronous messaging structure. This class stores a first in, first out (FIFO) queue of messages that can be written to by multiple sources or read from by multiple targets. When a target receives a message from an `unbounded_buffer` object, that message is removed from the message queue. Therefore, although an `unbounded_buffer` object can have multiple targets, only one target will receive each message. The `unbounded_buffer` class is useful when you want to pass multiple messages to another component, and that component must receive each message.

### Example

The following example shows the basic structure of how to work with the `unbounded_buffer` class. This example sends three values to an `unbounded_buffer` object and then reads those values back from the same object.

```
// unbounded_buffer-structure.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create an unbounded_buffer object that works with
    // int data.
    unbounded_buffer<int> items;

    // Send a few items to the unbounded_buffer object.
    send(items, 33);
    send(items, 44);
    send(items, 55);

    // Read the items from the unbounded_buffer object and print
    // them to the console.
    wcout << receive(items) << endl;
    wcout << receive(items) << endl;
    wcout << receive(items) << endl;
}
```

This example produces the following output:

```
334455
```

For a complete example that shows how to use the `unbounded_buffer` class, see [How to: Implement Various Producer-Consumer Patterns](#).

[\[Top\]](#)

## overwrite\_buffer Class

The `concurrency::overwrite_buffer` class resembles the `unbounded_buffer` class, except that an `overwrite_buffer` object stores just one message. In addition, when a target receives a message from an `overwrite_buffer` object, that message is not removed from the buffer. Therefore, multiple targets receive a copy of the message.

The `overwrite_buffer` class is useful when you want to pass multiple messages to another component, but that component needs only the most recent value. This class is also useful when you want to broadcast a message to multiple components.

### Example

The following example shows the basic structure of how to work with the `overwrite_buffer` class. This example sends three values to an `overwrite_buffer` object and then reads the current value from the same object three times. This example is similar to the example for the `unbounded_buffer` class. However, the `overwrite_buffer` class stores just one message. In addition, the runtime does not remove the message from an `overwrite_buffer` object after it is read.

```

// overwrite_buffer-structure.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create an overwrite_buffer object that works with
    // int data.
    overwrite_buffer<int> item;

    // Send a few items to the overwrite_buffer object.
    send(item, 33);
    send(item, 44);
    send(item, 55);

    // Read the current item from the overwrite_buffer object and print
    // it to the console three times.
    wcout << receive(item) << endl;
    wcout << receive(item) << endl;
    wcout << receive(item) << endl;
}

```

This example produces the following output:

```
555555
```

For a complete example that shows how to use the `overwrite_buffer` class, see [How to: Implement Various Producer-Consumer Patterns](#).

[\[Top\]](#)

## single\_assignment Class

The `concurrency::single_assignment` class resembles the `overwrite_buffer` class, except that a `single_assignment` object can be written to one time only. Like the `overwrite_buffer` class, when a target receives a message from a `single_assignment` object, that message is not removed from that object. Therefore, multiple targets receive a copy of the message. The `single_assignment` class is useful when you want to broadcast one message to multiple components.

### Example

The following example shows the basic structure of how to work with the `single_assignment` class. This example sends three values to a `single_assignment` object and then reads the current value from the same object three times. This example is similar to the example for the `overwrite_buffer` class. Although both the `overwrite_buffer` and `single_assignment` classes store a single message, the `single_assignment` class can be written to one time only.

```

// single_assignment-structure.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create an single_assignment object that works with
    // int data.
    single_assignment<int> item;

    // Send a few items to the single_assignment object.
    send(item, 33);
    send(item, 44);
    send(item, 55);

    // Read the current item from the single_assignment object and print
    // it to the console three times.
    wcout << receive(item) << endl;
    wcout << receive(item) << endl;
    wcout << receive(item) << endl;
}

```

This example produces the following output:

```
333333
```

For a complete example that shows how to use the `single_assignment` class, see [Walkthrough: Implementing Futures](#).

[\[Top\]](#)

## call Class

The `concurrency::call` class acts as a message receiver that performs a work function when it receives data. This work function can be a lambda expression, a function object, or a function pointer. A `call` object behaves differently than an ordinary function call because it acts in parallel to other components that send messages to it. If a `call` object is performing work when it receives a message, it adds that message to a queue. Every `call` object processes queued messages in the order in which they are received.

### Example

The following example shows the basic structure of how to work with the `call` class. This example creates a `call` object that prints each value that it receives to the console. The example then sends three values to the `call` object. Because the `call` object processes messages on a separate thread, this example also uses a counter variable and an `event` object to ensure that the `call` object processes all messages before the `wmain` function returns.

```

// call-structure.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // An event that is set when the call object receives all values.
    event received_all;

    // Counts the
    long receive_count = 0L;
    long max_receive_count = 3L;

    // Create an call object that works with int data.
    call<int> target([&received_all,&receive_count,max_receive_count](int n) {
        // Print the value that the call object receives to the console.
        wcout << n << endl;

        // Set the event when all messages have been processed.
        if (++receive_count == max_receive_count)
            received_all.set();
    });

    // Send a few items to the call object.
    send(target, 33);
    send(target, 44);
    send(target, 55);

    // Wait for the call object to process all items.
    received_all.wait();
}

```

This example produces the following output:

```
334455
```

For a complete example that shows how to use the `call` class, see [How to: Provide Work Functions to the call and transformer Classes](#).

[\[Top\]](#)

## transformer Class

The `concurrency::transformer` class acts as both a message receiver and as a message sender. The `transformer` class resembles the `call` class because it performs a user-defined work function when it receives data. However, the `transformer` class also sends the result of the work function to receiver objects. Like a `call` object, a `transformer` object acts in parallel to other components that send messages to it. If a `transformer` object is performing work when it receives a message, it adds that message to a queue. Every `transformer` object processes its queued messages in the order in which they are received.

The `transformer` class sends its message to one target. If you set the `_PTarget` parameter in the constructor to `NULL`, you can later specify the target by calling the `concurrency::link_target` method.

Unlike all other asynchronous message block types that are provided by the Agents Library, the `transformer` class can act on different input and output types. This ability to transform data from one type to another makes the `transformer` class a key component in many concurrent networks. In addition, you

can add more fine-grained parallel functionality in the work function of a `transformer` object.

## Example

The following example shows the basic structure of how to work with the `transformer` class. This example creates a `transformer` object that multiplies each input `int` value by 0.33 in order to produce a `double` value as output. The example then receives the transformed values from the same `transformer` object and prints them to the console.

```
// transformer-structure.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create an transformer object that receives int data and
    // sends double data.
    transformer<int, double> third([](int n) {
        // Return one-third of the input value.
        return n * 0.33;
    });

    // Send a few items to the transformer object.
    send(third, 33);
    send(third, 44);
    send(third, 55);

    // Read the processed items from the transformer object and print
    // them to the console.
    wcout << receive(third) << endl;
    wcout << receive(third) << endl;
    wcout << receive(third) << endl;
}
```

This example produces the following output:

```
10.8914.5218.15
```

For a complete example that shows how to use the `transformer` class, see [How to: Use transformer in a Data Pipeline](#).

[\[Top\]](#)

## choice Class

The `concurrency::choice` class selects the first available message from a set of sources. The `choice` class represents a control-flow mechanism instead of a dataflow mechanism (the topic [Asynchronous Agents Library](#) describes the differences between dataflow and control-flow).

Reading from a choice object resembles calling the Windows API function `WaitForMultipleObjects` when it has the `bWaitAll` parameter set to `FALSE`. However, the `choice` class binds data to the event itself instead of to an external synchronization object.

Typically, you use the `choice` class together with the `concurrency::receive` function to drive control-flow in your application. Use the `choice` class when you have to select among message buffers that have different types. Use the `single_assignment` class when you have to select among message buffers that have the same

type.

The order in which you link sources to a `choice` object is important because it can determine which message is selected. For example, consider the case where you link multiple message buffers that already contain a message to a `choice` object. The `choice` object selects the message from the first source that it is linked to. After you link all sources, the `choice` object preserves the order in which each source receives a message.

### **Example**

The following example shows the basic structure of how to work with the `choice` class. This example uses the `concurrency::make_choice` function to create a `choice` object that selects among three message blocks. The example then computes various Fibonacci numbers and stores each result in a different message block. The example then prints to the console a message that is based on the operation that finished first.

```

// choice-structure.cpp
// compile with: /EHsc
#include <agents.h>
#include <ppl.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Computes the nth Fibonacci number.
// This function illustrates a lengthy operation and is therefore
// not optimized for performance.
int fibonacci(int n)
{
    if (n < 2)
        return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

int wmain()
{
    // Although the following three message blocks are written to one time only,
    // this example illustrates the fact that the choice class works with
    // different message block types.

    // Holds the 35th Fibonacci number.
    single_assignment<int> fib35;
    // Holds the 37th Fibonacci number.
    overwrite_buffer<int> fib37;
    // Holds half of the 42nd Fibonacci number.
    unbounded_buffer<double> half_of_fib42;

    // Create a choice object that selects the first single_assignment
    // object that receives a value.
    auto select_one = make_choice(&fib35, &fib37, &half_of_fib42);

    // Execute a few lengthy operations in parallel. Each operation sends its
    // result to one of the single_assignment objects.
    parallel_invoke(
        [&fib35] { send(fib35, fibonacci(35)); },
        [&fib37] { send(fib37, fibonacci(37)); },
        [&half_of_fib42] { send(half_of_fib42, fibonacci(42) * 0.5); }
    );

    // Print a message that is based on the operation that finished first.
    switch (receive(select_one))
    {
    case 0:
        wcout << L"fib35 received its value first. Result = "
                << receive(fib35) << endl;
        break;
    case 1:
        wcout << L"fib37 received its value first. Result = "
                << receive(fib37) << endl;
        break;
    case 2:
        wcout << L"half_of_fib42 received its value first. Result = "
                << receive(half_of_fib42) << endl;
        break;
    default:
        wcout << L"Unexpected." << endl;
        break;
    }
}

```

This example produces the following sample output:

```
fib35 received its value first. Result = 9227465
```

Because the task that computes the 35<sup>th</sup> Fibonacci number is not guaranteed to finish first, the output of this example can vary.

This example uses the `concurrency::parallel_invoke` algorithm to compute the Fibonacci numbers in parallel. For more information about `parallel_invoke`, see [Parallel Algorithms](#).

For a complete example that shows how to use the `choice` class, see [How to: Select Among Completed Tasks](#).

[\[Top\]](#)

## join and multitype\_join Classes

The `concurrency::join` and `concurrency::multitype_join` classes let you wait for each member of a set of sources to receive a message. The `join` class acts on source objects that have a common message type. The `multitype_join` class acts on source objects that can have different message types.

Reading from a `join` or `multitype_join` object resembles calling the Windows API function `WaitForMultipleObjects` when it has the `bWaitAll` parameter set to `TRUE`. However, just like a `choice` object, `join` and `multitype_join` objects use an event mechanism that binds data to the event itself instead of to an external synchronization object.

Reading from a `join` object produces a `std::vector` object. Reading from a `multitype_join` object produces a `std::tuple` object. Elements appear in these objects in the same order as their corresponding source buffers are linked to the `join` or `multitype_join` object. Because the order in which you link source buffers to a `join` or `multitype_join` object is associated with the order of elements in the resulting `vector` or `tuple` object, we recommend that you do not unlink an existing source buffer from a join. Doing so can result in unspecified behavior.

### Greedy Versus Non-Greedy Joins

The `join` and `multitype_join` classes support the concept of greedy and non-greedy joins. A *greedy join* accepts a message from each of its sources as messages become available until all message are available. A *non-greedy join* receives messages in two phases. First, a non-greedy join waits until it is offered a message from each of its sources. Second, after all source messages are available, a non-greedy join attempts to reserve each of those messages. If it can reserve each message, it consumes all messages and propagates them to its target. Otherwise, it releases, or cancels, the message reservations and again waits for each source to receive a message.

Greedy joins perform better than non-greedy joins because they accept messages immediately. However, in rare cases, greedy joins can lead to deadlocks. Use a non-greedy join when you have multiple joins that contain one or more shared source objects.

### Example

The following example shows the basic structure of how to work with the `join` class. This example uses the `concurrency::make_join` function to create a `join` object that receives from three `single_assignment` objects. This example computes various Fibonacci numbers, stores each result in a different `single_assignment` object, and then prints to the console each result that the `join` object holds. This example is similar to the example for the `choice` class, except that the `join` class waits for all source message blocks to receive a message.

```

// join-structure.cpp
// compile with: /EHsc
#include <agents.h>
#include <ppl.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Computes the nth Fibonacci number.
// This function illustrates a lengthy operation and is therefore
// not optimized for performance.
int fibonacci(int n)
{
    if (n < 2)
        return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

int wmain()
{
    // Holds the 35th Fibonacci number.
    single_assignment<int> fib35;
    // Holds the 37th Fibonacci number.
    single_assignment<int> fib37;
    // Holds half of the 42nd Fibonacci number.
    single_assignment<double> half_of_fib42;

    // Create a join object that selects the values from each of the
    // single_assignment objects.
    auto join_all = make_join(&fib35, &fib37, &half_of_fib42);

    // Execute a few lengthy operations in parallel. Each operation sends its
    // result to one of the single_assignment objects.
    parallel_invoke(
        [&fib35] { send(fib35, fibonacci(35)); },
        [&fib37] { send(fib37, fibonacci(37)); },
        [&half_of_fib42] { send(half_of_fib42, fibonacci(42) * 0.5); }
    );

    auto result = receive(join_all);
    wcout << L"fib35 = " << get<0>(result) << endl;
    wcout << L"fib37 = " << get<1>(result) << endl;
    wcout << L"half_of_fib42 = " << get<2>(result) << endl;
}

```

This example produces the following output:

```

fib35 = 9227465fib37 = 24157817half_of_fib42 = 1.33957e+008

```

This example uses the `concurrency::parallel_invoke` algorithm to compute the Fibonacci numbers in parallel. For more information about `parallel_invoke`, see [Parallel Algorithms](#).

For complete examples that show how to use the `join` class, see [How to: Select Among Completed Tasks](#) and [Walkthrough: Using join to Prevent Deadlock](#).

[\[Top\]](#)

## timer Class

The `concurrency::timer class` acts as a message source. A `timer` object sends a message to a target after a specified period of time has elapsed. The `timer` class is useful when you must delay sending a message or

you want to send a message at a regular interval.

The `timer` class sends its message to just one target. If you set the `_Ptarget` parameter in the constructor to `NULL`, you can later specify the target by calling the `concurrency::Source::link_target` method.

A `timer` object can be repeating or non-repeating. To create a repeating timer, pass **true** for the `_Repeating` parameter when you call the constructor. Otherwise, pass **false** for the `_Repeating` parameter to create a non-repeating timer. If the timer is repeating, it sends the same message to its target after each interval.

The Agents Library creates `timer` objects in the non-started state. To start a timer object, call the `concurrency::timer::start` method. To stop a `timer` object, destroy the object or call the `concurrency::timer::stop` method. To pause a repeating timer, call the `concurrency::timer::pause` method.

## Example

The following example shows the basic structure of how to work with the `timer` class. The example uses `timer` and `call` objects to report the progress of a lengthy operation.

```
// timer-structure.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Computes the nth Fibonacci number.
// This function illustrates a lengthy operation and is therefore
// not optimized for performance.
int fibonacci(int n)
{
    if (n < 2)
        return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

int wmain()
{
    // Create a call object that prints characters that it receives
    // to the console.
    call<wchar_t> print_character([](wchar_t c) {
        wcout << c;
    });

    // Create a timer object that sends the period (.) character to
    // the call object every 100 milliseconds.
    timer<wchar_t> progress_timer(100u, L'.', &print_character, true);

    // Start the timer.
    wcout << L"Computing fib(42)";
    progress_timer.start();

    // Compute the 42nd Fibonacci number.
    int fib42 = fibonacci(42);

    // Stop the timer and print the result.
    progress_timer.stop();
    wcout << endl << L"result is " << fib42 << endl;
}
```

This example produces the following sample output:

```
Computing fib(42).....result is 267914296
```

For a complete example that shows how to use the `timer` class, see [How to: Send a Message at a Regular Interval](#).

[\[Top\]](#)

## Message Filtering

When you create a message block object, you can supply a *filter function* that determines whether the message block accepts or rejects a message. A filter function is a useful way to guarantee that a message block receives only certain values.

The following example shows how to create an `unbounded_buffer` object that uses a filter function to accept only even numbers. The `unbounded_buffer` object rejects odd numbers, and therefore does not propagate odd numbers to its target blocks.

```
// filter-function.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create an unbounded_buffer object that uses a filter
    // function to accept only even numbers.
    unbounded_buffer<int> accept_evens(
        [](int n) {
            return (n%2) == 0;
        });

    // Send a few values to the unbounded_buffer object.
    unsigned int accept_count = 0;
    for (int i = 0; i < 10; ++i)
    {
        // The asend function returns true only if the target
        // accepts the message. This enables us to determine
        // how many elements are stored in the unbounded_buffer
        // object.
        if (asend(accept_evens, i))
        {
            ++accept_count;
        }
    }

    // Print to the console each value that is stored in the
    // unbounded_buffer object. The unbounded_buffer object should
    // contain only even numbers.
    while (accept_count > 0)
    {
        wcout << receive(accept_evens) << L' ';
        --accept_count;
    }
}
```

This example produces the following output:

```
0 2 4 6 8
```

A filter function can be a lambda function, a function pointer, or a function object. Every filter function takes one of the following forms.

```
bool (T)  
bool (T const &)
```

To eliminate the unnecessary copying of data, use the second form when you have an aggregate type that is propagated by value.

Message filtering supports the *dataflow* programming model, in which components perform computations when they receive data. For examples that use filter functions to control the flow of data in a message passing network, see [How to: Use a Message Block Filter](#), [Walkthrough: Creating a Dataflow Agent](#), and [Walkthrough: Creating an Image-Processing Network](#).

[\[Top\]](#)

## Message Reservation

*Message reservation* enables a message block to reserve a message for later use. Typically, message reservation is not used directly. However, understanding message reservation can help you better understand the behavior of some of the predefined message block types.

Consider non-greedy and greedy joins. Both of these use message reservation to reserve messages for later use. As described earlier, a non-greedy join receives messages in two phases. During the first phase, a non-greedy `join` object waits for each of its sources to receive a message. A non-greedy join then attempts to reserve each of those messages. If it can reserve each message, it consumes all messages and propagates them to its target. Otherwise, it releases, or cancels, the message reservations and again waits for each source to receive a message.

A greedy join, which also reads input messages from a number of sources, uses message reservation to read additional messages while it waits to receive a message from each source. For example, consider a greedy join that receives messages from message blocks `A` and `B`. If the greedy join receives two messages from `B` but has not yet received a message from `A`, the greedy join saves the unique message identifier for the second message from `B`. After the greedy join receives a message from `A` and propagates out these messages, it uses the saved message identifier to see if the second message from `B` is still available.

You can use message reservation when you implement your own custom message block types. For an example about how to create a custom message block type, see [Walkthrough: Creating a Custom Message Block](#).

[\[Top\]](#)

## See also

[Asynchronous Agents Library](#)

# Message Passing Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

The Asynchronous Agents Library provides several functions that let you pass messages among components.

These message-passing functions are used with the various message-block types. For more information about the message-block types that are defined by the Concurrency Runtime, see [Asynchronous Message Blocks](#).

## Sections

This topic describes the following message-passing functions:

- [send and asend](#)
- [receive and try\\_receive](#)
- [Examples](#)

## send and asend

The `concurrency::send` function sends a message to the specified target synchronously and the `concurrency::asend` function sends a message to the specified target asynchronously. Both the `send` and `asend` functions wait until the target indicates that it will eventually accept or decline the message.

The `send` function waits until the target accepts or declines the message before it returns. The `send` function returns **true** if the message was delivered and **false** otherwise. Because the `send` function works synchronously, the `send` function waits for the target to receive the message before it returns.

Conversely, the `asend` function does not wait for the target to accept or decline the message before it returns. Instead, the `asend` function returns **true** if the target accepts the message and will eventually take it. Otherwise, `asend` returns **false** to indicate that the target either declined the message or postponed the decision about whether to take the message.

[\[Top\]](#)

## receive and try\_receive

The `concurrency::receive` and `concurrency::try_receive` functions read data from a given source. The `receive` function waits for data to become available, whereas the `try_receive` function returns immediately.

Use the `receive` function when you must have the data to continue. Use the `try_receive` function if you must not block the current context or you do not have to have the data to continue.

[\[Top\]](#)

## Examples

For examples that use the `send` and `asend`, and `receive` functions, see the following topics:

- [Asynchronous Message Blocks](#)
- [How to: Implement Various Producer-Consumer Patterns](#)
- [How to: Provide Work Functions to the call and transformer Classes](#)

- [How to: Use transformer in a Data Pipeline](#)
- [How to: Select Among Completed Tasks](#)
- [How to: Send a Message at a Regular Interval](#)
- [How to: Use a Message Block Filter](#)

[\[Top\]](#)

## See also

[Asynchronous Agents Library](#)

[Asynchronous Message Blocks](#)

[send Function](#)

[asend Function](#)

[receive Function](#)

[try\\_receive Function](#)

# How to: Implement Various Producer-Consumer Patterns

3/4/2019 • 5 minutes to read • [Edit Online](#)

This topic describes how to implement the producer-consumer pattern in your application. In this pattern, the *producer* sends messages to a message block, and the *consumer* reads messages from that block.

The topic demonstrates two scenarios. In the first scenario, the consumer must receive each message that the producer sends. In the second scenario, the consumer periodically polls for data, and therefore does not have to receive each message.

Both examples in this topic use agents, message blocks, and message-passing functions to transmit messages from the producer to the consumer. The producer agent uses the `concurrency::send` function to write messages to a `concurrency::ITarget` object. The consumer agent uses the `concurrency::receive` function to read messages from a `concurrency::ISource` object. Both agents hold a sentinel value to coordinate the end of processing.

For more information about asynchronous agents, see [Asynchronous Agents](#). For more information about message blocks and message-passing functions, see [Asynchronous Message Blocks](#) and [Message Passing Functions](#).

## Example

In this example, the producer agent sends a series of numbers to the consumer agent. The consumer receives each of these numbers and computes their average. The application writes the average to the console.

This example uses a `concurrency::unbounded_buffer` object to enable the producer to queue messages. The `unbounded_buffer` class implements `ITarget` and `ISource` so that the producer and the consumer can send and receive messages to and from a shared buffer. The `send` and `receive` functions coordinate the task of propagating the data from the producer to the consumer.

```
// producer-consumer-average.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Demonstrates a basic agent that produces values.
class producer_agent : public agent
{
public:
    explicit producer_agent(ITarget<int>& target, unsigned int count, int sentinel)
        : _target(target)
        , _count(count)
        , _sentinel(sentinel)
    {
    }
protected:
    void run()
    {
        // Send the value of each loop iteration to the target buffer.
        while (_count > 0)
        {
            send(_target, static_cast<int>(_count));
            -- count;
        }
    }
};
```

```

    }
    // Send the sentinel value.
    send(_target, _sentinel);

    // Set the agent to the finished state.
    done();
}
private:
    // The target buffer to write to.
    ITarget<int>& _target;
    // The number of values to send.
    unsigned int _count;
    // The sentinel value, which informs the consumer agent to stop processing.
    int _sentinel;
};

// Demonstrates a basic agent that consumes values.
class consumer_agent : public agent
{
public:
    explicit consumer_agent(ISource<int>& source, int sentinel)
        : _source(source)
        , _sentinel(sentinel)
    {
    }

    // Retrieves the average of all received values.
    int average()
    {
        return receive(_average);
    }
protected:
    void run()
    {
        // The sum of all values.
        int sum = 0;
        // The count of values received.
        int count = 0;

        // Read from the source block until we receive the
        // sentinel value.
        int n;
        while ((n = receive(_source)) != _sentinel)
        {
            sum += n;
            ++count;
        }

        // Write the average to the message buffer.
        send(_average, sum / count);

        // Set the agent to the finished state.
        done();
    }
private:
    // The source buffer to read from.
    ISource<int>& _source;
    // The sentinel value, which informs the agent to stop processing.
    int _sentinel;
    // Holds the average of all received values.
    single_assignment<int> _average;
};

int wmain()
{
    // Informs the consumer agent to stop processing.
    const int sentinel = 0;
    // The number of values for the producer agent to send.
    const unsigned int count = 100;

```

```

const unsigned int count = 100,

// A message buffer that is shared by the agents.
unbounded_buffer<int> buffer;

// Create and start the producer and consumer agents.
producer_agent producer(buffer, count, sentinel);
consumer_agent consumer(buffer, sentinel);
producer.start();
consumer.start();

// Wait for the agents to finish.
agent::wait(&producer);
agent::wait(&consumer);

// Print the average.
wcout << L"The average is " << consumer.average() << L'.' << endl;
}

```

This example produces the following output.

```
The average is 50.
```

## Example

In this example, the producer agent sends a series of stock quotes to the consumer agent. The consumer agent periodically reads the current quote and prints it to the console.

This example resembles the previous one, except that it uses a `concurrency::overwrite_buffer` object to enable the producer to share one message with the consumer. As in the previous example, `overwrite_buffer` class implements `ITarget` and `ISource` so that the producer and the consumer can act on a shared message buffer.

```

// producer-consumer-quotes.cpp
// compile with: /EHsc
#include <agents.h>
#include <array>
#include <algorithm>
#include <iostream>

using namespace concurrency;
using namespace std;

// Demonstrates a basic agent that produces values.
class producer_agent : public agent
{
public:
    explicit producer_agent(ITarget<double>& target)
        : _target(target)
    {
    }
protected:
    void run()
    {
        // For illustration, create a predefined array of stock quotes.
        // A real-world application would read these from an external source,
        // such as a network connection or a database.
        array<double, 6> quotes = { 24.44, 24.65, 24.99, 23.76, 22.30, 25.89 };

        // Send each quote to the target buffer.
        for_each (begin(quotes), end(quotes), [&] (double quote) {

            send(_target, quote);

            // Pause before sending the next quote.

```

```

        // Pause before sending the next quote.
        concurrency::wait(20);
    });
    // Send a negative value to indicate the end of processing.
    send(_target, -1.0);

    // Set the agent to the finished state.
    done();
}
private:
    // The target buffer to write to.
    ITarget<double>& _target;
};

// Demonstrates a basic agent that consumes values.
class consumer_agent : public agent
{
public:
    explicit consumer_agent(ISource<double>& source)
        : _source(source)
    {
    }

protected:
    void run()
    {
        // Read quotes from the source buffer until we receive
        // a negative value.
        double quote;
        while ((quote = receive(_source)) >= 0.0)
        {
            // Print the quote.
            wcout.setf(ios::fixed);
            wcout.precision(2);
            wcout << L"Current quote is " << quote << L'.' << endl;

            // Pause before reading the next quote.
            concurrency::wait(10);
        }

        // Set the agent to the finished state.
        done();
    }
private:
    // The source buffer to read from.
    ISource<double>& _source;
};

int wmain()
{
    // A message buffer that is shared by the agents.
    overwrite_buffer<double> buffer;

    // Create and start the producer and consumer agents.
    producer_agent producer(buffer);
    consumer_agent consumer(buffer);
    producer.start();
    consumer.start();

    // Wait for the agents to finish.
    agent::wait(&producer);
    agent::wait(&consumer);
}

```

This example produces the following sample output.

```
Current quote is 24.44.  
Current quote is 24.44.  
Current quote is 24.65.  
Current quote is 24.99.  
Current quote is 23.76.  
Current quote is 22.30.  
Current quote is 25.89.
```

Unlike with an `unbounded_buffer` object, the `receive` function does not remove the message from the `overwrite_buffer` object. If the consumer reads from the message buffer more than one time before the producer overwrites that message, the receiver obtains the same message every time.

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `producer-consumer.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc producer-consumer.cpp
```

## See also

- [Asynchronous Agents Library](#)
- [Asynchronous Agents](#)
- [Asynchronous Message Blocks](#)
- [Message Passing Functions](#)

# How to: Provide Work Functions to the call and transformer Classes

3/4/2019 • 3 minutes to read • [Edit Online](#)

This topic illustrates several ways to provide work functions to the `concurrency::call` and `concurrency::transformer` classes.

The first example shows how to pass a lambda expression to a `call` object. The second example shows how to pass a function object to a `call` object. The third example shows how to bind a class method to a `call` object.

For illustration, every example in this topic uses the `call` class. For an example that uses the `transformer` class, see [How to: Use transformer in a Data Pipeline](#).

## Example

The following example shows a common way to use the `call` class. This example passes a lambda function to the `call` constructor.

```
// call-lambda.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Stores the result of the computation.
    single_assignment<int> result;

    // Pass a lambda function to a call object that computes the square
    // of its input and then sends the result to the message buffer.
    call<int> c([&](int n) {
        send(result, n * n);
    });

    // Send a message to the call object and print the result.
    send(c, 13);
    wcout << L"13 squared is " << receive(result) << L'.' << endl;
}
```

This example produces the following output.

```
13 squared is 169.
```

## Example

The following example resembles the previous one, except that it uses the `call` class together with a function object (functor).

```

// call-functor.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Functor class that computes the square of its input.
class square
{
public:
    explicit square(ITarget<int>& target)
        : _target(target)
    {
    }

    // Function call operator for the functor class.
    void operator()(int n)
    {
        send(_target, n * n);
    }

private:
    ITarget<int>& _target;
};

int wmain()
{
    // Stores the result of the computation.
    single_assignment<int> result;

    // Pass a function object to the call constructor.
    square s(result);
    call<int> c(s);

    // Send a message to the call object and print the result.
    send(c, 13);
    wcout << L"13 squared is " << receive(result) << L'.' << endl;
}

```

## Example

The following example resembles the previous one, except that it uses the `std::bind1st` and `std::mem_fun` functions to bind a `call` object to a class method.

Use this technique if you have to bind a `call` or `transformer` object to a specific class method instead of the function call operator, `operator()`.

```

// call-method.cpp
// compile with: /EHsc
#include <agents.h>
#include <functional>
#include <iostream>

using namespace concurrency;
using namespace std;

// Class that computes the square of its input.
class square
{
public:
    explicit square(ITarget<int>& target)
        : _target(target)
    {
    }

    // Method that computes the square of its input.
    void square_value(int n)
    {
        send(_target, n * n);
    }

private:
    ITarget<int>& _target;
};

int wmain()
{
    // Stores the result of the computation.
    single_assignment<int> result;

    // Bind a class method to a call object.
    square s(result);
    call<int> c(bind1st(mem_fun(&square::square_value), &s));

    // Send a message to the call object and print the result.
    send(c, 13);
    wcout << L"13 squared is " << receive(result) << L'.' << endl;
}

```

You can also assign the result of the `bind1st` function to a `std::function` object or use the `auto` keyword, as shown in the following example.

```

// Assign to a function object.
function<void(int)> f1 = bind1st(mem_fun(&square::square_value), &s);
call<int> c1(f1);

// Alternatively, use the auto keyword to have the compiler deduce the type.
auto f2 = bind1st(mem_fun(&square::square_value), &s);
call<int> c2(f2);

```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `call.cpp` and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc call.cpp**

See also

[Asynchronous Agents Library](#)

[Asynchronous Message Blocks](#)

[How to: Use transformer in a Data Pipeline](#)

[call Class](#)

[transformer Class](#)

# How to: Use transformer in a Data Pipeline

3/4/2019 • 2 minutes to read • [Edit Online](#)

This topic contains a basic example that shows how to use the `concurrency::transformer` class in a data pipeline. For a more complete example that uses a data pipeline to perform image processing, see [Walkthrough: Creating an Image-Processing Network](#).

*Data pipelining* is a common pattern in concurrent programming. A data pipeline consists of a series of stages, where each stage performs work and then passes the result of that work to the next stage. The `transformer` class is a key component in data pipelines because it receives an input value, performs work on that value, and then produces a result for another component to use.

## Example

This example uses the following data pipeline to perform a series of transformations given an initial input value:

1. The first stage calculates the absolute value of its input.
2. The second stage calculates the square root of its input.
3. The third stage computes the square of its input.
4. The fourth stage negates its input.
5. The fifth stage writes the final result to a message buffer.

Finally, the example prints the result of the pipeline to the console.

```

// data-pipeline.cpp
// compile with: /EHsc
#include <agents.h>
#include <math.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Computes the absolute value of its input.
    transformer<int, int> t0([](int n) {
        return abs(n);
    });

    // Computes the square root of its input.
    transformer<int, double> t1([](int n) {
        return sqrt(static_cast<double>(n));
    });

    // Computes the square its input.
    transformer<double, int> t2([](double n) {
        return static_cast<int>(n * n);
    });

    // Negates its input.
    transformer<int, int> t3([](int n) {
        return -n;
    });

    // Holds the result of the pipeline computation.
    single_assignment<int> result;

    // Link together each stage of the pipeline.
    // t0 -> t1 -> t2 -> t3 -> result
    t0.link_target(&t1);
    t1.link_target(&t2);
    t2.link_target(&t3);
    t3.link_target(&result);

    // Propagate a message through the pipeline.
    send(t0, -42);

    // Print the result to the console.
    wcout << L"The result is " << receive(result) << L'.' << endl;
}

```

This example produces the following output:

```
The result is -42.
```

It is common for a stage in a data pipeline to output a value whose type differs from its input value. In this example, the second stage takes a value of type `int` as its input and produces the square root of that value (a `double`) as its output.

#### NOTE

The data pipeline in this example is for illustration. Because each transformation operation performs little work, the overhead that is required to perform message-passing can outweigh the benefits of using a data pipeline.

# Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `data-pipeline.cpp` and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc data-pipeline.cpp**

## See also

[Asynchronous Agents Library](#)

[Asynchronous Message Blocks](#)

[Walkthrough: Creating an Image-Processing Network](#)

# How to: Select Among Completed Tasks

3/4/2019 • 4 minutes to read • [Edit Online](#)

This example shows how to use the `concurrency::choice` and `concurrency::join` classes to select the first task to complete a search algorithm.

## Example

The following example performs two search algorithms in parallel and selects the first algorithm to complete. This example defines the `employee` type, which holds a numeric identifier and a salary for an employee. The `find_employee` function finds the first employee that has the provided identifier or the provided salary. The `find_employee` function also handles the case where no employee has the provided identifier or salary. The `wmain` function creates an array of `employee` objects and searches for several identifier and salary values.

The example uses a `choice` object to select among the following cases:

1. An employee who has the provided identifier exists.
2. An employee who has the provided salary exists.
3. No employee who has the provided identifier or salary exists.

For the first two cases, the example uses a `concurrency::single_assignment` object to hold the identifier and another `single_assignment` object to hold the salary. The example uses a `join` object for the third case. The `join` object is composed of two additional `single_assignment` objects, one for the case where no employee who has the provided identifier exists, and one for the case where no employee who has the provided salary exists. The `join` object sends a message when each of its members receives a message. In this example, the `join` object sends a message when no employee who has the provided identifier or salary exists.

The example uses a `concurrency::structured_task_group` object to run both search algorithms in parallel. Each search task writes to one of the `single_assignment` objects to indicate whether the given employee exists. The example uses the `concurrency::receive` function to obtain the index of the first buffer that contains a message and a `switch` block to print the result.

```
// find-employee.cpp
// compile with: /EHsc
#include <agents.h>
#include <ppl.h>
#include <array>
#include <iostream>
#include <random>

using namespace concurrency;
using namespace std;

// Contains information about an employee.
struct employee
{
    int id;
    float salary;
};

// Finds the first employee that has the provided id or salary.
template <typename T>
void find_employee(const T& employees, int id, float salary)
{
```

```

// Holds the salary for the employee with the provided id.
single_assignment<float> find_id_result;

// Holds the id for the employee with the provided salary.
single_assignment<int> find_salary_result;

// Holds a message if no employee with the provided id exists.
single_assignment<bool> id_not_found;

// Holds a message if no employee with the provided salary exists.
single_assignment<bool> salary_not_found;

// Create a join object for the "not found" buffers.
// This join object sends a message when both its members holds a message
// (in other words, no employee with the provided id or salary exists).
auto not_found = make_join(&id_not_found, &salary_not_found);

// Create a choice object to select among the following cases:
// 1. An employee with the provided id exists.
// 2. An employee with the provided salary exists.
// 3. No employee with the provided id or salary exists.
auto selector = make_choice(&find_id_result, &find_salary_result, &not_found);

// Create a task that searches for the employee with the provided id.
auto search_id_task = make_task([&]{
    auto result = find_if(begin(employees), end(employees),
        [&](const employee& e) { return e.id == id; });
    if (result != end(employees))
    {
        // The id was found, send the salary to the result buffer.
        send(find_id_result, result->salary);
    }
    else
    {
        // The id was not found.
        send(id_not_found, true);
    }
});

// Create a task that searches for the employee with the provided salary.
auto search_salary_task = make_task([&]{
    auto result = find_if(begin(employees), end(employees),
        [&](const employee& e) { return e.salary == salary; });
    if (result != end(employees))
    {
        // The salary was found, send the id to the result buffer.
        send(find_salary_result, result->id);
    }
    else
    {
        // The salary was not found.
        send(salary_not_found, true);
    }
});

// Use a structured_task_group object to run both tasks.
structured_task_group tasks;
tasks.run(search_id_task);
tasks.run(search_salary_task);

wcout.setf(ios::fixed, ios::fixed);
wcout.precision(2);

// Receive the first object that holds a message and print a message.
int index = receive(selector);
switch (index)

```

```

{
case 0:
    wcout << L"Employee with id " << id << L" has salary "
        << receive(find_id_result);
    break;
case 1:
    wcout << L"Employee with salary " << salary << L" has id "
        << receive(find_salary_result);
    break;
case 2:
    wcout << L"No employee has id " << id << L" or salary " << salary;
    break;
default:
    __assume(0);
}
wcout << L'.' << endl;

// Cancel any active tasks and wait for the task group to finish.
tasks.cancel();
tasks.wait();
}

int wmain()
{
    // Create an array of employees and assign each one a
    // random id and salary.

    array<employee, 10000> employees;

    mt19937 gen(15);
    const float base_salary = 25000.0f;
    for (int i = 0; i < employees.size(); ++i)
    {
        employees[i].id = gen()%100000;

        float bonus = static_cast<float>(gen()%5000);
        employees[i].salary = base_salary + bonus;
    }

    // Search for several id and salary values.

    find_employee(employees, 14758, 30210.00);
    find_employee(employees, 340, 29150.00);
    find_employee(employees, 61935, 29255.90);
    find_employee(employees, 899, 31223.00);
}

```

This example produces the following output.

```

Employee with id 14758 has salary 27780.00.
Employee with salary 29150.00 has id 84345.
Employee with id 61935 has salary 29905.00.
No employee has id 899 or salary 31223.00.

```

This example uses the [concurrency::make\\_choice](#) helper function to create `choice` objects and the [concurrency::make\\_join](#) helper function to create `join` objects.

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `find-employee.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc find-employee.cpp
```

## See also

[Asynchronous Agents Library](#)

[Asynchronous Message Blocks](#)

[Message Passing Functions](#)

[choice Class](#)

[join Class](#)

# How to: Send a Message at a Regular Interval

3/4/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use the `concurrency::timer` class to send a message at a regular interval.

## Example

The following example uses a `timer` object to report progress during a lengthy operation. This example links the `timer` object to a `concurrency::call` object. The `call` object prints a progress indicator to the console at a regular interval. The `concurrency::timer::start` method runs the timer on a separate context. The `perform_lengthy_operation` function calls the `concurrency::wait` function on the main context to simulate a time-consuming operation.

```
// report-progress.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Simulates a lengthy operation.
void perform_lengthy_operation()
{
    // Yield the current context for one second.
    wait(1000);
}

int wmain()
{
    // Create a call object that prints a single character to the console.
    call<wchar_t> report_progress([](wchar_t c) {
        wcout << c;
    });

    // Create a timer object that sends the dot character to the
    // call object every 100 milliseconds.
    timer<wchar_t> progress_timer(100, L'.', &report_progress, true);

    wcout << L"Performing a lengthy operation";

    // Start the timer on a separate context.
    progress_timer.start();

    // Perform a lengthy operation on the main context.
    perform_lengthy_operation();

    // Stop the timer and print a message.
    progress_timer.stop();

    wcout << L"done.";
}
```

This example produces the following sample output:

```
Performing a lengthy operation.....done.
```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `report-progress.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc report-progress.cpp
```

## See also

[Asynchronous Agents Library](#)

[Asynchronous Message Blocks](#)

[Message Passing Functions](#)

# How to: Use a Message Block Filter

3/4/2019 • 6 minutes to read • [Edit Online](#)

This document demonstrates how to use a filter function to enable an asynchronous message block to accept or reject a message on the basis of the payload of that message.

When you create a message block object such as a `concurrency::unbounded_buffer`, a `concurrency::call`, or a `concurrency::transformer`, you can supply a *filter function* that determines whether the message block accepts or rejects a message. A filter function is a useful way to guarantee that a message block receives only certain values.

Filter functions are important because they enable you to connect message blocks to form *dataflow networks*. In a dataflow network, message blocks control the flow of data by processing only those messages that meet specific criteria. Compare this to the control-flow model, where the flow of data is regulated by using control structures such as conditional statements, loops, and so on.

This document provides a basic example of how to use a message filter. For additional examples that use message filters and the dataflow model to connect message blocks, see [Walkthrough: Creating a Dataflow Agent](#) and [Walkthrough: Creating an Image-Processing Network](#).

## Example

Consider the following function, `count_primes`, which illustrates the basic usage of a message block that does not filter incoming messages. The message block appends prime numbers to a `std::vector` object. The `count_primes` function sends several numbers to the message block, receives the output values from the message block, and prints those numbers that are prime to the console.

```

// Illustrates usage of a message buffer that does not use filtering.
void count_primes(unsigned long random_seed)
{
    // Holds prime numbers.
    vector<unsigned long> primes;

    // Adds numbers that are prime to the vector object.
    transformer<unsigned long, unsigned long> t([&primes](unsigned long n) -> unsigned long
    {
        if (is_prime(n))
        {
            primes.push_back(n);
        }
        return n;
    });

    // Send random values to the message buffer.
    mt19937 generator(random_seed);
    for (int i = 0; i < 20; ++i)
    {
        send(t, static_cast<unsigned long>(generator()%10000));
    }

    // Receive from the message buffer the same number of times
    // to ensure that the message buffer has processed each message.
    for (int i = 0; i < 20; ++i)
    {
        receive(t);
    }

    // Print the prime numbers to the console.
    wcout << L"The following numbers are prime: " << endl;
    for(unsigned long prime : primes)
    {
        wcout << prime << endl;
    }
}

```

The `transformer` object processes all input values; however, it requires only those values that are prime. Although the application could be written so that the message sender sends only prime numbers, the requirements of the message receiver cannot always be known.

## Example

The following function, `count_primes_filter`, performs the same task as the `count_primes` function. However, the `transformer` object in this version uses a filter function to accept only those values that are prime. The function that performs the action only receives prime numbers; therefore, it does not have to call the `is_prime` function.

Because the `transformer` object receives only prime numbers, the `transformer` object itself can hold the prime numbers. In other words, the `transformer` object in this example is not required to add the prime numbers to the `vector` object.

```

// Illustrates usage of a message buffer that uses filtering.
void count_primes_filter(unsigned long random_seed)
{
    // Accepts numbers that are prime.
    transformer<unsigned long, unsigned long> t([](unsigned long n) -> unsigned long
    {
        // The filter function guarantees that the input value is prime.
        // Return the input value.
        return n;
    },
    nullptr,
    [](unsigned long n) -> bool
    {
        // Filter only values that are prime.
        return is_prime(n);
    });

    // Send random values to the message buffer.
    mt19937 generator(random_seed);
    size_t prime_count = 0;
    for (int i = 0; i < 20; ++i)
    {
        if (send(t, static_cast<unsigned long>(generator()%10000)))
        {
            ++prime_count;
        }
    }

    // Print the prime numbers to the console.
    wcout << L"The following numbers are prime: " << endl;
    while (prime_count-- > 0)
    {
        wcout << receive(t) << endl;
    }
}

```

The `transformer` object now processes only those values that are prime. In the previous example, `transformer` object processes all messages. Therefore, the previous example must receive the same number of messages that it sends. This example uses the result of the `concurrency::send` function to determine how many messages to receive from the `transformer` object. The `send` function returns **true** when the message buffer accepts the message and **false** when the message buffer rejects the message. Therefore, the number of times that the message buffer accepts the message matches the count of prime numbers.

## Example

The following code shows the complete example. The example calls both the `count_primes` function and the `count_primes_filter` function.

```

// primes-filter.cpp
// compile with: /EHsc
#include <agents.h>
#include <algorithm>
#include <iostream>
#include <random>

using namespace concurrency;
using namespace std;

// Determines whether the input value is prime.
bool is_prime(unsigned long n)
{
    if (n < 2)
        return false;
}

```

```

for (unsigned long i = 2; i < n; ++i)
{
    if ((n % i) == 0)
        return false;
}
return true;
}

// Illustrates usage of a message buffer that does not use filtering.
void count_primes(unsigned long random_seed)
{
    // Holds prime numbers.
    vector<unsigned long> primes;

    // Adds numbers that are prime to the vector object.
    transformer<unsigned long, unsigned long> t([&primes](unsigned long n) -> unsigned long
    {
        if (is_prime(n))
        {
            primes.push_back(n);
        }
        return n;
    });

    // Send random values to the message buffer.
    mt19937 generator(random_seed);
    for (int i = 0; i < 20; ++i)
    {
        send(t, static_cast<unsigned long>(generator()%10000));
    }

    // Receive from the message buffer the same number of times
    // to ensure that the message buffer has processed each message.
    for (int i = 0; i < 20; ++i)
    {
        receive(t);
    }

    // Print the prime numbers to the console.
    wcout << L"The following numbers are prime: " << endl;
    for(unsigned long prime : primes)
    {
        wcout << prime << endl;
    }
}

// Illustrates usage of a message buffer that uses filtering.
void count_primes_filter(unsigned long random_seed)
{
    // Accepts numbers that are prime.
    transformer<unsigned long, unsigned long> t([](unsigned long n) -> unsigned long
    {
        // The filter function guarantees that the input value is prime.
        // Return the input value.
        return n;
    },
    nullptr,
    [](unsigned long n) -> bool
    {
        // Filter only values that are prime.
        return is_prime(n);
    });

    // Send random values to the message buffer.
    mt19937 generator(random_seed);
    size_t prime_count = 0;
    for (int i = 0; i < 20; ++i)
    {
        if (send(t, static cast<unsigned long>(generator()%10000)))

```

```

    {
        ++prime_count;
    }
}

// Print the prime numbers to the console.
wcout << L"The following numbers are prime: " << endl;
while (prime_count-- > 0)
{
    wcout << receive(t) << endl;
}
}

int wmain()
{
    const unsigned long random_seed = 99714;

    wcout << L"Without filtering:" << endl;
    count_primes(random_seed);

    wcout << L"With filtering:" << endl;
    count_primes_filter(random_seed);

    /* Output:
    9973
    9349
    9241
    8893
    1297
    7127
    8647
    3229
    With filtering:
    The following numbers are prime:
    9973
    9349
    9241
    8893
    1297
    7127
    8647
    3229
    */
}

```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `primes-filter.cpp` and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc primes-filter.cpp**

## Robust Programming

A filter function can be a lambda function, a function pointer, or a function object. Every filter function takes one of the following forms:

```

bool (T)
bool (T const &)

```

To eliminate the unnecessary copying of data, use the second form when you have an aggregate type that is transmitted by value.

## See also

[Asynchronous Agents Library](#)

[Walkthrough: Creating a Dataflow Agent](#)

[Walkthrough: Creating an Image-Processing Network transformer Class](#)

# Synchronization Data Structures

3/4/2019 • 4 minutes to read • [Edit Online](#)

The Concurrency Runtime provides several data structures that let you synchronize access to shared data from multiple threads. These data structures are useful when you have shared data that you modify infrequently. A synchronization object, for example, a critical section, causes other threads to wait until the shared resource is available. Therefore, if you use such an object to synchronize access to data that is used frequently, you can lose scalability in your application. The [Parallel Patterns Library \(PPL\)](#) provides the `concurrency::combinable` class, which enables you to share a resource among several threads or tasks without the need for synchronization. For more information about the `combinable` class, see [Parallel Containers and Objects](#).

## Sections

This topic describes the following asynchronous message block types in detail:

- [critical\\_section](#)
- [reader\\_writer\\_lock](#)
- [scoped\\_lock](#) and [scoped\\_lock\\_read](#)
- [event](#)

## critical\_section

The `concurrency::critical_section` class represents a cooperative mutual exclusion object that yields to other tasks instead of preempting them. Critical sections are useful when multiple threads require exclusive read and write access to shared data.

The `critical_section` class is non-reentrant. The `concurrency::critical_section::lock` method throws an exception of type `concurrency::improper_lock` if it is called by the thread that already owns the lock.

### Methods and Features

The following table shows the important methods that are defined by the `critical_section` class.

METHOD	DESCRIPTION
<a href="#">lock</a>	Acquires the critical section. The calling context blocks until it acquires the lock.
<a href="#">try_lock</a>	Tries to acquire the critical section, but does not block.
<a href="#">unlock</a>	Releases the critical section.

[\[Top\]](#)

## reader\_writer\_lock

The `concurrency::reader_writer_lock` class provides thread-safe read/write operations to shared data. Use reader/writer locks when multiple threads require concurrent read access to a shared resource but rarely write to that shared resource. This class gives only one thread write access to an object at any time.

The `reader_writer_lock` class can perform better than the `critical_section` class because a `critical_section` object acquires exclusive access to a shared resource, which prevents concurrent read access.

Like the `critical_section` class, the `reader_writer_lock` class represents a cooperative mutual exclusion object that yields to other tasks instead of preempting them.

When a thread that must write to a shared resource acquires a reader/writer lock, other threads that also must access the resource are blocked until the writer releases the lock. The `reader_writer_lock` class is an example of a *write-preference* lock, which is a lock that unblocks waiting writers before it unblocks waiting readers.

Like the `critical_section` class, the `reader_writer_lock` class is non-reentrant. The `concurrency::reader_writer_lock::lock` and `concurrency::reader_writer_lock::lock_read` methods throw an exception of type `improper_lock` if they are called by a thread that already owns the lock.

#### NOTE

Because the `reader_writer_lock` class is non-reentrant, you cannot upgrade a read-only lock to a reader/writer lock or downgrade a reader/writer lock to a read-only lock. Performing either of these operations produces unspecified behavior.

### Methods and Features

The following table shows the important methods that are defined by the `reader_writer_lock` class.

METHOD	DESCRIPTION
<code>lock</code>	Acquires read/write access to the lock.
<code>try_lock</code>	Tries to acquire read/write access to the lock, but does not block.
<code>lock_read</code>	Acquires read-only access to the lock.
<code>try_lock_read</code>	Tries to acquire read-only access to the lock, but does not block.
<code>unlock</code>	Releases the lock.

[\[Top\]](#)

## scoped\_lock and scoped\_lock\_read

The `critical_section` and `reader_writer_lock` classes provide nested helper classes that simplify the way you work with mutual exclusion objects. These helper classes are known as *scoped locks*.

The `critical_section` class contains the `concurrency::critical_section::scoped_lock` class. The constructor acquires access to the provided `critical_section` object; the destructor releases access to that object. The `reader_writer_lock` class contains the `concurrency::reader_writer_lock::scoped_lock` class, which resembles `critical_section::scoped_lock`, except that it manages write access to the provided `reader_writer_lock` object. The `reader_writer_lock` class also contains the `concurrency::reader_writer_lock::scoped_lock_read` class. This class manages read access to the provided `reader_writer_lock` object.

Scoped locks provide several benefits when you are working with `critical_section` and `reader_writer_lock` objects manually. Typically, you allocate a scoped lock on the stack. A scoped lock releases access to its mutual exclusion object automatically when it is destroyed; therefore, you do not manually unlock the underlying object. This is useful when a function contains multiple `return` statements. Scoped locks can also help you write

exception-safe code. When a `throw` statement causes the stack to unwind, the destructor for any active scoped lock is called, and therefore the mutual exclusion object is always correctly released.

#### NOTE

When you use the `critical_section::scoped_lock`, `reader_writer_lock::scoped_lock`, and `reader_writer_lock::scoped_lock_read` classes, do not manually release access to the underlying mutual exclusion object. This can put the runtime in an invalid state.

## event

The `concurrency::event` class represents a synchronization object whose state can be signaled or non-signaled. Unlike synchronization objects, such as critical sections, whose purpose is to protect access to shared data, events synchronize flow of execution.

The `event` class is useful when one task has completed work for another task. For example, one task might signal another task that it has read data from a network connection or from a file.

### Methods and Features

The following table shows several of the important methods that are defined by the `event` class.

METHOD	DESCRIPTION
<a href="#">wait</a>	Waits for the event to become signaled.
<a href="#">set</a>	Sets the event to the signaled state.
<a href="#">reset</a>	Sets the event to the non-signaled state.
<a href="#">wait_for_multiple</a>	Waits for multiple events to become signaled.

### Example

For an example that shows how to use the `event` class, see [Comparing Synchronization Data Structures to the Windows API](#).

[\[Top\]](#)

## Related Sections

[Comparing Synchronization Data Structures to the Windows API](#)

Compares the behavior of the synchronization data structures to those provided by the Windows API.

[Concurrency Runtime](#)

Describes the Concurrency Runtime, which simplifies parallel programming, and contains links to related topics.

# Comparing Synchronization Data Structures to the Windows API

3/4/2019 • 4 minutes to read • [Edit Online](#)

This topic compares the behavior of the synchronization data structures that are provided by the Concurrency Runtime to those provided by the Windows API.

The synchronization data structures that are provided by the Concurrency Runtime follow the *cooperative threading model*. In the cooperative threading model, synchronization primitives explicitly yield their processing resources to other threads. This differs from the *preemptive threading model*, where processing resources are transferred to other threads by the controlling scheduler or operating system.

## critical\_section

The `concurrency::critical_section` class resembles the Windows `CRITICAL_SECTION` structure because it can be used only by the threads of one process. For more information about critical sections in the Windows API, see [Critical Section Objects](#).

## reader\_writer\_lock

The `concurrency::reader_writer_lock` class resembles Windows slim reader/writer (SRW) locks. The following table explains the similarities and differences.

FEATURE	<code>READER_WRITER_LOCK</code>	SRW LOCK
Non-reentrant	Yes	Yes
Can promote a reader to a writer (upgrade support)	No	No
Can demote a writer to a reader (downgrade support)	No	No
Write-preference lock	Yes	No
FIFO access to writers	Yes	No

For more information about SRW locks, see [Slim Reader/Writer \(SRW\) Locks](#) in the Platform SDK.

## event

The `concurrency::event` class resembles an unnamed, Windows manual-reset event. However, an `event` object behaves cooperatively, whereas a Windows event behaves preemptively. For more information about Windows events, see [Event Objects](#).

## Example

### Description

To better understand the difference between the `event` class and Windows events, consider the following

example. This example enables the scheduler to create at most two simultaneous tasks and then calls two similar functions that use the `event` class and a Windows manual-reset event. Each function first creates several tasks that wait for a shared event to become signaled. Each function then yields to the running tasks and then signals the event. Each function then waits for the signaled event.

## Code

```
// event-comparison.cpp
// compile with: /EHsc
#include <windows.h>
#include <concrtrm.h>
#include <ppl.h>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

// Demonstrates the usage of cooperative events.
void RunCooperativeEvents()
{
    // An event object.
    event e;

    // Create a task group and execute five tasks that wait for
    // the event to be set.
    task_group tasks;
    for (int i = 0; i < 5; ++i)
    {
        tasks.run([&] {
            // Print a message before waiting on the event.
            wstringstream ss;
            ss << L"\t\tContext " << GetExecutionContextId()
                << L": waiting on an event." << endl;
            wcout << ss.str();

            // Wait for the event to be set.
            e.wait();

            // Print a message after the event is set.
            ss = wstringstream();
            ss << L"\t\tContext " << GetExecutionContextId()
                << L": received the event." << endl;
            wcout << ss.str();
        });
    }

    // Wait a sufficient amount of time for all tasks to enter
    // the waiting state.
    Sleep(1000L);

    // Set the event.

    wstringstream ss;
    ss << L"\tSetting the event." << endl;
    wcout << ss.str();

    e.set();

    // Wait for all tasks to complete.
    tasks.wait();
}

// Demonstrates the usage of preemptive events.
void RunWindowsEvents()
{
    // A Windows event object.
```

```

HANDLE hEvent = CreateEvent(NULL, TRUE, FALSE, TEXT("Windows Event"));

// Create a task group and execute five tasks that wait for
// the event to be set.
task_group tasks;
for (int i = 0; i < 5; ++i)
{
    tasks.run([&] {
        // Print a message before waiting on the event.
        wstringstream ss;
        ss << L"\t\tContext " << GetExecutionContextId()
            << L": waiting on an event." << endl;
        wcout << ss.str();

        // Wait for the event to be set.
        WaitForSingleObject(hEvent, INFINITE);

        // Print a message after the event is set.
        ss = wstringstream();
        ss << L"\t\tContext " << GetExecutionContextId()
            << L": received the event." << endl;
        wcout << ss.str();
    });
}

// Wait a sufficient amount of time for all tasks to enter
// the waiting state.
Sleep(1000L);

// Set the event.

wstringstream ss;
ss << L"\tSetting the event." << endl;
wcout << ss.str();

SetEvent(hEvent);

// Wait for all tasks to complete.
tasks.wait();

// Close the event handle.
CloseHandle(hEvent);
}

int wmain()
{
    // Create a scheduler policy that allows up to two
    // simultaneous tasks.
    SchedulerPolicy policy(1, MaxConcurrency, 2);

    // Attach the policy to the current scheduler.
    CurrentScheduler::Create(policy);

    wcout << L"Cooperative event:" << endl;
    RunCooperativeEvents();

    wcout << L"Windows event:" << endl;
    RunWindowsEvents();
}

```

## Comments

This example produces the following sample output:

Cooperative event:

```
Context 0: waiting on an event.  
Context 1: waiting on an event.  
Context 2: waiting on an event.  
Context 3: waiting on an event.  
Context 4: waiting on an event.  
Setting the event.  
Context 5: received the event.  
Context 6: received the event.  
Context 7: received the event.  
Context 8: received the event.  
Context 9: received the event.
```

Windows event:

```
Context 10: waiting on an event.  
Context 11: waiting on an event.  
Setting the event.  
Context 12: received the event.  
Context 14: waiting on an event.  
Context 15: received the event.  
Context 16: waiting on an event.  
Context 17: received the event.  
Context 18: waiting on an event.  
Context 19: received the event.  
Context 13: received the event.
```

Because the `event` class behaves cooperatively, the scheduler can reallocate processing resources to another context when an event is waiting to enter the signaled state. Therefore, more work is accomplished by the version that uses the `event` class. In the version that uses Windows events, each waiting task must enter the signaled state before the next task is started.

For more information about tasks, see [Task Parallelism](#).

## See also

[Synchronization Data Structures](#)

# Task Scheduler (Concurrency Runtime)

3/4/2019 • 3 minutes to read • [Edit Online](#)

The topics in this part of the documentation describe the important features of the Concurrency Runtime Task Scheduler. The Task Scheduler is useful when you want fine-tune the performance of your existing code that uses the Concurrency Runtime.

## IMPORTANT

The Task Scheduler is not available from a Universal Windows Platform (UWP) app. For more information, see [Creating Asynchronous Operations in C++ for UWP Apps](#).

In Visual Studio 2015 and later, the `concurrency::task` class and related types in `ppltasks.h` use the Windows ThreadPool as their scheduler. This topic no longer applies to types that are defined in `ppltasks.h`. Parallel algorithms such as `parallel_for` continue to use the Concurrency Runtime as the default scheduler.

## TIP

The Concurrency Runtime provides a default scheduler, and therefore you are not required to create one in your application. Because the Task Scheduler helps you fine-tune the performance of your applications, we recommend that you start with the [Parallel Patterns Library \(PPL\)](#) or the [Asynchronous Agents Library](#) if you are new to the Concurrency Runtime.

The Task Scheduler schedules and coordinates tasks at run time. A *task* is a unit of work that performs a specific job. A task can typically run in parallel with other tasks. The work that is performed by task group items, parallel algorithms, and asynchronous agents are all examples of tasks.

The Task Scheduler manages the details that are related to efficiently scheduling tasks on computers that have multiple computing resources. The Task Scheduler also uses the newest features of the underlying operating system. Therefore, applications that use the Concurrency Runtime automatically scale and improve on hardware that has expanded capabilities.

[Comparing to Other Concurrency Models](#) describes the differences between preemptive and cooperative scheduling mechanisms. The Task Scheduler uses cooperative scheduling and a work-stealing algorithm together with the preemptive scheduler of the operating system to achieve maximum usage of processing resources.

The Concurrency Runtime provides a default scheduler so that you do not have to manage infrastructure details. Therefore, you typically do not use the Task Scheduler directly. However, to meet the quality needs of your application, you can use the Task Scheduler to provide your own scheduling policy or associate schedulers with specific tasks. For example, suppose you have a parallel sorting routine that does not scale beyond four processors. You can use *scheduler policies* to create a scheduler that generates no more than four concurrent tasks. Running the sorting routine on this scheduler enables other active schedulers to use any remaining processing resources.

## Related Topics

TITLE	DESCRIPTION
<a href="#">Scheduler Instances</a>	<p>Describes scheduler instances and how to use the <code>concurrency::Scheduler</code> and <code>concurrency::CurrentScheduler</code> classes to manage them. Use scheduler instances when you want to associate explicit scheduling policies with specific types of workloads.</p>
<a href="#">Scheduler Policies</a>	<p>Describes the role of scheduler policies. Use scheduler policies when you want to control the strategy that the scheduler uses when it manages tasks.</p>
<a href="#">Schedule Groups</a>	<p>Describes the role of schedule groups. Use schedule groups when you require a high degree of locality among tasks, for example, when a group of related tasks benefit from executing on the same processor node.</p>
<a href="#">Lightweight Tasks</a>	<p>Describes the role of lightweight tasks. Lightweight tasks are useful when you adapt existing code to use the scheduling functionality of the Concurrency Runtime.</p>
<a href="#">Contexts</a>	<p>Describes the role of contexts, the <code>concurrency::wait</code> function, and the <code>concurrency::Context</code> class. Use this functionality when you need control over when contexts block, unblock, and yield, or when you want to enable oversubscription in your application.</p>
<a href="#">Memory Management Functions</a>	<p>Describes the <code>concurrency::Alloc</code> and <code>concurrency::Free</code> functions. These functions can improve memory performance by allocating and freeing memory in a concurrent manner.</p>
<a href="#">Comparing to Other Concurrency Models</a>	<p>Describes the differences between preemptive and cooperative scheduling mechanisms.</p>
<a href="#">Parallel Patterns Library (PPL)</a>	<p>Describes how to use various parallel patterns, for example, parallel algorithms, in your applications.</p>
<a href="#">Asynchronous Agents Library</a>	<p>Describes how to use asynchronous agents in your applications.</p>
<a href="#">Concurrency Runtime</a>	<p>Describes the Concurrency Runtime, which simplifies parallel programming, and contains links to related topics.</p>

# Scheduler Instances

3/4/2019 • 6 minutes to read • [Edit Online](#)

This document describes the role of scheduler instances in the Concurrency Runtime and how to use the [concurrency::Scheduler](#) and [concurrency::CurrentScheduler](#) classes to create and manage scheduler instances. Scheduler instances are useful when you want to associate explicit scheduling policies with specific types of workloads. For example, you can create one scheduler instance to run some tasks at an elevated thread priority and use the default scheduler to run other tasks at the normal thread priority.

## TIP

The Concurrency Runtime provides a default scheduler, and therefore you are not required to create one in your application. Because the Task Scheduler helps you fine-tune the performance of your applications, we recommend that you start with the [Parallel Patterns Library \(PPL\)](#) or the [Asynchronous Agents Library](#) if you are new to the Concurrency Runtime.

## Sections

- [The Scheduler and CurrentScheduler Classes](#)
- [Creating a Scheduler Instance](#)
- [Managing the Lifetime of a Scheduler Instance](#)
- [Methods and Features](#)
- [Example](#)

## The Scheduler and CurrentScheduler Classes

The Task Scheduler enables applications to use one or more *scheduler instances* to schedule work. The [concurrency::Scheduler](#) class represents a scheduler instance and encapsulates the functionality that is related to scheduling tasks.

A thread that is attached to a scheduler is known as an *execution context*, or just *context*. One scheduler can be active on the current context at any time. The active scheduler is also known as the *current scheduler*. The Concurrency Runtime uses the [concurrency::CurrentScheduler](#) class to provide access to the current scheduler. The current scheduler for one context can differ from the current scheduler for another context. The runtime does not provide a process-level representation of the current scheduler.

Typically, the `CurrentScheduler` class is used to access the current scheduler. The `Scheduler` class is useful when you need to manage a scheduler that is not the current one.

The following sections describe how to create and manage a scheduler instance. For a complete example that illustrates these tasks, see [How to: Manage a Scheduler Instance](#).

[\[Top\]](#)

## Creating a Scheduler Instance

There are three ways to create a `Scheduler` object:

- If no scheduler exists, the runtime creates a default scheduler for you when you use runtime functionality, for example, a parallel algorithm, to perform work. The default scheduler becomes the current scheduler for

the context that initiates the parallel work.

- The `concurrency::CurrentScheduler::Create` method creates a `Scheduler` object that uses a specific policy and associates that scheduler with the current context.
- The `concurrency::Scheduler::Create` method creates a `Scheduler` object that uses a specific policy, but does not associate it with the current context.

Allowing the runtime to create a default scheduler enables all concurrent tasks to share the same scheduler. Typically, the functionality that is provided by the [Parallel Patterns Library](#) (PPL) or the [Asynchronous Agents Library](#) is used to perform parallel work. Therefore, you do not have to work directly with the scheduler to control its policy or lifetime. When you use the PPL or the Agents Library, the runtime creates the default scheduler if it does not exist and makes it the current scheduler for each context. When you create a scheduler and set it as the current scheduler, then the runtime uses that scheduler to schedule tasks. Create additional scheduler instances only when you require a specific scheduling policy. For more information about the policies that are associated with a scheduler, see [Scheduler Policies](#).

[\[Top\]](#)

## Managing the Lifetime of a Scheduler Instance

The runtime uses a reference-counting mechanism to control the lifetime of `Scheduler` objects.

When you use the `CurrentScheduler::Create` method or the `Scheduler::Create` method to create a `Scheduler` object, the runtime sets the initial reference count of that scheduler to one. The runtime increments the reference count when you call the `concurrency::Scheduler::Attach` method. The `Scheduler::Attach` method associates the `Scheduler` object together with the current context. This makes it the current scheduler. When you call the `CurrentScheduler::Create` method, the runtime both creates a `Scheduler` object and attaches it to the current context (and sets the reference count to one). You can also use the `concurrency::Scheduler::Reference` method to increment the reference count of a `Scheduler` object.

The runtime decrements the reference count when you call the `concurrency::CurrentScheduler::Detach` method to detach the current scheduler, or call the `concurrency::Scheduler::Release` method. When the reference count reaches zero, the runtime destroys the `Scheduler` object after all scheduled tasks finish. A running task is allowed to increment the reference count of the current scheduler. Therefore, if the reference count reaches zero and a task increments the reference count, the runtime does not destroy the `Scheduler` object until the reference count again reaches zero and all tasks finish.

The runtime maintains an internal stack of `Scheduler` objects for each context. When you call the `Scheduler::Attach` or `CurrentScheduler::Create` method, the runtime pushes that `Scheduler` object onto the stack for the current context. This makes it the current scheduler. When you call `CurrentScheduler::Detach`, the runtime pops the current scheduler from the stack for current context and sets the previous one as the current scheduler.

The runtime provides several ways to manage the lifetime of a scheduler instance. The following table shows the appropriate method that releases or detaches the scheduler from the current context for each method that creates or attaches a scheduler to the current context.

CREATE OR ATTACH METHOD	RELEASE OR DETACH METHOD
<code>CurrentScheduler::Create</code>	<code>CurrentScheduler::Detach</code>
<code>Scheduler::Create</code>	<code>Scheduler::Release</code>
<code>Scheduler::Attach</code>	<code>CurrentScheduler::Detach</code>

CREATE OR ATTACH METHOD	RELEASE OR DETACH METHOD
<code>Scheduler::Reference</code>	<code>Scheduler::Release</code>

Calling the inappropriate release or detach method produces unspecified behavior in the runtime.

When you use functionality, for example, the PPL, that causes the runtime to create the default scheduler for you, do not release or detach this scheduler. The runtime manages the lifetime of any scheduler that it creates.

Because the runtime does not destroy a `Scheduler` object before all tasks have finished, you can use the [concurrency::Scheduler::RegisterShutdownEvent](#) method or the [concurrency::CurrentScheduler::RegisterShutdownEvent](#) method to receive a notification when a `Scheduler` object is destroyed. This is useful when you must wait for every task that is scheduled by a `Scheduler` object to finish.

[\[Top\]](#)

## Methods and Features

This section summarizes the important methods of the `CurrentScheduler` and `Scheduler` classes.

Think of the `CurrentScheduler` class as a helper for creating a scheduler for use on the current context. The `Scheduler` class lets you control a scheduler that belongs to another context.

The following table shows the important methods that are defined by the `CurrentScheduler` class.

METHOD	DESCRIPTION
<a href="#">Create</a>	Creates a <code>Scheduler</code> object that uses the specified policy and associates it with the current context.
<a href="#">Get</a>	Retrieves a pointer to the <code>Scheduler</code> object that is associated with the current context. This method does not increment the reference count of the <code>Scheduler</code> object.
<a href="#">Detach</a>	Detaches the current scheduler from the current context and sets the previous one as the current scheduler.
<a href="#">RegisterShutdownEvent</a>	Registers an event that the runtime sets when the current scheduler is destroyed.
<a href="#">CreateScheduleGroup</a>	Creates a <a href="#">concurrency::ScheduleGroup</a> object in the current scheduler.
<a href="#">ScheduleTask</a>	Adds a lightweight task to the scheduling queue of the current scheduler.
<a href="#">GetPolicy</a>	Retrieves a copy of the policy that is associated with the current scheduler.

The following table shows the important methods that are defined by the `Scheduler` class.

METHOD	DESCRIPTION
<a href="#">Create</a>	Creates a <code>Scheduler</code> object that uses the specified policy.

METHOD	DESCRIPTION
<a href="#">Attach</a>	Associates the <code>Scheduler</code> object together with the current context.
<a href="#">Reference</a>	Increments the reference counter of the <code>Scheduler</code> object.
<a href="#">Release</a>	Decrements the reference counter of the <code>Scheduler</code> object.
<a href="#">RegisterShutdownEvent</a>	Registers an event that the runtime sets when the <code>Scheduler</code> object is destroyed.
<a href="#">CreateScheduleGroup</a>	Creates a <code>concurrency::ScheduleGroup</code> object in the <code>Scheduler</code> object.
<a href="#">ScheduleTask</a>	Schedules a lightweight task from the <code>Scheduler</code> object.
<a href="#">GetPolicy</a>	Retrieves a copy of the policy that is associated with the <code>Scheduler</code> object.
<a href="#">SetDefaultSchedulerPolicy</a>	Sets the policy for the runtime to use when it creates the default scheduler.
<a href="#">ResetDefaultSchedulerPolicy</a>	Restores the default policy to the one that was active before the call to <code>SetDefaultSchedulerPolicy</code> . If the default scheduler is created after this call, the runtime uses default policy settings to create the scheduler.

[\[Top\]](#)

## Example

For basic examples of how to create and manage a scheduler instance, see [How to: Manage a Scheduler Instance](#).

## See also

- [Task Scheduler](#)
- [How to: Manage a Scheduler Instance](#)
- [Scheduler Policies](#)
- [Schedule Groups](#)

# How to: Manage a Scheduler Instance

3/4/2019 • 4 minutes to read • [Edit Online](#)

Scheduler instances let you associate specific scheduling policies with various kinds of workloads. This topic contains two basic examples that show how to create and manage a scheduler instance.

The examples create schedulers that use the default scheduler policies. For an example that creates a scheduler that uses a custom policy, see [How to: Specify Specific Scheduler Policies](#).

## To manage a scheduler instance in your application

1. Create a `concurrency::SchedulerPolicy` object that contains the policy values for the scheduler to use.
2. Call the `concurrency::CurrentScheduler::Create` method or the `concurrency::Scheduler::Create` method to create a scheduler instance.

If you use the `Scheduler::Create` method, call the `concurrency::Scheduler::Attach` method when you need to associate the scheduler with the current context.

3. Call the `CreateEvent` function to create a handle to a non-signaled, auto-reset event object.
4. Pass the handle to the event object that you just created to the `concurrency::CurrentScheduler::RegisterShutdownEvent` method or the `concurrency::Scheduler::RegisterShutdownEvent` method. This registers the event to be set when the scheduler is destroyed.
5. Perform the tasks that you want the current scheduler to schedule.
6. Call the `concurrency::CurrentScheduler::Detach` method to detach the current scheduler and restore the previous scheduler as the current one.

If you use the `Scheduler::Create` method, call the `concurrency::Scheduler::Release` method to decrement the reference count of the `Scheduler` object.

7. Pass the handle to the event to the `WaitForSingleObject` function to wait for the scheduler to shut down.
8. Call the `CloseHandle` function to close the handle to the event object.

## Example

The following code shows two ways to manage a scheduler instance. Each example first uses the default scheduler to perform a task that prints out the unique identifier of the current scheduler. Each example then uses a scheduler instance to perform the same task again. Finally, each example restores the default scheduler as the current one and performs the task one more time.

The first example uses the `concurrency::CurrentScheduler` class to create a scheduler instance and associate it with the current context. The second example uses the `concurrency::Scheduler` class to perform the same task. Typically, the `CurrentScheduler` class is used to work with the current scheduler. The second example, which uses the `Scheduler` class, is useful when you want to control when the scheduler is associated with the current context or when you want to associate specific schedulers with specific tasks.

```
// scheduler-instance.cpp
// compile with: /EHsc
#include <windows.h>
#include <pp1.h>
#include <iostream>
```

```

using namespace concurrency;
using namespace std;

// Prints the identifier of the current scheduler to the console.
void perform_task()
{
    // A task group.
    task_group tasks;

    // Run a task in the group. The current scheduler schedules the task.
    tasks.run_and_wait([] {
        wcout << L"Current scheduler id: " << CurrentScheduler::Id() << endl;
    });
}

// Uses the CurrentScheduler class to manage a scheduler instance.
void current_scheduler()
{
    // Run the task.
    // This prints the identifier of the default scheduler.
    perform_task();

    // For demonstration, create a scheduler object that uses
    // the default policy values.
    wcout << L"Creating and attaching scheduler..." << endl;
    CurrentScheduler::Create(SchedulerPolicy());

    // Register to be notified when the scheduler shuts down.
    HANDLE hShutdownEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
    CurrentScheduler::RegisterShutdownEvent(hShutdownEvent);

    // Run the task again.
    // This prints the identifier of the new scheduler.
    perform_task();

    // Detach the current scheduler. This restores the previous scheduler
    // as the current one.
    wcout << L"Detaching scheduler..." << endl;
    CurrentScheduler::Detach();

    // Wait for the scheduler to shut down and destroy itself.
    WaitForSingleObject(hShutdownEvent, INFINITE);

    // Close the event handle.
    CloseHandle(hShutdownEvent);

    // Run the sample task again.
    // This prints the identifier of the default scheduler.
    perform_task();
}

// Uses the Scheduler class to manage a scheduler instance.
void explicit_scheduler()
{
    // Run the task.
    // This prints the identifier of the default scheduler.
    perform_task();

    // For demonstration, create a scheduler object that uses
    // the default policy values.
    wcout << L"Creating scheduler..." << endl;
    Scheduler* scheduler = Scheduler::Create(SchedulerPolicy());

    // Register to be notified when the scheduler shuts down.
    HANDLE hShutdownEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
    scheduler->RegisterShutdownEvent(hShutdownEvent);

    // Associate the scheduler with the current thread

```

```

// Associate the scheduler with the current thread.
wcout << L"Attaching scheduler..." << endl;
scheduler->Attach();

// Run the sample task again.
// This prints the identifier of the new scheduler.
perform_task();

// Detach the current scheduler. This restores the previous scheduler
// as the current one.
wcout << L"Detaching scheduler..." << endl;
CurrentScheduler::Detach();

// Release the final reference to the scheduler. This causes the scheduler
// to shut down after all tasks finish.
scheduler->Release();

// Wait for the scheduler to shut down and destroy itself.
WaitForSingleObject(hShutdownEvent, INFINITE);

// Close the event handle.
CloseHandle(hShutdownEvent);

// Run the sample task again.
// This prints the identifier of the default scheduler.
perform_task();
}

int wmain()
{
// Use the CurrentScheduler class to manage a scheduler instance.
wcout << L"Using CurrentScheduler class..." << endl << endl;
current_scheduler();

wcout << endl << endl;

// Use the Scheduler class to manage a scheduler instance.
wcout << L"Using Scheduler class..." << endl << endl;
explicit_scheduler();
}

```

This example produces the following output.

```

Using CurrentScheduler class...

Current scheduler id: 0
Creating and attaching scheduler...
Current scheduler id: 1
Detaching scheduler...
Current scheduler id: 0

Using Scheduler class...

Current scheduler id: 0
Creating scheduler...
Attaching scheduler...
Current scheduler id: 2
Detaching scheduler...
Current scheduler id: 0

```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `scheduler-instance.cpp` and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc scheduler-instance.cpp**

## See also

[Scheduler Instances](#)

[How to: Specify Specific Scheduler Policies](#)

# Scheduler Policies

3/4/2019 • 3 minutes to read • [Edit Online](#)

This document describes the role of scheduler policies in the Concurrency Runtime. A *scheduler policy* controls the strategy that the scheduler uses when it manages tasks. For example, consider an application that requires some tasks to execute at `THREAD_PRIORITY_NORMAL` and other tasks to execute at `THREAD_PRIORITY_HIGHEST`. You can create two scheduler instances: one that specifies the `ContextPriority` policy to be `THREAD_PRIORITY_NORMAL` and another that specifies the same policy to be `THREAD_PRIORITY_HIGHEST`.

By using scheduler policies, you can divide the available processing resources and assign a fixed set of resources to each scheduler. For example, consider a parallel algorithm that does not scale beyond four processors. You can create a scheduler policy that limits its tasks to use no more than four processors concurrently.

## TIP

The Concurrency Runtime provides a default scheduler. Therefore, you don't have to create one in your application. Because the Task Scheduler helps you fine-tune the performance of your applications, we recommend that you start with the [Parallel Patterns Library \(PPL\)](#) or the [Asynchronous Agents Library](#) if you are new to the Concurrency Runtime.

When you use the `concurrency::CurrentScheduler::Create`, `concurrency::Scheduler::Create`, or `concurrency::Scheduler::SetDefaultSchedulerPolicy` method to create a scheduler instance, you provide a `concurrency::SchedulerPolicy` object that contains a collection of key-value pairs that specify the behavior of the scheduler. The `SchedulerPolicy` constructor takes a variable number of arguments. The first argument is the number of policy elements that you are about to specify. The remaining arguments are key-value pairs for each policy element. The following example creates a `SchedulerPolicy` object that specifies three policy elements. The runtime uses default values for the policy keys that are not specified.

```
SchedulerPolicy policy(3,
    MinConcurrency, 2,
    MaxConcurrency, 4,
    ContextPriority, THREAD_PRIORITY_HIGHEST
);
```

The `concurrency::PolicyElementKey` enumeration defines the policy keys that are associated with the Task Scheduler. The following table describes the policy keys and the default value that the runtime uses for each of them.

POLICY KEY	DESCRIPTION	DEFAULT VALUE
<code>SchedulerKind</code>	A <code>concurrency::SchedulerType</code> value that specifies the type of threads to use to schedule tasks.	<code>ThreadScheduler</code> (use normal threads). This is the only valid value for this key.
<code>MaxConcurrency</code>	An <code>unsigned int</code> value that specifies the maximum number of concurrency resources that the scheduler uses.	<code>concurrency::MaxExecutionResources</code>
<code>MinConcurrency</code>	An <code>unsigned int</code> value that specifies the minimum number of concurrency resources that the scheduler uses.	<code>1</code>

POLICY KEY	DESCRIPTION	DEFAULT VALUE
<code>TargetOversubscriptionFactor</code>	An <code>unsigned int</code> value that specifies how many threads to allocate to each processing resource.	<code>1</code>
<code>LocalContextCacheSize</code>	An <code>unsigned int</code> value that specifies the maximum number of contexts that can be cached in the local queue of each virtual processor.	<code>8</code>
<code>ContextStackSize</code>	An <code>unsigned int</code> value that specifies the size of the stack, in kilobytes, to reserve for each context.	<code>0</code> (use the default stack size)
<code>ContextPriority</code>	An <code>int</code> value that specifies the thread priority of each context. This can be any value that you can pass to <a href="#">SetThreadPriority</a> or <code>INHERIT_THREAD_PRIORITY</code> .	<code>THREAD_PRIORITY_NORMAL</code>

`SchedulingProtocol` | A [concurrency::SchedulingProtocolType](#) value that specifies the scheduling algorithm to use. | `EnhanceScheduleGroupLocality` | `DynamicProgressFeedback` | A [concurrency::DynamicProgressFeedbackType](#) value that specifies whether to rebalance resources according to statistics-based progress information.

**Note** Do not set this policy to `ProgressFeedbackDisabled` because it is reserved for use by the runtime. | `ProgressFeedbackEnabled` |

Each scheduler uses its own policy when it schedules tasks. The policies that are associated with one scheduler do not affect the behavior of any other scheduler. In addition, you cannot change the scheduler policy after you create the `Scheduler` object.

#### IMPORTANT

Use only scheduler policies to control the attributes for threads that the runtime creates. Do not change the thread affinity or priority of threads that are created by the runtime because that might cause undefined behavior.

The runtime creates a default scheduler for you if you do not explicitly create one. If you want to use the default scheduler in your application, but you want to specify a policy for that scheduler to use, call the [concurrency::Scheduler::SetDefaultSchedulerPolicy](#) method before you schedule parallel work. If you do not call the `Scheduler::SetDefaultSchedulerPolicy` method, the runtime uses the default policy values from the table.

Use the [concurrency::CurrentScheduler::GetPolicy](#) and the [concurrency::Scheduler::GetPolicy](#) methods to retrieve a copy of the scheduler policy. The policy values that you receive from these methods can differ from the policy values that you specify when you create the scheduler.

## Example

To examine examples that use specific scheduler policies to control the behavior of the scheduler, see [How to: Specify Specific Scheduler Policies](#) and [How to: Create Agents that Use Specific Scheduler Policies](#).

## See also

[Task Scheduler](#)

[How to: Specify Specific Scheduler Policies](#)

## How to: Create Agents that Use Specific Scheduler Policies

# How to: Specify Specific Scheduler Policies

3/4/2019 • 4 minutes to read • [Edit Online](#)

Scheduler policies let you control the strategy that the scheduler uses when it manages tasks. This topic shows how to use a scheduler policy to increase the thread priority of a task that prints a progress indicator to the console.

For an example that uses custom scheduler policies together with asynchronous agents, see [How to: Create Agents that Use Specific Scheduler Policies](#).

## Example

The following example performs two tasks in parallel. The first task computes the  $n^{\text{th}}$  Fibonacci number. The second task prints a progress indicator to the console.

The first task uses recursive decomposition to compute the Fibonacci number. That is, each task recursively creates subtasks to compute the overall result. A task that uses recursive decomposition might use all available resources, and thereby starve other tasks. In this example, the task that prints the progress indicator might not receive timely access to computing resources.

To provide the task that prints a progress message fair access to computing resources, this example uses steps that are described in [How to: Manage a Scheduler Instance](#) to create a scheduler instance that has a custom policy. The custom policy specifies the thread priority to be the highest priority class.

This example uses the [concurrency::call](#) and [concurrency::timer](#) classes to print the progress indicator. These classes have versions of their constructors that take a reference to a [concurrency::Scheduler](#) object that schedules them. The example uses the default scheduler to schedule the task that computes the Fibonacci number and the scheduler instance to schedule the task that prints the progress indicator.

To illustrate the benefits of using a scheduler that has a custom policy, this example performs the overall task two times. The example first uses the default scheduler to schedule both tasks. The example then uses the default scheduler to schedule the first task, and a scheduler that has a custom policy to schedule the second task.

```
// scheduler-policy.cpp
// compile with: /EHsc
#include <windows.h>
#include <ppl.h>
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Computes the nth Fibonacci number.
// This function illustrates a lengthy operation and is therefore
// not optimized for performance.
int fibonacci(int n)
{
    if (n < 2)
        return n;

    // Compute the components in parallel.
    int n1, n2;
    parallel_invoke(
        [n,&n1] { n1 = fibonacci(n-1); },
        [n,&n2] { n2 = fibonacci(n-2); }
    );
}
```

```

};

return n1 + n2;
}

// Prints a progress indicator while computing the nth Fibonacci number.
void fibonacci_with_progress(Scheduler& progress_scheduler, int n)
{
    // Use a task group to compute the Fibonacci number.
    // The tasks in this group are scheduled by the current scheduler.
    structured_task_group tasks;

    auto task = make_task([n] {
        fibonacci(n);
    });
    tasks.run(task);

    // Create a call object that prints its input to the console.
    // This example uses the provided scheduler to schedule the
    // task that the call object performs.
    call<wchar_t> c(progress_scheduler, [](wchar_t c) {
        wcout << c;
    });

    // Connect the call object to a timer object. The timer object
    // sends a progress message to the call object every 100 ms.
    // This example also uses the provided scheduler to schedule the
    // task that the timer object performs.
    timer<wchar_t> t(progress_scheduler, 100, L'.'. ', &c, true);
    t.start();

    // Wait for the task that computes the Fibonacci number to finish.
    tasks.wait();

    // Stop the timer.
    t.stop();

    wcout << L"done" << endl;
}

int wmain()
{
    // Calculate the 38th Fibonacci number.
    const int n = 38;

    // Use the default scheduler to schedule the progress indicator while
    // the Fibonacci number is calculated in the background.

    wcout << L"Default scheduler:" << endl;
    fibonacci_with_progress(*CurrentScheduler::Get(), n);

    // Now use a scheduler that has a custom policy for the progress indicator.
    // The custom policy specifies the thread priority to the highest
    // priority class.

    SchedulerPolicy policy(1, ContextPriority, THREAD_PRIORITY_HIGHEST);
    Scheduler* scheduler = Scheduler::Create(policy);

    // Register to be notified when the scheduler shuts down.
    HANDLE hShutdownEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
    scheduler->RegisterShutdownEvent(hShutdownEvent);

    wcout << L"Scheduler that has a custom policy:" << endl;
    fibonacci_with_progress(*scheduler, n);

    // Release the final reference to the scheduler. This causes the scheduler
    // to shut down.
    scheduler->Release();
}

```

```
// Wait for the scheduler to shut down and destroy itself.
WaitForSingleObject(hShutdownEvent, INFINITE);

// Close the event handle.
CloseHandle(hShutdownEvent);
}
```

This example produces the following output.

```
Default scheduler:
.....done
Scheduler that has a custom policy:
.....done
```

Although both sets of tasks produce the same result, the version that uses a custom policy enables the task that prints the progress indicator to run at an elevated priority so that it behaves more responsively.

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `scheduler-policy.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc scheduler-policy.cpp
```

## See also

[Scheduler Policies](#)

[How to: Manage a Scheduler Instance](#)

[How to: Create Agents that Use Specific Scheduler Policies](#)

# How to: Create Agents that Use Specific Scheduler Policies

3/4/2019 • 5 minutes to read • [Edit Online](#)

An agent is an application component that works asynchronously with other components to solve larger computing tasks. An agent typically has a set life cycle and maintains state.

Every agent can have unique application requirements. For example, an agent that enables user interaction (either retrieving input or displaying output) might require higher priority access to computing resources. Scheduler policies let you control the strategy that the scheduler uses when it manages tasks. This topic demonstrates how to create agents that use specific scheduler policies.

For a basic example that uses custom scheduler policies together with asynchronous message blocks, see [How to: Specify Specific Scheduler Policies](#).

This topic uses functionality from the Asynchronous Agents Library, such as agents, message blocks, and message-passing functions, to perform work. For more information about the Asynchronous Agents Library, see [Asynchronous Agents Library](#).

## Example

The following example defines two classes that derive from `concurrency::agent`: `permutor` and `printer`. The `permutor` class computes all permutations of a given input string. The `printer` class prints progress messages to the console. The `permutor` class performs a computationally-intensive operation, which might use all available computing resources. To be useful, the `printer` class must print each progress message in a timely manner.

To provide the `printer` class fair access to computing resources, this example uses steps that are described in [How to: Manage a Scheduler Instance](#) to create a scheduler instance that has a custom policy. The custom policy specifies the thread priority to be the highest priority class.

To illustrate the benefits of using a scheduler that has a custom policy, this example performs the overall task two times. The example first uses the default scheduler to schedule both tasks. The example then uses the default scheduler to schedule the `permutor` object, and a scheduler that has a custom policy to schedule the `printer` object.

```
// permute-strings.cpp
// compile with: /EHsc
#include <windows.h>
#include <ppl.h>
#include <agents.h>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

// Computes all permutations of a given input string.
class permutor : public agent
{
public:
    explicit permutor(ISource<wstring>& source,
        ITarget<unsigned int>& progress)
        : _source(source)
        , _progress(progress)
    {
```

```

}
}

explicit permutor(ISource<wstring>& source,
    ITarget<unsigned int>& progress,
    Scheduler& scheduler)
    : agent(scheduler)
    , _source(source)
    , _progress(progress)
{
}

explicit permutor(ISource<wstring>& source,
    ITarget<unsigned int>& progress,
    ScheduleGroup& group)
    : agent(group)
    , _source(source)
    , _progress(progress)
{
}

protected:
    // Performs the work of the agent.
    void run()
    {
        // Read the source string from the buffer.
        wstring s = receive(_source);

        // Compute all permutations.
        permute(s);

        // Set the status of the agent to agent_done.
        done();
    }

    // Computes the factorial of the given value.
    unsigned int factorial(unsigned int n)
    {
        if (n == 0)
            return 0;
        if (n == 1)
            return 1;
        return n * factorial(n - 1);
    }

    // Computes the nth permutation of the given wstring.
    wstring permutation(int n, const wstring& s)
    {
        wstring t(s);

        size_t len = t.length();
        for (unsigned int i = 2; i < len; ++i)
        {
            swap(t[n % i], t[i]);
            n = n / i;
        }
        return t;
    }

    // Computes all permutations of the given string.
    void permute(const wstring& s)
    {
        // The factorial gives us the number of permutations.
        unsigned int permutation_count = factorial(s.length());

        // The number of computed permutations.
        LONG count = 0L;

        // Tracks the previous percentage so that we only send the percentage

```

```

// when it changes.
unsigned int previous_percent = 0u;

// Send initial progress message.
send(_progress, previous_percent);

// Compute all permutations in parallel.
parallel_for (0u, permutation_count, [&](unsigned int i) {
    // Compute the permutation.
    permutation(i, s);

    // Send the updated status to the progress reader.
    unsigned int percent = 100 * InterlockedIncrement(&count) / permutation_count;
    if (percent > previous_percent)
    {
        send(_progress, percent);
        previous_percent = percent;
    }
});

// Send final progress message.
send(_progress, 100u);
}

private:
    // The buffer that contains the source string to permute.
    ISource<wstring>& _source;

    // The buffer to write progress status to.
    ITarget<unsigned int>& _progress;
};

// Prints progress messages to the console.
class printer : public agent
{
public:
    explicit printer(ISource<wstring>& source,
        ISource<unsigned int>& progress)
        : _source(source)
        , _progress(progress)
    {
    }

    explicit printer(ISource<wstring>& source,
        ISource<unsigned int>& progress, Scheduler& scheduler)
        : agent(scheduler)
        , _source(source)
        , _progress(progress)
    {
    }

    explicit printer(ISource<wstring>& source,
        ISource<unsigned int>& progress, ScheduleGroup& group)
        : agent(group)
        , _source(source)
        , _progress(progress)
    {
    }

protected:
    // Performs the work of the agent.
    void run()
    {
        // Read the source string from the buffer and print a message.
        wstringstream ss;
        ss << L"Computing all permutations of '" << receive(_source) << L"'..." << endl;
        wcout << ss.str();

        // Print each progress message.

```

```

unsigned int previous_progress = 0u;
while (true)
{
    unsigned int progress = receive(_progress);

    if (progress > previous_progress || progress == 0u)
    {
        wstringstream ss;
        ss << L'\r' << progress << L"% complete...";
        wcout << ss.str();
        previous_progress = progress;
    }

    if (progress == 100)
        break;
}
wcout << endl;

// Set the status of the agent to agent_done.
done();
}

private:
// The buffer that contains the source string to permute.
ISource<wstring>& _source;

// The buffer that contains progress status.
ISource<unsigned int>& _progress;
};

// Computes all permutations of the given string.
void permute_string(const wstring& source,
    Scheduler& permutor_scheduler, Scheduler& printer_scheduler)
{
    // Message buffer that contains the source string.
    // The permutor and printer agents both read from this buffer.
    single_assignment<wstring> source_string;

    // Message buffer that contains the progress status.
    // The permutor agent writes to this buffer and the printer agent reads
    // from this buffer.
    unbounded_buffer<unsigned int> progress;

    // Create the agents with the appropriate schedulers.
    permutor agent1(source_string, progress, permutor_scheduler);
    printer agent2(source_string, progress, printer_scheduler);

    // Start the agents.
    agent1.start();
    agent2.start();

    // Write the source string to the message buffer. This will unblock the agents.
    send(source_string, source);

    // Wait for both agents to finish.
    agent::wait(&agent1);
    agent::wait(&agent2);
}

int wmain()
{
    const wstring source(L"Grapefruit");

    // Compute all permutations on the default scheduler.

    Scheduler* default_scheduler = CurrentScheduler::Get();

    wcout << L"With default scheduler: " << endl;
    permute_string(source, *default_scheduler, *default_scheduler);
}

```

```

wcout << endl;

// Compute all permutations again. This time, provide a scheduler that
// has higher context priority to the printer agent.

SchedulerPolicy printer_policy(1, ContextPriority, THREAD_PRIORITY_HIGHEST);
Scheduler* printer_scheduler = Scheduler::Create(printer_policy);

// Register to be notified when the scheduler shuts down.
HANDLE hShutdownEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
printer_scheduler->RegisterShutdownEvent(hShutdownEvent);

wcout << L"With higher context priority: " << endl;
permute_string(source, *default_scheduler, *printer_scheduler);
wcout << endl;

// Release the printer scheduler.
printer_scheduler->Release();

// Wait for the scheduler to shut down and destroy itself.
WaitForSingleObject(hShutdownEvent, INFINITE);

// Close the event handle.
CloseHandle(hShutdownEvent);
}

```

This example produces the following output.

```

With default scheduler:
Computing all permutations of 'Grapefruit'...
100% complete...

With higher context priority:
Computing all permutations of 'Grapefruit'...
100% complete...

```

Although both sets of tasks produce the same result, the version that uses a custom policy enables the `printer` object to run at an elevated priority so that it behaves more responsively.

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `permute-strings.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc permute-strings.cpp
```

## See also

[Scheduler Policies](#)

[Asynchronous Agents](#)

# Schedule Groups

3/4/2019 • 2 minutes to read • [Edit Online](#)

This document describes the role of schedule groups in the Concurrency Runtime. A *schedule group* affinitizes, or groups, related tasks together. Every scheduler has one or more schedule groups. Use schedule groups when you require a high degree of locality among tasks, for example, when a group of related tasks benefit from executing on the same processor node. Conversely, use scheduler instances when your application has specific quality requirements, for example, when you want to limit the amount of processing resources that are allocated to a set of tasks. For more information about scheduler instances, see [Scheduler Instances](#).

## TIP

The Concurrency Runtime provides a default scheduler, and therefore you are not required to create one in your application. Because the Task Scheduler helps you fine-tune the performance of your applications, we recommend that you start with the [Parallel Patterns Library \(PPL\)](#) or the [Asynchronous Agents Library](#) if you are new to the Concurrency Runtime.

Every `Scheduler` object has a default schedule group for every scheduling node. A *scheduling node* maps to the underlying system topology. The runtime creates one scheduling node for every processor package or Non-Uniform Memory Architecture (NUMA) node, whichever number is larger. If you do not explicitly associate a task with a schedule group, the scheduler chooses which group to add the task to.

The `SchedulingProtocol` scheduler policy influences the order in which the scheduler executes the tasks in each schedule group. When `SchedulingProtocol` is set to `EnhanceScheduleGroupLocality` (which is the default), the Task Scheduler chooses the next task from the schedule group that it is working on when the current task finishes or cooperatively yields. The Task Scheduler searches the current schedule group for work before it moves to the next available group. Conversely, when `SchedulingProtocol` is set to `EnhanceForwardProgress`, the scheduler moves to the next schedule group after each task finishes or yields. For an example that compares these policies, see [How to: Use Schedule Groups to Influence Order of Execution](#).

The runtime uses the `concurrency::ScheduleGroup` class to represent schedule groups. To create a `ScheduleGroup` object, call the `concurrency::CurrentScheduler::CreateScheduleGroup` or `concurrency::Scheduler::CreateScheduleGroup` method. The runtime uses a reference-counting mechanism to control the lifetime of `ScheduleGroup` objects, just as it does with `Scheduler` objects. When you create a `ScheduleGroup` object, the runtime sets the reference counter to one. The `concurrency::ScheduleGroup::Reference` method increments the reference counter by one. The `concurrency::ScheduleGroup::Release` method decrements the reference counter by one.

Many types in the Concurrency Runtime let you associate an object together with a schedule group. For example, the `concurrency::agent` class and message block classes such as `concurrency::unbounded_buffer`, `concurrency::join`, and `concurrency::timer`, provide overloaded versions of the constructor that take a `ScheduleGroup` object. The runtime uses the `Scheduler` object that is associated with this `ScheduleGroup` object to schedule the task.

You can also use the `concurrency::ScheduleGroup::ScheduleTask` method to schedule a lightweight task. For more information about lightweight tasks, see [Lightweight Tasks](#).

## Example

For an example that uses schedule groups to control the order of task execution, see [How to: Use Schedule Groups to Influence Order of Execution](#).

## See also

[Task Scheduler](#)

[Scheduler Instances](#)

[How to: Use Schedule Groups to Influence Order of Execution](#)

# How to: Use Schedule Groups to Influence Order of Execution

3/4/2019 • 5 minutes to read • [Edit Online](#)

In the Concurrency Runtime, the order in which tasks are scheduled is non-deterministic. However, you can use scheduling policies to influence the order in which tasks run. This topic shows how to use schedule groups together with the `concurrency::SchedulingProtocol` scheduler policy to influence the order in which tasks run.

The example runs a set of tasks two times, each with a different scheduling policy. Both policies limit the maximum number of processing resources to two. The first run uses the `EnhanceScheduleGroupLocality` policy, which is the default, and the second run uses the `EnhanceForwardProgress` policy. Under the `EnhanceScheduleGroupLocality` policy, the scheduler runs all tasks in one schedule group until each task finishes or yields. Under the `EnhanceForwardProgress` policy, the scheduler moves to the next schedule group in a round-robin manner after just one task finishes or yields.

When each schedule group contains related tasks, the `EnhanceScheduleGroupLocality` policy typically results in improved performance because cache locality is preserved between tasks. The `EnhanceForwardProgress` policy enables tasks to make forward progress and is useful when you require scheduling fairness across schedule groups.

## Example

This example defines the `work_yield_agent` class, which derives from `concurrency::agent`. The `work_yield_agent` class performs a unit of work, yields the current context, and then performs another unit of work. The agent uses the `concurrency::wait` function to cooperatively yield the current context so that other contexts can run.

This example creates four `work_yield_agent` objects. To illustrate how to set scheduler policies to affect the order in which the agents run, the example associates the first two agents with one schedule group and the other two agents with another schedule group. The example uses the `concurrency::CurrentScheduler::CreateScheduleGroup` method to create the `concurrency::ScheduleGroup` objects. The example runs all four agents two times, each time with a different scheduling policy.

```
// scheduling-protocol.cpp
// compile with: /EHsc
#include <agents.h>
#include <vector>
#include <algorithm>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

#pragma optimize( "", off )
// Simulates work by performing a long spin loop.
void spin_loop()
{
    for (int i = 0; i < 500000000; ++i)
    {
    }
}
#pragma optimize( "", on )

// Agent that performs some work and then yields the current context.
```

```

class work_yield_agent : public agent
{
public:
    explicit work_yield_agent(
        unsigned int group_number, unsigned int task_number)
        : _group_number(group_number)
        , _task_number(task_number)
    {
    }

    explicit work_yield_agent(Scheduler& scheduler,
        unsigned int group_number, unsigned int task_number)
        : agent(scheduler)
        , _group_number(group_number)
        , _task_number(task_number)
    {
    }

    explicit work_yield_agent(ScheduleGroup& group,
        unsigned int group_number, unsigned int task_number)
        : agent(group)
        , _group_number(group_number)
        , _task_number(task_number)
    {
    }

protected:
    // Performs the work of the agent.
    void run()
    {
        wstringstream header, ss;

        // Create a string that is prepended to each message.
        header << L"group " << _group_number
            << L",task " << _task_number << L": ";

        // Perform work.
        ss << header.str() << L"first loop..." << endl;
        wcout << ss.str();
        spin_loop();

        // Cooperatively yield the current context.
        // The task scheduler will then run all blocked contexts.
        ss = wstringstream();
        ss << header.str() << L"waiting..." << endl;
        wcout << ss.str();
        concurrency::wait(0);

        // Perform more work.
        ss = wstringstream();
        ss << header.str() << L"second loop..." << endl;
        wcout << ss.str();
        spin_loop();

        // Print a final message and then set the agent to the
        // finished state.
        ss = wstringstream();
        ss << header.str() << L"finished..." << endl;
        wcout << ss.str();

        done();
    }

private:
    // The group number that the agent belongs to.
    unsigned int _group_number;
    // A task number that is associated with the agent.
    unsigned int _task_number;
};

```

```

// Creates and runs several groups of agents. Each group of agents is associated
// with a different schedule group.
void run_agents()
{
    // The number of schedule groups to create.
    const unsigned int group_count = 2;
    // The number of agent to create per schedule group.
    const unsigned int tasks_per_group = 2;

    // A collection of schedule groups.
    vector<ScheduleGroup*> groups;
    // A collection of agents.
    vector<agent*> agents;

    // Create a series of schedule groups.
    for (unsigned int group = 0; group < group_count; ++group)
    {
        groups.push_back(CurrentScheduler::CreateScheduleGroup());

        // For each schedule group, create a series of agents.
        for (unsigned int task = 0; task < tasks_per_group; ++task)
        {
            // Add an agent to the collection. Pass the current schedule
            // group to the work_yield_agent constructor to schedule the agent
            // in this group.
            agents.push_back(new work_yield_agent(*groups.back(), group, task));
        }
    }

    // Start each agent.
    for_each(begin(agents), end(agents), [](agent* a) {
        a->start();
    });

    // Wait for all agents to finish.
    agent::wait_for_all(agents.size(), &agents[0]);

    // Free the memory that was allocated for each agent.
    for_each(begin(agents), end(agents), [](agent* a) {
        delete a;
    });

    // Release each schedule group.
    for_each(begin(groups), end(groups), [](ScheduleGroup* group) {
        group->Release();
    });
}

int wmain()
{
    // Run the agents two times. Each run uses a scheduler
    // policy that limits the maximum number of processing resources to two.

    // The first run uses the EnhanceScheduleGroupLocality
    // scheduling protocol.
    wcout << L"Using EnhanceScheduleGroupLocality..." << endl;
    CurrentScheduler::Create(SchedulerPolicy(3,
        MinConcurrency, 1,
        MaxConcurrency, 2,
        SchedulingProtocol, EnhanceScheduleGroupLocality));

    run_agents();
    CurrentScheduler::Detach();

    wcout << endl << endl;

    // The second run uses the EnhanceForwardProgress
    // scheduling protocol.

```

```

wcout << L"Using EnhanceForwardProgress..." << endl;
CurrentScheduler::Create(SchedulerPolicy(3,
    MinConcurrency, 1,
    MaxConcurrency, 2,
    SchedulingProtocol, EnhanceForwardProgress));

run_agents();
CurrentScheduler::Detach();
}

```

This example produces the following output.

```
Using EnhanceScheduleGroupLocality...
```

```

group 0,
    task 0: first loop...
group 0,
    task 1: first loop...
group 0,
    task 0: waiting...
group 1,
    task 0: first loop...
group 0,
    task 1: waiting...
group 1,
    task 1: first loop...
group 1,
    task 0: waiting...
group 0,
    task 0: second loop...
group 1,
    task 1: waiting...
group 0,
    task 1: second loop...
group 0,
    task 0: finished...
group 1,
    task 0: second loop...
group 0,
    task 1: finished...
group 1,
    task 1: second loop...
group 1,
    task 0: finished...
group 1,
    task 1: finished...

```

```
Using EnhanceForwardProgress...
```

```

group 0,
    task 0: first loop...
group 1,
    task 0: first loop...
group 0,
    task 0: waiting...
group 0,
    task 1: first loop...
group 1,
    task 0: waiting...
group 1,
    task 1: first loop...
group 0,
    task 1: waiting...
group 0,
    task 0: second loop...
group 1,
    task 1: waiting...
group 1,
    task 0: second loop...

```

```
group 0,
  task 0: finished...
group 0,
  task 1: second loop...
group 1,
  task 0: finished...
group 1,
  task 1: second loop...
group 0,
  task 1: finished...
group 1,
  task 1: finished...
```

Both policies produce the same sequence of events. However, the policy that uses `EnhanceScheduleGroupLocality` starts both agents that are part of the first schedule group before it starts the agents that are part of the second group. The policy that uses `EnhanceForwardProgress` starts one agent from the first group and then starts the first agent in the second group.

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `scheduling-protocol.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc scheduling-protocol.cpp
```

## See also

[Schedule Groups](#)

[Asynchronous Agents](#)

# Lightweight Tasks

3/4/2019 • 2 minutes to read • [Edit Online](#)

This document describes the role of lightweight tasks in the Concurrency Runtime. A *lightweight task* is a task that you schedule directly from a `concurrency::Scheduler` or `concurrency::ScheduleGroup` object. A lightweight task resembles the function that you provide to the Windows API `CreateThread` function. Therefore, lightweight tasks are useful when you adapt existing code to use the scheduling functionality of the Concurrency Runtime. The Concurrency Runtime itself uses lightweight tasks to schedule asynchronous agents and send messages between asynchronous message blocks.

## TIP

The Concurrency Runtime provides a default scheduler, and therefore you are not required to create one in your application. Because the Task Scheduler helps you fine-tune the performance of your applications, we recommend that you start with the [Parallel Patterns Library \(PPL\)](#) or the [Asynchronous Agents Library](#) if you are new to the Concurrency Runtime.

Lightweight tasks carry less overhead than asynchronous agents and task groups. For example, the runtime does not inform you when a lightweight task finishes. In addition, the runtime does not catch or handle exceptions that are thrown from a lightweight task. For more information about exception handling and lightweight tasks, see [Exception Handling](#).

For most tasks, we recommend that you use more robust functionality such as task groups and parallel algorithms because they let you more easily break complex tasks into more basic ones. For more information about task groups, see [Task Parallelism](#). For more information about parallel algorithms, see [Parallel Algorithms](#).

To create a lightweight task, call the `concurrency::ScheduleGroup::ScheduleTask`, `concurrency::CurrentScheduler::ScheduleTask`, or `concurrency::Scheduler::ScheduleTask` method. To wait for a lightweight task to finish, wait for the parent scheduler to shut down or use a synchronization mechanism such as a `concurrency::event` object.

## Example

For an example that demonstrates how to adapt existing code to use a lightweight task, see [Walkthrough: Adapting Existing Code to Use Lightweight Tasks](#).

## See also

[Task Scheduler](#)

[Walkthrough: Adapting Existing Code to Use Lightweight Tasks](#)

# Contexts

3/4/2019 • 4 minutes to read • [Edit Online](#)

This document describes the role of contexts in the Concurrency Runtime. A thread that is attached to a scheduler is known as an *execution context*, or just *context*. The `concurrency::wait` function and the `concurrency::Context` class enable you to control the behavior of contexts. Use the `wait` function to suspend the current context for a specified time. Use the `Context` class when you need more control over when contexts block, unblock, and yield, or when you want to oversubscribe the current context.

## TIP

The Concurrency Runtime provides a default scheduler, and therefore you are not required to create one in your application. Because the Task Scheduler helps you fine-tune the performance of your applications, we recommend that you start with the [Parallel Patterns Library \(PPL\)](#) or the [Asynchronous Agents Library](#) if you are new to the Concurrency Runtime.

## The wait Function

The `concurrency::wait` function cooperatively yields the execution of the current context for a specified number of milliseconds. The runtime uses the yield time to perform other tasks. After the specified time has elapsed, the runtime reschedules the context for execution. Therefore, the `wait` function might suspend the current context longer than the value provided for the `milliseconds` parameter.

Passing 0 (zero) for the `milliseconds` parameter causes the runtime to suspend the current context until all other active contexts are given the opportunity to perform work. This lets you yield a task to all other active tasks.

### Example

For an example that uses the `wait` function to yield the current context, and thus allow for other contexts to run, see [How to: Use Schedule Groups to Influence Order of Execution](#).

## The Context Class

The `concurrency::Context` class provides a programming abstraction for an execution context and offers two important features: the ability to cooperatively block, unblock, and yield the current context, and the ability to oversubscribe the current context.

### Cooperative Blocking

The `Context` class lets you block or yield the current execution context. Blocking or yielding is useful when the current context cannot continue because a resource is not available.

The `concurrency::Context::Block` method blocks the current context. A context that is blocked yields its processing resources so that the runtime can perform other tasks. The `concurrency::Context::Unblock` method unblocks a blocked context. The `Context::Unblock` method must be called from a different context than the one that called `Context::Block`. The runtime throws `concurrency::context_self_unblock` if a context attempts to unblock itself.

To cooperatively block and unblock a context, you typically call `concurrency::Context::CurrentContext` to retrieve a pointer to the `Context` object that is associated with the current thread and save the result. You then call the `Context::Block` method to block the current context. Later, call `Context::Unblock` from a separate context to unblock the blocked context.

You must match each pair of calls to `Context::Block` and `Context::Unblock`. The runtime throws

`concurrency::context_unblock_unbalanced` when the `Context::Block` or `Context::Unblock` method is called consecutively without a matching call to the other method. However, you do not have to call `Context::Block` before you call `Context::Unblock`. For example, if one context calls `Context::Unblock` before another context calls `Context::Block` for the same context, that context remains unblocked.

The `concurrency::Context::Yield` method yields execution so that the runtime can perform other tasks and then reschedule the context for execution. When you call the `Context::Block` method, the runtime does not reschedule the context.

#### Example

For an example that uses the `Context::Block`, `Context::Unblock`, and `Context::Yield` methods to implement a cooperative semaphore class, see [How to: Use the Context Class to Implement a Cooperative Semaphore](#).

#### Oversubscription

The default scheduler creates the same number of threads as there are available hardware threads. You can use *oversubscription* to create additional threads for a given hardware thread.

For computationally intensive operations, oversubscription typically does not scale because it introduces additional overhead. However, for tasks that have a high amount of latency, for example, reading data from disk or from a network connection, oversubscription can improve the overall efficiency of some applications.

#### NOTE

Enable oversubscription only from a thread that was created by the Concurrency Runtime. Oversubscription has no effect when it is called from a thread that was not created by the runtime (including the main thread).

To enable oversubscription in the current context, call the `concurrency::Context::Oversubscribe` method with the `_BeginOversubscription` parameter set to **true**. When you enable oversubscription on a thread that was created by the Concurrency Runtime, it causes the runtime to create one additional thread. After all tasks that require oversubscription finish, call `Context::Oversubscribe` with the `_BeginOversubscription` parameter set to **false**.

You can enable oversubscription multiple times from the current context, but you must disable it the same number of times that you enable it. Oversubscription can also be nested; that is, a task that is created by another task that uses oversubscription can also oversubscribe its context. However, if both a nested task and its parent belong to the same context, only the outermost call to `Context::Oversubscribe` causes the creation of an additional thread.

#### NOTE

The runtime throws `concurrency::invalid_oversubscribe_operation` if oversubscription is disabled before it is enabled.

#### Example

For an example that uses oversubscription to offset the latency that is caused by reading data from a network connection, see [How to: Use Oversubscription to Offset Latency](#).

## See also

[Task Scheduler](#)

[How to: Use Schedule Groups to Influence Order of Execution](#)

[How to: Use the Context Class to Implement a Cooperative Semaphore](#)

[How to: Use Oversubscription to Offset Latency](#)

# How to: Use the Context Class to Implement a Cooperative Semaphore

3/4/2019 • 6 minutes to read • [Edit Online](#)

This topic shows how to use the `concurrency::Context` class to implement a cooperative semaphore class.

The `Context` class lets you block or yield the current execution context. Blocking or yielding the current context is useful when the current context cannot proceed because a resource is not available. A *semaphore* is an example of one situation where the current execution context must wait for a resource to become available. A semaphore, like a critical section object, is a synchronization object that enables code in one context to have exclusive access to a resource. However, unlike a critical section object, a semaphore enables more than one context to access the resource concurrently. If the maximum number of contexts holds a semaphore lock, each additional context must wait for another context to release the lock.

## To implement the semaphore class

1. Declare a class that is named `semaphore`. Add `public` and `private` sections to this class.

```
// A semaphore type that uses cooperative blocking semantics.
class semaphore
{
public:
private:
};
```

1. In the `private` section of the `semaphore` class, declare a `std::atomic` variable that holds the semaphore count and a `concurrency::concurrent_queue` object that holds the contexts that must wait to acquire the semaphore.

```
// The semaphore count.
atomic<long long> _semaphore_count;

// A concurrency-safe queue of contexts that must wait to
// acquire the semaphore.
concurrent_queue<Context*> _waiting_contexts;
```

1. In the `public` section of the `semaphore` class, implement the constructor. The constructor takes a `long long` value that specifies the maximum number of contexts that can concurrently hold the lock.

```
explicit semaphore(long long capacity)
: _semaphore_count(capacity)
{
}
```

1. In the `public` section of the `semaphore` class, implement the `acquire` method. This method decrements the semaphore count as an atomic operation. If the semaphore count becomes negative, add the current context to the end of the wait queue and call the `concurrency::Context::Block` method to block the current context.

```
// Acquires access to the semaphore.
void acquire()
{
    // The capacity of the semaphore is exceeded when the semaphore count
    // falls below zero. When this happens, add the current context to the
    // back of the wait queue and block the current context.
    if (--_semaphore_count < 0)
    {
        _waiting_contexts.push(Context::CurrentContext());
        Context::Block();
    }
}
```

1. In the `public` section of the `semaphore` class, implement the `release` method. This method increments the semaphore count as an atomic operation. If the semaphore count is negative before the increment operation, there is at least one context that is waiting for the lock. In this case, unblock the context that is at the front of the wait queue.

```
// Releases access to the semaphore.
void release()
{
    // If the semaphore count is negative, unblock the first waiting context.
    if (++_semaphore_count <= 0)
    {
        // A call to acquire might have decremented the counter, but has not
        // yet finished adding the context to the queue.
        // Create a spin loop that waits for the context to become available.
        Context* waiting = NULL;
        while (!_waiting_contexts.try_pop(waiting))
        {
            Context::Yield();
        }

        // Unblock the context.
        waiting->Unblock();
    }
}
```

## Example

The `semaphore` class in this example behaves cooperatively because the `Context::Block` and `Context::Yield` methods yield execution so that the runtime can perform other tasks.

The `acquire` method decrements the counter, but it might not finish adding the context to the wait queue before another context calls the `release` method. To account for this, the `release` method uses a spin loop that calls the `concurrency::Context::Yield` method to wait for the `acquire` method to finish adding the context.

The `release` method can call the `Context::Unblock` method before the `acquire` method calls the `Context::Block` method. You do not have to protect against this race condition because the runtime allows for these methods to be called in any order. If the `release` method calls `Context::Unblock` before the `acquire` method calls `Context::Block` for the same context, that context remains unblocked. The runtime only requires that each call to `Context::Block` is matched with a corresponding call to `Context::Unblock`.

The following example shows the complete `semaphore` class. The `wmain` function shows basic usage of this class. The `wmain` function uses the `concurrency::parallel_for` algorithm to create several tasks that require access to the semaphore. Because three threads can hold the lock at any time, some tasks must wait for another task to finish and release the lock.

```

// cooperative-semaphore.cpp
// compile with: /EHsc
#include <atomic>
#include <concrct.h>
#include <ppl.h>
#include <concurrent_queue.h>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

// A semaphore type that uses cooperative blocking semantics.
class semaphore
{
public:
    explicit semaphore(long long capacity)
        : _semaphore_count(capacity)
    {
    }

    // Acquires access to the semaphore.
    void acquire()
    {
        // The capacity of the semaphore is exceeded when the semaphore count
        // falls below zero. When this happens, add the current context to the
        // back of the wait queue and block the current context.
        if (--_semaphore_count < 0)
        {
            _waiting_contexts.push(Context::CurrentContext());
            Context::Block();
        }
    }

    // Releases access to the semaphore.
    void release()
    {
        // If the semaphore count is negative, unblock the first waiting context.
        if (++_semaphore_count <= 0)
        {
            // A call to acquire might have decremented the counter, but has not
            // yet finished adding the context to the queue.
            // Create a spin loop that waits for the context to become available.
            Context* waiting = NULL;
            while (!_waiting_contexts.try_pop(waiting))
            {
                Context::Yield();
            }

            // Unblock the context.
            waiting->Unblock();
        }
    }

private:
    // The semaphore count.
    atomic<long long> _semaphore_count;

    // A concurrency-safe queue of contexts that must wait to
    // acquire the semaphore.
    concurrent_queue<Context*> _waiting_contexts;
};

int wmain()
{
    // Create a semaphore that allows at most three threads to
    // hold the lock.
    semaphore s(3);
}

```

```

parallel_for(0, 10, [&](int i) {
    // Acquire the lock.
    s.acquire();

    // Print a message to the console.
    wstringstream ss;
    ss << L"In loop iteration " << i << L"..." << endl;
    wcout << ss.str();

    // Simulate work by waiting for two seconds.
    wait(2000);

    // Release the lock.
    s.release();
});
}

```

This example produces the following sample output.

```

In loop iteration 5...
In loop iteration 0...
In loop iteration 6...
In loop iteration 1...
In loop iteration 2...
In loop iteration 7...
In loop iteration 3...
In loop iteration 8...
In loop iteration 9...
In loop iteration 4...

```

For more information about the `concurrent_queue` class, see [Parallel Containers and Objects](#). For more information about the `parallel_for` algorithm, see [Parallel Algorithms](#).

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `cooperative-semaphore.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc cooperative-semaphore.cpp
```

## Robust Programming

You can use the *Resource Acquisition Is Initialization* (RAII) pattern to limit access to a `semaphore` object to a given scope. Under the RAII pattern, a data structure is allocated on the stack. That data structure initializes or acquires a resource when it is created and destroys or releases that resource when the data structure is destroyed. The RAII pattern guarantees that the destructor is called before the enclosing scope exits. Therefore, the resource is correctly managed when an exception is thrown or when a function contains multiple `return` statements.

The following example defines a class that is named `scoped_lock`, which is defined in the `public` section of the `semaphore` class. The `scoped_lock` class resembles the `concurrency::critical_section::scoped_lock` and `concurrency::reader_writer_lock::scoped_lock` classes. The constructor of the `semaphore::scoped_lock` class acquires access to the given `semaphore` object and the destructor releases access to that object.

```

// An exception-safe RAI wrapper for the semaphore class.
class scoped_lock
{
public:
    // Acquires access to the semaphore.
    scoped_lock(semaphore& s)
        : _s(s)
    {
        _s.acquire();
    }
    // Releases access to the semaphore.
    ~scoped_lock()
    {
        _s.release();
    }

private:
    semaphore& _s;
};

```

The following example modifies the body of the work function that is passed to the `parallel_for` algorithm so that it uses RAI to ensure that the semaphore is released before the function returns. This technique ensures that the work function is exception-safe.

```

parallel_for(0, 10, [&](int i) {
    // Create an exception-safe scoped_lock object that holds the lock
    // for the duration of the current scope.
    semaphore::scoped_lock auto_lock(s);

    // Print a message to the console.
    wstringstream ss;
    ss << L"In loop iteration " << i << L"..." << endl;
    wcout << ss.str();

    // Simulate work by waiting for two seconds.
    wait(2000);
});

```

## See also

[Contexts](#)

[Parallel Containers and Objects](#)

# How to: Use Oversubscription to Offset Latency

3/4/2019 • 6 minutes to read • [Edit Online](#)

Oversubscription can improve the overall efficiency of some applications that contain tasks that have a high amount of latency. This topic illustrates how to use oversubscription to offset the latency that is caused by reading data from a network connection.

## Example

This example uses the [Asynchronous Agents Library](#) to download files from HTTP servers. The `http_reader` class derives from `concurrency::agent` and uses message passing to asynchronously read which URL names to download.

The `http_reader` class uses the `concurrency::task_group` class to concurrently read each file. Each task calls the `concurrency::Context::Oversubscribe` method with the `_BeginOversubscription` parameter set to **true** to enable oversubscription in the current context. Each task then uses the Microsoft Foundation Classes (MFC) `InternetSession` and `CHttpFile` classes to download the file. Finally, each task calls `Context::Oversubscribe` with the `_BeginOversubscription` parameter set to **false** to disable oversubscription.

When oversubscription is enabled, the runtime creates one additional thread in which to run tasks. Each of these threads can also oversubscribe the current context and thereby create additional threads. The `http_reader` class uses a `concurrency::unbounded_buffer` object to limit the number of threads that the application uses. The agent initializes the buffer with a fixed number of token values. For each download operation, the agent reads a token value from the buffer before the operation starts and then writes that value back to the buffer after the operation finishes. When the buffer is empty, the agent waits for one of the download operations to write a value back to the buffer.

The following example limits the number of simultaneous tasks to two times the number of available hardware threads. This value is a good starting point to use when you experiment with oversubscription. You can use a value that fits a particular processing environment or dynamically change this value to respond to the actual workload.

```
// download-oversubscription.cpp
// compile with: /EHsc /MD /D "_AFXDLL"
#define _WIN32_WINNT 0x0501
#include <afxinet.h>
#include <conctrm.h>
#include <agents.h>
#include <pp1.h>
#include <sstream>
#include <iostream>
#include <array>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}
```

```

// Downloads the file at the given URL.
CString GetHttpFile(CInternetSession& session, const CString& strUrl);

// Reads files from HTTP servers.
class http_reader : public agent
{
public:
    explicit http_reader(CInternetSession& session,
        ISource<string>& source,
        unsigned int& total_bytes,
        unsigned int max_concurrent_reads)
        : _session(session)
        , _source(source)
        , _total_bytes(total_bytes)
    {
        // Add one token to the available tasks buffer for each
        // possible concurrent read operation. The value of each token
        // is not important, but can be useful for debugging.
        for (unsigned int i = 0; i < max_concurrent_reads; ++i)
            send(_available_tasks, i);
    }

    // Signals to the agent that there are no more items to download.
    static const string input_sentinel;

protected:
    void run()
    {
        // A task group. Each task in the group downloads one file.
        task_group tasks;

        // Holds the total number of bytes downloaded.
        combinable<unsigned int> total_bytes;

        // Read from the source buffer until the application
        // sends the sentinel value.
        string url;
        while ((url = receive(_source)) != input_sentinel)
        {
            // Wait for a task to release an available slot.
            unsigned int token = receive(_available_tasks);

            // Create a task to download the file.
            tasks.run([&, token, url] {

                // Print a message.
                wstringstream ss;
                ss << L"Downloading " << url.c_str() << L"..." << endl;
                wcout << ss.str();

                // Download the file.
                string content = download(url);

                // Update the total number of bytes downloaded.
                total_bytes.local() += content.size();

                // Release the slot for another task.
                send(_available_tasks, token);
            });
        }

        // Wait for all tasks to finish.
        tasks.wait();

        // Compute the total number of bytes download on all threads.
        _total_bytes = total_bytes.combine(plus<unsigned int>());

        // Set the status of the agent to agent_done.
    }
};

```

```

done();
}

// Downloads the file at the given URL.
string download(const string& url)
{
    // Enable oversubscription.
    Context::Oversubscribe(true);

    // Download the file.
    string content = GetHttpFile(_session, url.c_str());

    // Disable oversubscription.
    Context::Oversubscribe(false);

    return content;
}

private:
    // Manages the network connection.
    CInternetSession& _session;
    // A message buffer that holds the URL names to download.
    ISource<string>& _source;
    // The total number of bytes downloaded
    unsigned int& _total_bytes;
    // Limits the agent to a given number of simultaneous tasks.
    unbounded_buffer<unsigned int> _available_tasks;
};
const string http_reader::input_sentinel("");

int wmain()
{
    // Create an array of URL names to download.
    // A real-world application might read the names from user input.
    array<string, 21> urls = {
        "http://www.adatum.com/",
        "http://www.adventure-works.com/",
        "http://www.alpineskihouse.com/",
        "http://www.cpandl.com/",
        "http://www.cohovineyard.com/",
        "http://www.cohowinery.com/",
        "http://www.cohovineyardandwinery.com/",
        "http://www.contoso.com/",
        "http://www.consolidatedmessenger.com/",
        "http://www.fabrikam.com/",
        "http://www.fourthcoffee.com/",
        "http://www.graphicdesigninstitute.com/",
        "http://www.humongousinsurance.com/",
        "http://www.litwareinc.com/",
        "http://www.lucernepublishing.com/",
        "http://www.margiestravel.com/",
        "http://www.northwindtraders.com/",
        "http://www.proseware.com/",
        "http://www.fineartschool.net",
        "http://www.tailspintoys.com/",
        http_reader::input_sentinel,
    };

    // Manages the network connection.
    CInternetSession session("Microsoft Internet Browser");

    // A message buffer that enables the application to send URL names to the
    // agent.
    unbounded_buffer<string> source_urls;

    // The total number of bytes that the agent has downloaded.
    unsigned int total_bytes = 0u;

    // Create an http_reader object that can oversubscribe each processor by one.

```

```

http_reader reader(session, source_urls, total_bytes, 2*GetProcessorCount());

// Compute the amount of time that it takes for the agent to download all files.
__int64 elapsed = time_call([&] {

    // Start the agent.
    reader.start();

    // Use the message buffer to send each URL name to the agent.
    for_each(begin(urls), end(urls), [&](const string& url) {
        send(source_urls, url);
    });

    // Wait for the agent to finish downloading.
    agent::wait(&reader);
});

// Print the results.
wcout << L"Downloaded " << total_bytes
        << L" bytes in " << elapsed << " ms." << endl;
}

// Downloads the file at the given URL and returns the size of that file.
CString GetHttpFile(CInternetSession& session, const CString& strUrl)
{
    CString strResult;

    // Reads data from an HTTP server.
    CHttpFile* pHttpFile = NULL;

    try
    {
        // Open URL.
        pHttpFile = (CHttpFile*)session.OpenURL(strUrl, 1,
            INTERNET_FLAG_TRANSFER_ASCII |
            INTERNET_FLAG_RELOAD | INTERNET_FLAG_DONT_CACHE);

        // Read the file.
        if(pHttpFile != NULL)
        {
            UINT uiBytesRead;
            do
            {
                char chBuffer[10000];
                uiBytesRead = pHttpFile->Read(chBuffer, sizeof(chBuffer));
                strResult += chBuffer;
            }
            while (uiBytesRead > 0);
        }
    }
    catch (CInternetException)
    {
        // TODO: Handle exception
    }

    // Clean up and return.
    delete pHttpFile;

    return strResult;
}

```

This example produces the following output on a computer that has four processors:

```
Downloading http://www.adatum.com/...
Downloading http://www.adventure-works.com/...
Downloading http://www.alpineskihouse.com/...
Downloading http://www.cpandl.com/...
Downloading http://www.cohovineyard.com/...
Downloading http://www.cohowinery.com/...
Downloading http://www.cohovineyardandwinery.com/...
Downloading http://www.contoso.com/...
Downloading http://www.consolidatedmessenger.com/...
Downloading http://www.fabrikam.com/...
Downloading http://www.fourthcoffee.com/...
Downloading http://www.graphicdesigninstitute.com/...
Downloading http://www.humongousinsurance.com/...
Downloading http://www.litwareinc.com/...
Downloading http://www.lucernepublishing.com/...
Downloading http://www.margiestravel.com/...
Downloading http://www.northwindtraders.com/...
Downloading http://www.proseware.com/...
Downloading http://www.fineartschool.net...
Downloading http://www.tailspintoys.com/...
Downloaded 1801040 bytes in 3276 ms.
```

The example can run faster when oversubscription is enabled because additional tasks run while other tasks wait for a latent operation to finish.

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `download-oversubscription.cpp` and then run one of the following commands in a **Visual Studio Command Prompt** window.

```
cl.exe /EHsc /MD /D "_AFXDLL" download-oversubscription.cpp
```

```
cl.exe /EHsc /MT download-oversubscription.cpp
```

## Robust Programming

Always disable oversubscription after you no longer require it. Consider a function that does not handle an exception that is thrown by another function. If you do not disable oversubscription before the function returns, any additional parallel work will also oversubscribe the current context.

You can use the *Resource Acquisition Is Initialization* (RAII) pattern to limit oversubscription to a given scope. Under the RAII pattern, a data structure is allocated on the stack. That data structure initializes or acquires a resource when it is created and destroys or releases that resource when the data structure is destroyed. The RAII pattern guarantees that the destructor is called before the enclosing scope exits. Therefore, the resource is correctly managed when an exception is thrown or when a function contains multiple `return` statements.

The following example defines a structure that is named `scoped_blocking_signal`. The constructor of the `scoped_blocking_signal` structure enables oversubscription and the destructor disables oversubscription.

```
struct scoped_blocking_signal
{
    scoped_blocking_signal()
    {
        concurrency::Context::Oversubscribe(true);
    }
    ~scoped_blocking_signal()
    {
        concurrency::Context::Oversubscribe(false);
    }
};
```

The following example modifies the body of the `download` method to use RAII to ensure that oversubscription is disabled before the function returns. This technique ensures that the `download` method is exception-safe.

```
// Downloads the file at the given URL.
string download(const string& url)
{
    scoped_blocking_signal signal;

    // Download the file.
    return string(GetHttpFile(_session, url.c_str()));
}
```

## See also

[Contexts](#)

[Context::Oversubscribe Method](#)

# Memory Management Functions

3/4/2019 • 2 minutes to read • [Edit Online](#)

This document describes the memory management functions that the Concurrency Runtime provides to help you allocate and free memory in a concurrent manner.

## TIP

The Concurrency Runtime provides a default scheduler, and therefore you are not required to create one in your application. Because the Task Scheduler helps you fine-tune the performance of your applications, we recommend that you start with the [Parallel Patterns Library \(PPL\)](#) or the [Asynchronous Agents Library](#) if you are new to the Concurrency Runtime.

The Concurrency Runtime provides two memory management functions that are optimized for allocating and freeing blocks of memory in a concurrent manner. The `concurrency::Alloc` function allocates a block of memory by using the specified size. The `concurrency::Free` function frees the memory that was allocated by `Alloc`.

## NOTE

The `Alloc` and `Free` functions rely on each other. Use the `Free` function only to release memory that you allocate by using the `Alloc` function. Also, when you use the `Alloc` function to allocate memory, use only the `Free` function to release that memory.

Use the `Alloc` and `Free` functions when you allocate and free a fixed set of allocation sizes from different threads or tasks. The Concurrency Runtime caches memory that it allocates from the C Runtime heap. The Concurrency Runtime holds a separate memory cache for each running thread; therefore, the runtime manages memory without the use of locks or memory barriers. An application benefits more from the `Alloc` and `Free` functions when the memory cache is accessed more frequently. For example, a thread that frequently calls both `Alloc` and `Free` benefits more than a thread that primarily calls `Alloc` or `Free`.

## NOTE

When you use these memory management functions, and your application uses lots of memory, the application may enter a low-memory condition sooner than you expect. Because the memory blocks that are cached by one thread are not available to any other thread, if one thread holds lots of memory, that memory is not available.

## Example

For an example that uses the `Alloc` and `Free` functions to improve memory performance, see [How to: Use Alloc and Free to Improve Memory Performance](#).

## See also

[Task Scheduler](#)

[How to: Use Alloc and Free to Improve Memory Performance](#)

# How to: Use Alloc and Free to Improve Memory Performance

3/4/2019 • 5 minutes to read • [Edit Online](#)

This document shows how to use the `concurrency::Alloc` and `concurrency::Free` functions to improve memory performance. It compares the time that is required to reverse the elements of an array in parallel for three different types that each specify the `new` and `delete` operators.

The `Alloc` and `Free` functions are most useful when multiple threads frequently call both `Alloc` and `Free`. The runtime holds a separate memory cache for each thread; therefore, the runtime manages memory without the use of locks or memory barriers.

## Example

The following example shows three types that each specify the `new` and `delete` operators. The `new_delete` class uses the global `new` and `delete` operators, the `malloc_free` class uses the C Runtime `malloc` and `free` functions, and the `Alloc_Free` class uses the Concurrency Runtime `Alloc` and `Free` functions.

```

// A type that defines the new and delete operators. These operators
// call the global new and delete operators, respectively.
class new_delete
{
public:
    static void* operator new(size_t size)
    {
        return ::operator new(size);
    }

    static void operator delete(void *p)
    {
        return ::operator delete(p);
    }

    int _data;
};

// A type that defines the new and delete operators. These operators
// call the C Runtime malloc and free functions, respectively.
class malloc_free
{
public:
    static void* operator new(size_t size)
    {
        return malloc(size);
    }
    static void operator delete(void *p)
    {
        return free(p);
    }

    int _data;
};

// A type that defines the new and delete operators. These operators
// call the Concurrency Runtime Alloc and Free functions, respectively.
class Alloc_Free
{
public:
    static void* operator new(size_t size)
    {
        return Alloc(size);
    }
    static void operator delete(void *p)
    {
        return Free(p);
    }

    int _data;
};

```

## Example

The following example shows the `swap` and `reverse_array` functions. The `swap` function exchanges the contents of the array at the specified indices. It allocates memory from the heap for the temporary variable. The `reverse_array` function creates a large array and computes the time that is required to reverse that array several times in parallel.

```

// Exchanges the contents of a[index1] with a[index2].
template<class T>
void swap(T* a, int index1, int index2)
{
    // For illustration, allocate memory from the heap.
    // This is useful when sizeof(T) is large.
    T* temp = new T;

    *temp = a[index1];
    a[index1] = a[index2];
    a[index2] = *temp;

    delete temp;
}

// Computes the time that it takes to reverse the elements of a
// large array of the specified type.
template <typename T>
__int64 reverse_array()
{
    const int size = 5000000;
    T* a = new T[size];

    __int64 time = 0;
    const int repeat = 11;

    // Repeat the operation several times to amplify the time difference.
    for (int i = 0; i < repeat; ++i)
    {
        time += time_call([&] {
            parallel_for(0, size/2, [&](int index)
            {
                swap(a, index, size-index-1);
            });
        });
    }

    delete[] a;
    return time;
}

```

## Example

The following example shows the `wmain` function, which computes the time that is required for the `reverse_array` function to act on the `new_delete`, `malloc_free`, and `Alloc_Free` types, each of which uses a different memory allocation scheme.

```

int wmain()
{
    // Compute the time that it takes to reverse large arrays of
    // different types.

    // new_delete
    wcout << L"Took " << reverse_array<new_delete>()
        << " ms with new/delete." << endl;

    // malloc_free
    wcout << L"Took " << reverse_array<malloc_free>()
        << " ms with malloc/free." << endl;

    // Alloc_Free
    wcout << L"Took " << reverse_array<Alloc_Free>()
        << " ms with Alloc/Free." << endl;
}

```

## Example

The complete example follows.

```

// allocators.cpp
// compile with: /EHsc
#include <windows.h>
#include <ppl.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}

// A type that defines the new and delete operators. These operators
// call the global new and delete operators, respectively.
class new_delete
{
public:
    static void* operator new(size_t size)
    {
        return ::operator new(size);
    }

    static void operator delete(void *p)
    {
        return ::operator delete(p);
    }

    int _data;
};

// A type that defines the new and delete operators. These operators
// call the C Runtime malloc and free functions, respectively.
class malloc_free
{
public:
    static void* operator new(size_t size)

```

```

{
    return malloc(size);
}
static void operator delete(void *p)
{
    return free(p);
}

int _data;
};

// A type that defines the new and delete operators. These operators
// call the Concurrency Runtime Alloc and Free functions, respectively.
class Alloc_Free
{
public:
    static void* operator new(size_t size)
    {
        return Alloc(size);
    }
    static void operator delete(void *p)
    {
        return Free(p);
    }

    int _data;
};

// Exchanges the contents of a[index1] with a[index2].
template<class T>
void swap(T* a, int index1, int index2)
{
    // For illustration, allocate memory from the heap.
    // This is useful when sizeof(T) is large.
    T* temp = new T;

    *temp = a[index1];
    a[index1] = a[index2];
    a[index2] = *temp;

    delete temp;
}

// Computes the time that it takes to reverse the elements of a
// large array of the specified type.
template <typename T>
__int64 reverse_array()
{
    const int size = 5000000;
    T* a = new T[size];

    __int64 time = 0;
    const int repeat = 11;

    // Repeat the operation several times to amplify the time difference.
    for (int i = 0; i < repeat; ++i)
    {
        time += time_call([&] {
            parallel_for(0, size/2, [&](int index)
            {
                swap(a, index, size-index-1);
            });
        });
    }

    delete[] a;
    return time;
}

```

```
int wmain()
{
    // Compute the time that it takes to reverse large arrays of
    // different types.

    // new_delete
    wcout << L"Took " << reverse_array<new_delete>()
        << " ms with new/delete." << endl;

    // malloc_free
    wcout << L"Took " << reverse_array<malloc_free>()
        << " ms with malloc/free." << endl;

    // Alloc_Free
    wcout << L"Took " << reverse_array<Alloc_Free>()
        << " ms with Alloc/Free." << endl;
}
```

This example produces the following sample output for a computer that has four processors.

```
Took 2031 ms with new/delete.
Took 1672 ms with malloc/free.
Took 656 ms with Alloc/Free.
```

In this example, the type that uses the `Alloc` and `Free` functions provides the best memory performance because the `Alloc` and `Free` functions are optimized for frequently allocating and freeing blocks of memory from multiple threads.

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `allocators.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc allocators.cpp
```

## See also

[Memory Management Functions](#)

[Alloc Function](#)

[Free Function](#)

# Concurrency Runtime Walkthroughs

3/4/2019 • 2 minutes to read • [Edit Online](#)

The scenario-based topics in this section show how to use many of the features of the Concurrency Runtime.

## In This Section

### [Walkthrough: Connecting Using Tasks and XML HTTP Requests](#)

Shows how to use the [IXMLHTTPRequest2](#) and [IXMLHTTPRequest2Callback](#) interfaces together with tasks to send HTTP GET and POST requests to a web service in a Universal Windows Platform (UWP) app.

### [Walkthrough: Creating an Agent-Based Application](#)

Describes how to create a basic agent-based application.

### [Walkthrough: Creating a Dataflow Agent](#)

Demonstrates how to create agent-based applications that are based on dataflow, instead of on control flow.

### [Walkthrough: Creating an Image-Processing Network](#)

Demonstrates how to create a network of asynchronous message blocks that perform image processing.

### [Walkthrough: Implementing Futures](#)

Shows how to asynchronously compute values for later use.

### [Walkthrough: Using join to Prevent Deadlock](#)

Uses the dining philosophers problem to illustrate how to use the [concurrency::join](#) class to prevent deadlock in your application.

### [Walkthrough: Removing Work from a User-Interface Thread](#)

Demonstrates how to improve the performance of an MFC application that draws the Mandelbrot fractal.

### [Walkthrough: Using the Concurrency Runtime in a COM-Enabled Application](#)

Demonstrates how to use the Concurrency Runtime in an application that uses the Component Object Model (COM).

### [Walkthrough: Adapting Existing Code to Use Lightweight Tasks](#)

Shows how to adapt existing code that uses the Windows API to create and execute a thread to use a lightweight task.

### [Walkthrough: Creating a Custom Message Block](#)

Describes how to create a custom message block type that orders incoming messages by priority.

## Related Sections

### [Concurrency Runtime](#)

Introduces the concurrent programming framework for Visual C++.

# Walkthrough: Connecting Using Tasks and XML HTTP Requests

4/25/2019 • 18 minutes to read • [Edit Online](#)

This example shows how to use the `IXMLHttpRequest2` and `IXMLHttpRequest2Callback` interfaces together with tasks to send HTTP GET and POST requests to a web service in a Universal Windows Platform (UWP) app. By combining `IXMLHttpRequest2` together with tasks, you can write code that composes with other tasks. For example, you can use the download task as part of a chain of tasks. The download task can also respond when work is canceled.

## TIP

You can also use the C++ REST SDK to perform HTTP requests from a UWP app using C++ app or from a desktop C++ app. For more info, see [C++ REST SDK \(Codename "Casablanca"\)](#).

For more information about tasks, see [Task Parallelism](#). For more information about how to use tasks in a UWP app, see [Asynchronous programming in C++](#) and [Creating Asynchronous Operations in C++ for UWP Apps](#).

This document first shows how to create `HttpRequest` and its supporting classes. It then shows how to use this class from a UWP app that uses C++ and XAML.

For an example that uses `IXMLHttpRequest2` but does not use tasks, see [Quickstart: Connecting using XML HTTP Request \(IXMLHttpRequest2\)](#).

## TIP

`IXMLHttpRequest2` and `IXMLHttpRequest2Callback` are the interfaces that we recommend for use in a UWP app. You can also adapt this example for use in a desktop app.

## Prerequisites

UWP support is optional in Visual Studio 2017 and later. To install it, open the Visual Studio Installer from the Windows Start menu and choose the version of Visual Studio you are using. Click the **Modify** button and make sure the **UWP Development** tile is checked. Under **Optional Components** make sure that **C++ UWP Tools** is checked. Use v141 for Visual Studio 2017 or v142 for Visual Studio 2019.

## Defining the `HttpRequest`, `HttpRequestBuffersCallback`, and `HttpRequestStringCallback` Classes

When you use the `IXMLHttpRequest2` interface to create web requests over HTTP, you implement the `IXMLHttpRequest2Callback` interface to receive the server response and react to other events. This example defines the `HttpRequest` class to create web requests, and the `HttpRequestBuffersCallback` and `HttpRequestStringCallback` classes to process responses. The `HttpRequestBuffersCallback` and `HttpRequestStringCallback` classes support the `HttpRequest` class; you work only with the `HttpRequest` class from application code.

The `GetAsync`, `PostAsync` methods of the `HttpRequest` class enable you to start HTTP GET and POST operations, respectively. These methods use the `HttpRequestStringCallback` class to read the server response as a string. The `SendAsync` and `ReadAsync` methods enable you to stream large content in chunks. These methods each return

`concurrency::task` to represent the operation. The `GetAsync` and `PostAsync` methods produce `task<std::wstring>` value, where the `wstring` part represents the server's response. The `SendAsync` and `ReadAsync` methods produce `task<void>` values; these tasks complete when the send and read operations complete.

Because the `IXMLHttpRequest2` interfaces act asynchronously, this example uses `concurrency::task_completion_event` to create a task that completes after the callback object completes or cancels the download operation. The `HttpRequest` class creates a task-based continuation from this task to set the final result. The `HttpRequest` class uses a task-based continuation to ensure that the continuation task runs even if the previous task produces an error or is canceled. For more information about task-based continuations, see [Task Parallelism](#)

To support cancellation, the `HttpRequest`, `HttpRequestBuffersCallback`, and `HttpRequestStringCallback` classes use cancellation tokens. The `HttpRequestBuffersCallback` and `HttpRequestStringCallback` classes use the `concurrency::cancellation_token::register_callback` method to enable the task completion event to respond to cancellation. This cancellation callback aborts the download. For more info about cancellation, see [Cancellation](#).

#### To Define the HttpRequest Class

1. From the main menu, choose **File > New > Project**.
2. Use the C++ **Blank App (Universal Windows)** template to create a blank XAML app project. This example names the project `UsingIXMLHttpRequest2`.
3. Add to the project a header file that is named `HttpRequest.h` and a source file that is named `HttpRequest.cpp`.
4. In `pch.h`, add this code:

```
#include <ppltasks.h>
#include <string>
#include <sstream>
#include <wrl.h>
#include <msxml6.h>
```

5. In `HttpRequest.h`, add this code:

```
#pragma once
#include "pch.h"

inline void CheckHResult(HRESULT hResult)
{
    if (hResult == E_ABORT)
    {
        concurrency::cancel_current_task();
    }
    else if (FAILED(hResult))
    {
        throw Platform::Exception::CreateException(hResult);
    }
}

namespace Web
{
    namespace Details
    {
        // Implementation of IXMLHttpRequest2Callback used when partial buffers are needed from the response.
        // When only the complete response is needed, use HttpRequestStringCallback instead.
        class HttpRequestBuffersCallback
        : public Microsoft::WRL::RuntimeClass<
            Microsoft::WRL::RuntimeClassFlags<Microsoft::WRL::ClassicCom>,
```

```

        IXMLHttpRequest2Callback,
        Microsoft::WRL::FtmBase>
{
public:
    HttpRequestBuffersCallback(IXMLHttpRequest2* httpRequest,
        concurrency::cancellation_token ct = concurrency::cancellation_token::none()) :
        request(httpRequest), cancellationToken(ct), responseReceived(false), dataHRESULT(S_OK),
        statusCode(200)
    {
        // Register a callback function that aborts the HTTP operation when
        // the cancellation token is canceled.
        if (cancellationToken != concurrency::cancellation_token::none())
        {
            registrationToken = cancellationToken.register_callback([this]()
            {
                if (request != nullptr)
                {
                    request->Abort();
                }
            });
        }

        dataEvent = concurrency::task_completion_event<void>();
    }

    // Called when the HTTP request is being redirected to a new URL.
    IFACEMETHODIMP OnRedirect(IXMLHttpRequest2*, PCWSTR)
    {
        return S_OK;
    }

    // Called when HTTP headers have been received and processed.
    IFACEMETHODIMP OnHeadersAvailable(IXMLHttpRequest2*, DWORD statusCode, PCWSTR reasonPhrase)
    {
        HRESULT hr = S_OK;

        // We must not propagate exceptions back to IXHR2.
        try
        {
            {
                this->statusCode = statusCode;
                this->reasonPhrase = reasonPhrase;

                concurrency::critical_section::scoped_lock lock(dataEventLock);
                dataEvent.set();
            }
            catch (std::bad_alloc&)
            {
                hr = E_OUTOFMEMORY;
            }
            return hr;
        }
    }

    // Called when a portion of the entity body has been received.
    IFACEMETHODIMP OnDataAvailable(IXMLHttpRequest2*, ISequentialStream* stream)
    {
        HRESULT hr = S_OK;

        // We must not propagate exceptions back to IXHR2.
        try
        {
            {
                // Store a reference on the stream so it can be accessed by the task.
                dataStream = stream;

                // The work must be done as fast as possible, and must not block this thread,
                // for example, waiting on another event object. Here we simply set an event
                // that can be processed by another thread.
                concurrency::critical_section::scoped_lock lock(dataEventLock);
                dataEvent.set();
            }
        }
    }
}

```

```

        catch (std::bad_alloc&)
        {
            hr = E_OUTOFMEMORY;
        }
        return hr;
    }

    // Called when the entire entity response has been received.
    IFACEMETHODIMP OnResponseReceived(IXMLHttpRequest2* xhr, ISequentialStream* responseStream)
    {
        responseReceived = true;
        return OnDataAvailable(xhr, responseStream);
    }

    // Called when an error occurs during the HTTP request.
    IFACEMETHODIMP OnError(IXMLHttpRequest2*, HRESULT hrError)
    {
        HRESULT hr = S_OK;

        // We must not propagate exceptions back to IXHR2.
        try
        {
            concurrency::critical_section::scoped_lock lock(dataEventLock);
            dataHResult = hrError;
            dataEvent.set();
        }
        catch (std::bad_alloc&)
        {
            hr = E_OUTOFMEMORY;
        }

        return hr;
    }

    // Create a task that completes when data is available, in an exception-safe way.
    concurrency::task<void> CreateDataTask();

    HRESULT GetError() const
    {
        return dataHResult;
    }

    int GetStatusCode() const
    {
        return statusCode;
    }

    std::wstring const& GetReasonPhrase() const
    {
        return reasonPhrase;
    }

    bool IsResponseReceived() const
    {
        return responseReceived;
    }

    // Copy bytes from the sequential stream into the buffer provided until
    // we reach the end of one or the other.
    unsigned int ReadData(
        _Out_writes_(outputBufferSize) byte* outputBuffer,
        unsigned int outputBufferSize);

private:
    ~HttpRequestBuffersCallback()
    {
        // Unregister the callback.
        if (cancellationToken != concurrency::cancellation_token::none())
        {

```

```

        cancellationToken.deregister_callback(registrationToken);
    }
}

// Signals that the download operation was canceled.
concurrency::cancellation_token cancellationToken;

// Used to unregister the cancellation token callback.
concurrency::cancellation_token_registration registrationToken;

// The IXMLHttpRequest2 that processes the HTTP request.
Microsoft::WRL::ComPtr<IXMLHttpRequest2> request;

// Task completion event that is set when data is available or error is triggered.
concurrency::task_completion_event<void> dataEvent;
concurrency::critical_section dataEventLock;

// We cannot store the error obtained from IXHR2 in the dataEvent since any value there is first-
writer-wins,
// whereas we want a subsequent error to override an initial success.
HRESULT dataHResult;

// Referenced pointer to the data stream.
Microsoft::WRL::ComPtr<ISequentialStream> dataStream;

// HTTP status code and reason returned by the server.
int statusCode;
std::wstring reasonPhrase;

// Whether the response has been completely received.
bool responseReceived;
};

};

// Utility class for performing asynchronous HTTP requests.
// This class only supports one outstanding request at a time.
class HttpRequest
{
public:
    HttpRequest();

    int GetStatusCode() const
    {
        return statusCode;
    }

    std::wstring const& GetReasonPhrase() const
    {
        return reasonPhrase;
    }

    // Whether the response has been completely received, if using ReadAsync().
    bool IsResponseComplete() const
    {
        return responseComplete;
    }

    // Start an HTTP GET on the specified URI. The returned task completes once the entire response
    // has been received, and the task produces the HTTP response text. The status code and reason
    // can be read with GetStatusCode() and GetReasonPhrase().
    concurrency::task<std::wstring> GetAsync(
        Windows::Foundation::Uri^ uri,
        concurrency::cancellation_token cancellationToken = concurrency::cancellation_token::none());

    // Start an HTTP POST on the specified URI, using a string body. The returned task produces the
    // HTTP response text. The status code and reason can be read with GetStatusCode() and
    GetReasonPhrase().
    concurrency::task<std::wstring> PostAsync(

```

```

    Windows::Foundation::Uri^ uri,
    PCWSTR contentType,
    IStream* postStream,
    uint64 postStreamSizeToSend,
    concurrency::cancellation_token cancellationToken = concurrency::cancellation_token::none());

    // Start an HTTP POST on the specified URI, using a stream body. The returned task produces the
    // HTTP response text. The status code and reason can be read with GetStatusCode() and
    GetReasonPhrase().
    concurrency::task<std::wstring> PostAsync(
        Windows::Foundation::Uri^ uri,
        const std::wstring& str,
        concurrency::cancellation_token cancellationToken = concurrency::cancellation_token::none());

    // Send a request but don't return the response. Instead, let the caller read it with ReadAsync().
    concurrency::task<void> SendAsync(
        const std::wstring& httpMethod,
        Windows::Foundation::Uri^ uri,
        concurrency::cancellation_token cancellationToken = concurrency::cancellation_token::none());

    // Read a chunk of data from the HTTP response, up to a specified length or until we reach the end
    // of the response, and store the value in the provided buffer. This is useful for large content,
    // enabling the streaming of the result.
    concurrency::task<void> ReadAsync(
        Windows::Storage::Streams::IBuffer^ readBuffer,
        unsigned int offsetInBuffer,
        unsigned int requestedBytesToRead);

    static void CreateMemoryStream(IStream **stream);

private:
    // Start a download of the specified URI using the specified method. The returned task produces
    the
    // HTTP response text. The status code and reason can be read with GetStatusCode() and
    GetReasonPhrase().
    concurrency::task<std::wstring> DownloadAsync(
        PCWSTR httpMethod,
        PCWSTR uri,
        concurrency::cancellation_token cancellationToken,
        PCWSTR contentType,
        IStream* postStream,
        uint64 postStreamBytesToSend);

    // Referenced pointer to the callback, if using SendAsync/ReadAsync.
    Microsoft::WRL::ComPtr<Details::HttpRequestBuffersCallback> buffersCallback;

    int statusCode;
    std::wstring reasonPhrase;

    // Whether the response has been completely received, if using ReadAsync().
    bool responseComplete;
};

};

```

6. In HttpRequest.cpp, add this code:

```

#include "pch.h"
#include "HttpRequest.h"
#include <robuffer.h>
#include <shcore.h>

using namespace concurrency;
using namespace Microsoft::WRL;
using namespace Platform;
using namespace std;
using namespace Web;

```

```

using namespace Windows::Foundation;
using namespace Windows::Storage::Streams;

// Implementation of IXMLHttpRequest2Callback used when only the complete response is needed.
// When processing chunks of response data as they are received, use HttpRequestBuffersCallback
instead.
class HttpRequestStringCallback
    : public RuntimeClass<RuntimeClassFlags<ClassicCom>, IXMLHttpRequest2Callback, FtmBase>
{
public:
    HttpRequestStringCallback(IXMLHttpRequest2* httpRequest,
        cancellation_token ct = concurrency::cancellation_token::none()) :
        request(httpRequest), cancellationToken(ct)
    {
        // Register a callback function that aborts the HTTP operation when
        // the cancellation token is canceled.
        if (cancellationToken != cancellation_token::none())
        {
            registrationToken = cancellationToken.register_callback([this]()
            {
                if (request != nullptr)
                {
                    request->Abort();
                }
            });
        }

        // Called when the HTTP request is being redirected to a new URL.
        IFACEMETHODIMP OnRedirect(IXMLHttpRequest2*, PCWSTR)
        {
            return S_OK;
        }

        // Called when HTTP headers have been received and processed.
        IFACEMETHODIMP OnHeadersAvailable(IXMLHttpRequest2*, DWORD statusCode, PCWSTR reasonPhrase)
        {
            HRESULT hr = S_OK;

            // We must not propagate exceptions back to IXHR2.
            try
            {
                this->statusCode = statusCode;
                this->reasonPhrase = reasonPhrase;
            }
            catch (std::bad_alloc&)
            {
                hr = E_OUTOFMEMORY;
            }

            return hr;
        }

        // Called when a portion of the entity body has been received.
        IFACEMETHODIMP OnDataAvailable(IXMLHttpRequest2*, ISequentialStream*)
        {
            return S_OK;
        }

        // Called when the entire entity response has been received.
        IFACEMETHODIMP OnResponseReceived(IXMLHttpRequest2*, ISequentialStream* responseStream)
        {
            wstring wstr;
            HRESULT hr = ReadUtf8StringFromSequentialStream(responseStream, wstr);

            // We must not propagate exceptions back to IXHR2.
            try
            {
                completionEvent.set(make_tuple<HRESULT, wstring>(move(hr), move(wstr)));
            }
        }
    }
};

```

```

    }
    catch (std::bad_alloc&)
    {
        hr = E_OUTOFMEMORY;
    }

    return hr;
}

// Simulate the functionality of DataReader.ReadString().
// This is needed because DataReader requires IRandomAccessStream and this
// code has an ISequentialStream that does not have a conversion to IRandomAccessStream like
IStream does.
HRESULT ReadUtf8StringFromSequentialStream(ISequentialStream* readStream, wstring& str)
{
    // Convert the response to Unicode wstring.
    HRESULT hr;

    // Holds the response as a Unicode string.
    wstringstream ss;

    while (true)
    {
        ULONG cb;
        char buffer[4096];

        // Read the response as a UTF-8 string. Since UTF-8 characters are 1-4 bytes long,
        // we need to make sure we only read an integral number of characters. So we'll
        // start with 4093 bytes.
        hr = readStream->Read(buffer, sizeof(buffer) - 3, &cb);
        if (FAILED(hr) || (cb == 0))
        {
            break; // Error or no more data to process, exit loop.
        }

        if (cb == sizeof(buffer) - 3)
        {
            ULONG subsequentBytesRead;
            unsigned int i, c1;

            // Find the first byte of the last UTF-8 character in the buffer.
            for (i = cb - 1; (i >= 0) && ((buffer[i] & 0xC0) == 0x80); i--);

            // Calculate the number of subsequent bytes in the UTF-8 character.
            if (((unsigned char)buffer[i]) < 0x80)
            {
                c1 = 1;
            }
            else if (((unsigned char)buffer[i]) < 0xE0)
            {
                c1 = 2;
            }
            else if (((unsigned char)buffer[i]) < 0xF0)
            {
                c1 = 3;
            }
            else
            {
                c1 = 4;
            }

            // Read any remaining bytes.
            if (cb < i + c1)
            {
                hr = readStream->Read(buffer + cb, i + c1 - cb, &subsequentBytesRead);
                if (FAILED(hr))
                {
                    break; // Error, exit loop.
                }
            }
        }
    }
}

```

```

        }
        cb += subsequentBytesRead;
    }
}

// First determine the size required to store the Unicode string.
int const sizeRequired = MultiByteToWideChar(CP_UTF8, 0, buffer, cb, nullptr, 0);
if (sizeRequired == 0)
{
    // Invalid UTF-8.
    hr = HRESULT_FROM_WIN32(GetLastError());
    break;
}
unique_ptr<char16[]> wstr(new(std::nothrow) char16[sizeRequired + 1]);
if (wstr.get() == nullptr)
{
    hr = E_OUTOFMEMORY;
    break;
}

// Convert the string from UTF-8 to UTF-16LE. This can never fail, since
// the previous call above succeeded.
MultiByteToWideChar(CP_UTF8, 0, buffer, cb, wstr.get(), sizeRequired);
wstr[sizeRequired] = L'\0'; // Terminate the string.
ss << wstr.get(); // Write the string to the stream.
}

str = SUCCEEDED(hr) ? ss.str() : wstring();
return (SUCCEEDED(hr)) ? S_OK : hr; // Don't return S_FALSE.
}

// Called when an error occurs during the HTTP request.
IFACEMETHODIMP OnError(IXMLHttpRequest2*, HRESULT hrError)
{
    HRESULT hr = S_OK;

    // We must not propagate exceptions back to IXHR2.
    try
    {
        completionEvent.set(make_tuple<HRESULT, wstring>(move(hrError), wstring()));
    }
    catch (std::bad_alloc&)
    {
        hr = E_OUTOFMEMORY;
    }

    return hr;
}

// Retrieves the completion event for the HTTP operation.
task_completion_event<tuple<HRESULT, wstring>> const& GetCompletionEvent() const
{
    return completionEvent;
}

int GetStatusCode() const
{
    return statusCode;
}

wstring GetReasonPhrase() const
{
    return reasonPhrase;
}

private:
~HttpRequestStringCallback()
{
    // Unregister the callback.
}

```

```

        if (cancellationToken != cancellation_token::none())
        {
            cancellationToken.deregister_callback(registrationToken);
        }
    }

    // Signals that the download operation was canceled.
    cancellation_token cancellationToken;

    // Used to unregister the cancellation token callback.
    cancellation_token_registration registrationToken;

    // The IXMLHttpRequest2 that processes the HTTP request.
    ComPtr<IXMLHttpRequest2> request;

    // Task completion event that is set when the
    // download operation completes.
    task_completion_event<tuple<HRESULT, wstring>> completionEvent;

    int statusCode;
    wstring reasonPhrase;
};

// Copy bytes from the sequential stream into the buffer provided until
// we reach the end of one or the other.
unsigned int Web::Details::HttpRequestBuffersCallback::ReadData(
    _Out_writes_(outputBufferSize) byte* outputBuffer,
    unsigned int outputBufferSize)
{
    // Lock the data event while doing the read, to ensure that any bytes we don't read will
    // result in the correct event getting triggered.
    concurrency::critical_section::scoped_lock lock(dataEventLock);

    ULONG bytesRead;
    CheckHRESULT(dataStream.Get()->Read(outputBuffer, outputBufferSize, &bytesRead));
    if (bytesRead < outputBufferSize)
    {
        // We need to reset the data event, which we can only do by creating a new one.
        dataEvent = task_completion_event<void>();
    }

    return bytesRead;
}

// Create a task that completes when data is available, in an exception-safe way.
task<void> Web::Details::HttpRequestBuffersCallback::CreateDataTask()
{
    concurrency::critical_section::scoped_lock lock(dataEventLock);
    return create_task(dataEvent, cancellationToken);
}

HttpRequest::HttpRequest() : responseComplete(true), statusCode(200)
{
}

// Start a download of the specified URI using the specified method. The returned task produces the
// HTTP response text. The status code and reason can be read with GetStatusCode() and
// GetReasonPhrase().
task<wstring> HttpRequest::DownloadAsync(PCWSTR httpMethod, PCWSTR uri, cancellation_token
cancellationToken,
    PCWSTR contentType, IStream* postStream, uint64 postStreamSizeToSend)
{
    // Create an IXMLHttpRequest2 object.
    ComPtr<IXMLHttpRequest2> xhr;
    CheckHRESULT(CoCreateInstance(CLSID_XmlHttpRequest, nullptr, CLSCTX_INPROC, IID_PPV_ARGS(&xhr)));

    // Create callback.
    auto stringCallback = Make<HttpRequestStringCallback>(xhr.Get(), cancellationToken);
    CheckHRESULT(stringCallback ? S_OK : E_OUTOFMEMORY);
}

```

```

auto completionTask = create_task(stringCallback->GetCompletionEvent());

// Create a request.
CheckHRESULT(xhr->Open(httpMethod, uri, stringCallback.Get(), nullptr, nullptr, nullptr, nullptr));

if (postStream != nullptr && contentType != nullptr)
{
    CheckHRESULT(xhr->SetRequestHeader(L"Content-Type", contentType));
}

// Send the request.
CheckHRESULT(xhr->Send(postStream, postStreamSizeToSend));

// Return a task that completes when the HTTP operation completes.
// We pass the callback to the continuation because the lifetime of the
// callback must exceed the operation to ensure that cancellation
// works correctly.
return completionTask.then([this, stringCallback](tuple<HRESULT, wstring> resultTuple)
{
    // If the GET operation failed, throw an Exception.
    CheckHRESULT(std::get<0>(resultTuple));

    statusCode = stringCallback->GetStatusCode();
    reasonPhrase = stringCallback->GetReasonPhrase();

    return std::get<1>(resultTuple);
});
}

// Start an HTTP GET on the specified URI. The returned task completes once the entire response
// has been received, and the task produces the HTTP response text. The status code and reason
// can be read with GetStatusCode() and GetReasonPhrase().
task<wstring> HttpRequest::GetAsync(Uri^ uri, cancellation_token cancellationToken)
{
    return DownloadAsync(L"GET",
        uri->AbsoluteUri->Data(),
        cancellationToken,
        nullptr,
        nullptr,
        0);
}

void HttpRequest::CreateMemoryStream(IStream **stream)
{
    auto randomAccessStream = ref new Windows::Storage::Streams::InMemoryRandomAccessStream();
    CheckHRESULT(CreateStreamOverRandomAccessStream(randomAccessStream, IID_PPV_ARGS(stream)));
}

// Start an HTTP POST on the specified URI, using a string body. The returned task produces the
// HTTP response text. The status code and reason can be read with GetStatusCode() and
// GetReasonPhrase().
task<wstring> HttpRequest::PostAsync(Uri^ uri, const wstring& body, cancellation_token
cancellationToken)
{
    int length = 0;
    ComPtr<IStream> postStream;
    CreateMemoryStream(&postStream);

    if (body.length() > 0)
    {
        // Get the required buffer size.
        int size = WideCharToMultiByte(CP_UTF8, // UTF-8
            0, // Conversion type
            body.c_str(), // Unicode string to convert
            static_cast<int>(body.length()), // Size
            nullptr, // Output buffer
            0, // Output buffer size
            nullptr,

```

```

        nullptr);
    CheckHRESULT((size != 0) ? S_OK : HRESULT_FROM_WIN32(GetLastError()));

    std::unique_ptr<char[]> tempData(new char[size]);
    length = WideCharToMultiByte(CP_UTF8, // UTF-8
                                0, // Conversion type
                                body.c_str(), // Unicode string to convert
                                static_cast<int>(body.length()), // Size
                                tempData.get(), // Output buffer
                                size, // Output buffer size
                                nullptr,
                                nullptr);
    CheckHRESULT((length != 0) ? S_OK : HRESULT_FROM_WIN32(GetLastError()));
    CheckHRESULT(postStream->Write(tempData.get(), length, nullptr));
}

return DownloadAsync(L"POST",
                    uri->AbsoluteUri->Data(),
                    cancellationToken,
                    L"text/plain;charset=utf-8",
                    postStream.Get(),
                    length);
}

// Start an HTTP POST on the specified URI, using a stream body. The returned task produces the
// HTTP response text. The status code and reason can be read with GetStatusCode() and
GetReasonPhrase().
task<wstring> HttpRequest::PostAsync(Uri^ uri, PCWSTR contentType, IStream* postStream,
    uint64 postStreamSizeToSend, cancellation_token cancellationToken)
{
    return DownloadAsync(L"POST",
                        uri->AbsoluteUri->Data(),
                        cancellationToken,
                        contentType,
                        postStream,
                        postStreamSizeToSend);
}

// Send a request but don't return the response. Instead, let the caller read it with ReadAsync().
task<void> HttpRequest::SendAsync(const wstring& httpMethod, Uri^ uri, cancellation_token
cancellationToken)
{
    // Create an IXMLHttpRequest2 object.
    ComPtr<IXMLHttpRequest2> xhr;
    CheckHRESULT(CoCreateInstance(CLSID_XmlHttpRequest, nullptr, CLSCTX_INPROC, IID_PPV_ARGS(&xhr)));

    // Create callback.
    buffersCallback = Make<Web::Details::HttpRequestBuffersCallback>(xhr.Get(), cancellationToken);
    CheckHRESULT(buffersCallback ? S_OK : E_OUTOFMEMORY);

    ComPtr<IXMLHttpRequest2Callback> xhrCallback;
    CheckHRESULT(buffersCallback.As(&xhrCallback));

    // Open and send the request.
    CheckHRESULT(xhr->Open(httpMethod.c_str(),
                          uri->AbsoluteUri->Data(),
                          xhrCallback.Get(),
                          nullptr,
                          nullptr,
                          nullptr,
                          nullptr));

    responseComplete = false;

    CheckHRESULT(xhr->Send(nullptr, 0));

    // Return a task that completes when the HTTP operation completes.
    // Since buffersCallback holds a reference on the callback, the lifetime of the callback will
exceed

```

```

// the operation and ensure that cancellation works correctly.
return buffersCallback->CreateDataTask().then([this]()
{
    CheckHRESULT(buffersCallback->GetError());

    statusCode = buffersCallback->GetStatusCode();
    reasonPhrase = buffersCallback->GetReasonPhrase();
});
}

// Read a chunk of data from the HTTP response, up to a specified length or until we reach the end
// of the response, and store the value in the provided buffer. This is useful for large content,
// enabling the streaming of the result.
task<void> HttpRequest::ReadAsync(Windows::Storage::Streams::IBuffer^ readBuffer, unsigned int
offsetInBuffer,
    unsigned int requestedBytesToRead)
{
    if (offsetInBuffer + requestedBytesToRead > readBuffer->Capacity)
    {
        throw ref new InvalidArgumentException();
    }

    // Return a task that completes when a read completes.
    // We pass the callback to the continuation because the lifetime of the
    // callback must exceed the operation to ensure that cancellation
    // works correctly.
    return buffersCallback->CreateDataTask().then([this, readBuffer, offsetInBuffer,
requestedBytesToRead]()
    {
        CheckHRESULT(buffersCallback->GetError());

        // Get a pointer to the location to copy data into.
        ComPtr<IBufferByteAccess> bufferByteAccess;
        CheckHRESULT(reinterpret_cast<IUnknown*>(readBuffer)-
>QueryInterface(IID_PPV_ARGS(&bufferByteAccess)));
        byte* outputBuffer; // Returned internal pointer, do not free this value.
        CheckHRESULT(bufferByteAccess->Buffer(&outputBuffer));

        // Copy bytes from the sequential stream into the buffer provided until
        // we reach the end of one or the other.
        readBuffer->Length = buffersCallback->ReadData(outputBuffer + offsetInBuffer,
requestedBytesToRead);
        if (buffersCallback->IsResponseReceived() && (readBuffer->Length < requestedBytesToRead))
        {
            responseComplete = true;
        }
    });
}

```

## Using the HttpRequest Class in a UWP App

This section demonstrates how to use the `HttpRequest` class in a UWP app. The app provides an input box that defines a URL resource, and button commands that perform GET and POST operations, and a button command that cancels the current operation.

### To Use the HttpRequest Class

1. In `MainPage.xaml`, define the `StackPanel` element as follows.

```

<StackPanel HorizontalAlignment="Left" Width="440"
    Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <TextBox x:Name="InputTextBox" TextWrapping="Wrap"
        Text="http://www.fourthcoffee.com/" />
    <StackPanel Orientation="Horizontal">
        <Button x:Name="GetButton" Content="Get" Background="Green"
            Click="GetButton_Click" />
        <Button x:Name="PostButton" Content="Post" Background="Blue"
            Click="PostButton_Click" />
        <Button x:Name="CancelButton" Content="Cancel" Background="Red"
            IsEnabled="False" Click="CancelButton_Click" />
        <ProgressRing x:Name="ResponseProgressRing" />
    </StackPanel>
    <TextBlock x:Name="ResponseTextBlock" TextWrapping="Wrap" />
</StackPanel>

```

2. In MainPage.xaml.h, add this `#include` directive:

```
#include "HttpRequest.h"
```

3. In MainPage.xaml.h, add these `private` member variables to the `MainPage` class:

```

// Produces HTTP requests.
Web::HttpRequest m_httpRequest;
// Enables us to cancel the active HTTP request.
concurrency::cancellation_token_source m_cancelHttpRequestSource;

```

4. In MainPage.xaml.h, declare the `private` method `ProcessHttpRequest`:

```

// Displays the result of the provided HTTP request on the UI.
void ProcessHttpRequest(concurrency::task<std::wstring> httpRequest);

```

5. In MainPage.xaml.cpp, add these `using` statements:

```

using namespace concurrency;
using namespace std;
using namespace Web;

```

6. In MainPage.xaml.cpp, implement the `GetButton_Click`, `PostButton_Click`, and `CancelButton_Click` methods of the `MainPage` class.

```

void MainPage::GetButton_Click(Object^ sender, RoutedEventArgs^ e)
{
    // Create a new cancellation token source for the web request.
    m_cancelHttpRequestSource = cancellation_token_source();

    // Set up the GET request parameters.
    auto uri = ref new Uri(InputTextBox->Text);
    auto token = m_cancelHttpRequestSource.get_token();

    // Send the request and then update the UI.
    ProcessHttpRequest(m_httpRequest.GetAsync(uri, token));
}

void MainPage::PostButton_Click(Object^ sender, RoutedEventArgs^ e)
{
    // Create a new cancellation token source for the web request.
    m_cancelHttpRequestSource = cancellation_token_source();

    // Set up the POST request parameters.
    auto uri = ref new Uri(InputTextBox->Text);
    wstring postData(L"This is sample POST data.");
    auto token = m_cancelHttpRequestSource.get_token();

    // Send the request and then update the UI.
    ProcessHttpRequest(m_httpRequest.PostAsync(uri, postData, token));
}

void MainPage::CancelButton_Click(Object^ sender, RoutedEventArgs^ e)
{
    // Disable the Cancel button.
    // It will be re-enabled during the next web request.
    CancelButton->IsEnabled = false;

    // Initiate cancellation.
    m_cancelHttpRequestSource.cancel();
}

```

#### TIP

If your app does not require support for cancellation, pass `concurrency::cancellation_token::none` to the `HttpRequest::GetAsync` and `HttpRequest::PostAsync` methods.

7. In `MainPage.xaml.cpp`, implement the `MainPage::ProcessHttpRequest` method.

```

// Displays the result of the provided HTTP request on the UI.
void MainPage::ProcessHttpRequest(task<wstring> httpRequest)
{
    // Enable only the Cancel button.
    GetButton->IsEnabled = false;
    PostButton->IsEnabled = false;
    CancelButton->IsEnabled = true;

    // Clear the previous response and start the progress ring.
    ResponseTextBlock->Text = "";
    ResponseProgressRing->IsActive = true;

    // Create a continuation that shows the results on the UI.
    // The UI must be updated on the ASTA thread.
    // Therefore, schedule the continuation to run on the current context.
    httpRequest.then([this](task<wstring> previousTask)
    {
        try
        {
            //
            // Show the result on the UI.

            wstring response = previousTask.get();
            if (m_httpRequest.GetStatusCode() == 200)
            {
                // The request succeeded. Show the response.
                ResponseTextBlock->Text = ref new String(response.c_str());
            }
            else
            {
                // The request failed. Show the status code and reason.
                wstringstream ss;
                ss << L"The server returned "
                    << m_httpRequest.GetStatusCode()
                    << L" ("
                    << m_httpRequest.GetReasonPhrase()
                    << L')';
                ResponseTextBlock->Text = ref new String(ss.str().c_str());
            }
        }
        catch (const task_canceled&)
        {
            // Indicate that the operation was canceled.
            ResponseTextBlock->Text = "The operation was canceled";
        }
        catch (Exception^ e)
        {
            // Indicate that the operation failed.
            ResponseTextBlock->Text = "The operation failed";

            // TODO: Handle the error further.
            (void)e;
        }

        // Enable the Get and Post buttons.
        GetButton->IsEnabled = true;
        PostButton->IsEnabled = true;
        CancelButton->IsEnabled = false;

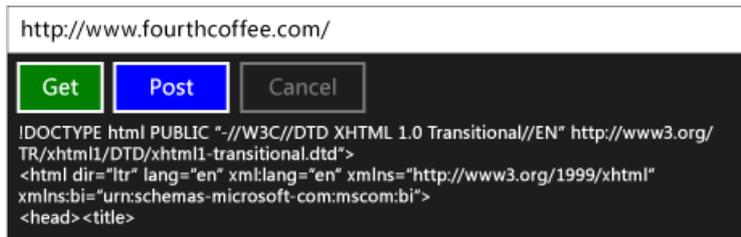
        // Stop the progress ring.
        ResponseProgressRing->IsActive = false;

    }, task_continuation_context::use_current());
}

```

8. In the project properties, under **Linker, Input**, specify `shcore.lib` and `msxml6.lib`.

Here is the running app:



## Next Steps

[Concurrency Runtime Walkthroughs](#)

## See also

[Task Parallelism](#)

[Cancellation in the PPL](#)

[Asynchronous programming in C++](#)

[Creating Asynchronous Operations in C++ for UWP Apps](#)

[Quickstart: Connecting using XML HTTP Request \(IXMLHttpRequest2\) task Class \(Concurrency Runtime\)](#)

[task\\_completion\\_event Class](#)

# Walkthrough: Creating an Agent-Based Application

4/25/2019 • 9 minutes to read • [Edit Online](#)

This topic describes how to create a basic agent-based application. In this walkthrough, you can create an agent that reads data from a text file asynchronously. The application uses the Adler-32 checksum algorithm to calculate the checksum of the contents of that file.

## Prerequisites

You must understand the following topics to complete this walkthrough:

- [Asynchronous Agents](#)
- [Asynchronous Message Blocks](#)
- [Message Passing Functions](#)
- [Synchronization Data Structures](#)

## Sections

This walkthrough demonstrates how to perform the following tasks:

- [Creating the Console Application](#)
- [Creating the file\\_reader Class](#)
- [Using the file\\_reader Class in the Application](#)

## Creating the Console Application

This section shows how to create a C++ console application that references the header files that the program will use. The initial steps vary depending on which version of Visual Studio you are using. Make sure the version selector is set correctly in the upper left of this page.

### To create a C++ console application in Visual Studio 2019

1. From the main menu, choose **File** > **New** > **Project** to open the **Create a New Project** dialog box.
2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Console**.
3. From the filtered list of project types, choose **Console App** then choose **Next**. In the next page, enter `BasicAgent` as the name for the project, and specify the project location if desired.
4. Choose the **Create** button to create the project.
5. Right-click the project node in **Solution Explorer**, and choose **Properties**. Under **Configuration Properties** > **C/C++** > **Precompiled Headers** > **Precompiled header** choose **Create**.

### To create a C++ console application in Visual Studio 2017 and earlier

1. On the **File** menu, click **New**, and then click **Project** to display the **New Project** dialog box.
2. In the **New Project** dialog box, select the **Visual C++** node in the **Project types** pane and then select **Win32 Console Application** in the **Templates** pane. Type a name for the project, for example, `BasicAgent`, and then click **OK** to display the **Win32 Console Application Wizard**.

3. In the **Win32 Console Application Wizard** dialog box, click **Finish**.

1. In `stdafx.h` (or `pch.h` depending on your version of Visual Studio), add the following code.

```
#include <agents.h>
#include <string>
#include <iostream>
#include <algorithm>
```

The header file `agents.h` contains the functionality of the `concurrency::agent` class.

1. Verify that the application was created successfully by building and running it. To build the application, on the **Build** menu, click **Build Solution**. If the application builds successfully, run the application by clicking **Start Debugging** on the **Debug** menu.

[\[Top\]](#)

## Creating the `file_reader` Class

This section shows how to create the `file_reader` class. The runtime schedules each agent to perform work in its own context. Therefore, you can create an agent that performs work synchronously, but interacts with other components asynchronously. The `file_reader` class reads data from a given input file and sends data from that file to a given target component.

### To create the `file_reader` class

1. Add a new C++ header file to your project. To do so, right-click the **Header Files** node in **Solution Explorer**, click **Add**, and then click **New Item**. In the **Templates** pane, select **Header File (.h)**. In the **Add New Item** dialog box, type `file_reader.h` in the **Name** box and then click **Add**.

2. In `file_reader.h`, add the following code.

```
#pragma once
```

1. In `file_reader.h`, create a class that is named `file_reader` that derives from `agent`.

```
class file_reader : public concurrency::agent
{
public:
protected:
private:
};
```

1. Add the following data members to the `private` section of your class.

```
std::string _file_name;
concurrency::ITarget<std::string>& _target;
concurrency::overwrite_buffer<std::exception> _error;
```

The `_file_name` member is the file name that the agent reads from. The `_target` member is a `concurrency::ITarget` object that the agent writes the contents of the file to. The `_error` member holds any error that occurs during the life of the agent.

1. Add the following code for the `file_reader` constructors to the `public` section of the `file_reader` class.

```

explicit file_reader(const std::string& file_name,
                    concurrency::ITarget<std::string>& target)
    : _file_name(file_name)
    , _target(target)
{
}

explicit file_reader(const std::string& file_name,
                    concurrency::ITarget<std::string>& target,
                    concurrency::Scheduler& scheduler)
    : agent(scheduler)
    , _file_name(file_name)
    , _target(target)
{
}

explicit file_reader(const std::string& file_name,
                    concurrency::ITarget<std::string>& target,
                    concurrency::ScheduleGroup& group)
    : agent(group)
    , _file_name(file_name)
    , _target(target)
{
}

```

Each constructor overload sets the `file_reader` data members. The second and third constructor overload enables your application to use a specific scheduler with your agent. The first overload uses the default scheduler with your agent.

1. Add the `get_error` method to the public section of the `file_reader` class.

```

bool get_error(std::exception& e)
{
    return try_receive(_error, e);
}

```

The `get_error` method retrieves any error that occurs during the life of the agent.

1. Implement the `concurrency::agent::run` method in the `protected` section of your class.

```

void run()
{
    FILE* stream;
    try
    {
        // Open the file.
        if (fopen_s(&stream, _file_name.c_str(), "r") != 0)
        {
            // Throw an exception if an error occurs.
            throw std::exception("Failed to open input file.");
        }

        // Create a buffer to hold file data.
        char buf[1024];

        // Set the buffer size.
        setvbuf(stream, buf, _IOFBF, sizeof buf);

        // Read the contents of the file and send the contents
        // to the target.
        while (fgets(buf, sizeof buf, stream))
        {
            asend(_target, std::string(buf));
        }

        // Send the empty string to the target to indicate the end of processing.
        asend(_target, std::string(""));

        // Close the file.
        fclose(stream);
    }
    catch (const std::exception& e)
    {
        // Send the empty string to the target to indicate the end of processing.
        asend(_target, std::string(""));

        // Write the exception to the error buffer.
        send(_error, e);
    }

    // Set the status of the agent to agent_done.
    done();
}

```

The `run` method opens the file and reads data from it. The `run` method uses exception handling to capture any errors that occur during file processing.

Each time this method reads data from the file, it calls the `concurrency::asend` function to send that data to the target buffer. It sends the empty string to its target buffer to indicate the end of processing.

The following example shows the complete contents of `file_reader.h`.

```

#pragma once

class file_reader : public concurrency::agent
{
public:
    explicit file_reader(const std::string& file_name,
        concurrency::ITarget<std::string>& target)
        : _file_name(file_name)
        , _target(target)
    {
    }

    explicit file_reader(const std::string& file_name,

```

```

        concurrency::ITarget<std::string>& target,
        concurrency::Scheduler& scheduler)
        : agent(scheduler)
        , _file_name(file_name)
        , _target(target)
    {
    }

    explicit file_reader(const std::string& file_name,
        concurrency::ITarget<std::string>& target,
        concurrency::ScheduleGroup& group)
        : agent(group)
        , _file_name(file_name)
        , _target(target)
    {
    }

    // Retrieves any error that occurs during the life of the agent.
    bool get_error(std::exception& e)
    {
        return try_receive(_error, e);
    }

protected:
    void run()
    {
        FILE* stream;
        try
        {
            // Open the file.
            if (fopen_s(&stream, _file_name.c_str(), "r") != 0)
            {
                // Throw an exception if an error occurs.
                throw std::exception("Failed to open input file.");
            }

            // Create a buffer to hold file data.
            char buf[1024];

            // Set the buffer size.
            setvbuf(stream, buf, _IOFBF, sizeof buf);

            // Read the contents of the file and send the contents
            // to the target.
            while (fgets(buf, sizeof buf, stream))
            {
                asend(_target, std::string(buf));
            }

            // Send the empty string to the target to indicate the end of processing.
            asend(_target, std::string(""));

            // Close the file.
            fclose(stream);
        }
        catch (const std::exception& e)
        {
            // Send the empty string to the target to indicate the end of processing.
            asend(_target, std::string(""));

            // Write the exception to the error buffer.
            send(_error, e);
        }

        // Set the status of the agent to agent_done.
        done();
    }

private:

```

```
std::string _file_name;
concurrency::ITarget<std::string>& _target;
concurrency::overwrite_buffer<std::exception> _error;
};
```

[\[Top\]](#)

## Using the file\_reader Class in the Application

This section shows how to use the `file_reader` class to read the contents of a text file. It also shows how to create a `concurrency::call` object that receives this file data and calculates its Adler-32 checksum.

### To use the file\_reader class in your application

1. In BasicAgent.cpp, add the following `#include` statement.

```
#include "file_reader.h"
```

1. In BasicAgent.cpp, add the following `using` directives.

```
using namespace concurrency;
using namespace std;
```

1. In the `_tmain` function, create a `concurrency::event` object that signals the end of processing.

```
event e;
```

1. Create a `call` object that updates the checksum when it receives data.

```
// The components of the Adler-32 sum.
unsigned int a = 1;
unsigned int b = 0;

// A call object that updates the checksum when it receives data.
call<string> calculate_checksum([&] (string s) {
    // If the input string is empty, set the event to signal
    // the end of processing.
    if (s.size() == 0)
        e.set();
    // Perform the Adler-32 checksum algorithm.
    for_each(begin(s), end(s), [&] (char c) {
        a = (a + c) % 65521;
        b = (b + a) % 65521;
    });
});
```

This `call` object also sets the `event` object when it receives the empty string to signal the end of processing.

1. Create a `file_reader` object that reads from the file test.txt and writes the contents of that file to the `call` object.

```
file_reader reader("test.txt", calculate_checksum);
```

1. Start the agent and wait for it to finish.

```
reader.start();
agent::wait(&reader);
```

1. Wait for the `call` object to receive all data and finish.

```
e.wait();
```

1. Check the file reader for errors. If no error occurred, calculate the final Adler-32 sum and print the sum to the console.

```
std::exception error;
if (reader.get_error(error))
{
    wcout << error.what() << endl;
}
else
{
    unsigned int adler32_sum = (b << 16) | a;
    wcout << L"Adler-32 sum is " << hex << adler32_sum << endl;
}
```

The following example shows the complete BasicAgent.cpp file.

```

// BasicAgent.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "file_reader.h"

using namespace concurrency;
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    // An event object that signals the end of processing.
    event e;

    // The components of the Adler-32 sum.
    unsigned int a = 1;
    unsigned int b = 0;

    // A call object that updates the checksum when it receives data.
    call<string> calculate_checksum([&] (string s) {
        // If the input string is empty, set the event to signal
        // the end of processing.
        if (s.size() == 0)
            e.set();
        // Perform the Adler-32 checksum algorithm.
        for_each(begin(s), end(s), [&] (char c) {
            a = (a + c) % 65521;
            b = (b + a) % 65521;
        });
    });

    // Create the agent.
    file_reader reader("test.txt", calculate_checksum);

    // Start the agent and wait for it to complete.
    reader.start();
    agent::wait(&reader);

    // Wait for the call object to receive all data and complete.
    e.wait();

    // Check the file reader for errors.
    // If no error occurred, calculate the final Adler-32 sum and print it
    // to the console.
    std::exception error;
    if (reader.get_error(error))
    {
        wcout << error.what() << endl;
    }
    else
    {
        unsigned int adler32_sum = (b << 16) | a;
        wcout << L"Adler-32 sum is " << hex << adler32_sum << endl;
    }
}

```

[\[Top\]](#)

## Sample Input

This is the sample contents of the input file text.txt:

```
The quick brown fox  
jumps  
over the lazy dog
```

## Sample Output

When used with the sample input, this program produces the following output:

```
Adler-32 sum is fefb0d75
```

## Robust Programming

To prevent concurrent access to data members, we recommend that you add methods that perform work to the `protected` or `private` section of your class. Only add methods that send or receive messages to or from the agent to the `public` section of your class.

Always call the `concurrency::agent::done` method to move your agent to the completed state. You typically call this method before you return from the `run` method.

## Next Steps

For another example of an agent-based application, see [Walkthrough: Using join to Prevent Deadlock](#).

## See also

[Asynchronous Agents Library](#)

[Asynchronous Message Blocks](#)

[Message Passing Functions](#)

[Synchronization Data Structures](#)

[Walkthrough: Using join to Prevent Deadlock](#)

# Walkthrough: Creating a Dataflow Agent

4/25/2019 • 16 minutes to read • [Edit Online](#)

This document demonstrates how to create agent-based applications that are based on dataflow, instead of control flow.

*Control flow* refers to the execution order of operations in a program. Control flow is regulated by using control structures such as conditional statements, loops, and so on. Alternatively, *dataflow* refers to a programming model in which computations are made only when all required data is available. The dataflow programming model is related to the concept of message passing, in which independent components of a program communicate with one another by sending messages.

Asynchronous agents support both the control-flow and dataflow programming models. Although the control-flow model is appropriate in many cases, the dataflow model is appropriate in others, for example, when an agent receives data and performs an action that is based on the payload of that data.

## Prerequisites

Read the following documents before you start this walkthrough:

- [Asynchronous Agents](#)
- [Asynchronous Message Blocks](#)
- [How to: Use a Message Block Filter](#)

## Sections

This walkthrough contains the following sections:

- [Creating a Basic Control-Flow Agent](#)
- [Creating a Basic Dataflow Agent](#)
- [Creating a Message-Logging Agent](#)

## Creating a Basic Control-Flow Agent

Consider the following example that defines the `control_flow_agent` class. The `control_flow_agent` class acts on three message buffers: one input buffer and two output buffers. The `run` method reads from the source message buffer in a loop and uses a conditional statement to direct the flow of program execution. The agent increments one counter for non-zero, negative values and increments another counter for non-zero, positive values. After the agent receives the sentinel value of zero, it sends the values of the counters to the output message buffers. The `negatives` and `positives` methods enable the application to read the counts of negative and positive values from the agent.

```

// A basic agent that uses control-flow to regulate the order of program
// execution. This agent reads numbers from a message buffer and counts the
// number of positive and negative values.
class control_flow_agent : public agent
{
public:
    explicit control_flow_agent(ISource<int>& source)
        : _source(source)
    {
    }

    // Retrieves the count of negative numbers that the agent received.
    size_t negatives()
    {
        return receive(_negatives);
    }

    // Retrieves the count of positive numbers that the agent received.
    size_t positives()
    {
        return receive(_positives);
    }

protected:
    void run()
    {
        // Counts the number of negative and positive values that
        // the agent receives.
        size_t negative_count = 0;
        size_t positive_count = 0;

        // Read from the source buffer until we receive
        // the sentinel value of 0.
        int value = 0;
        while ((value = receive(_source)) != 0)
        {
            // Send negative values to the first target and
            // non-negative values to the second target.
            if (value < 0)
                ++negative_count;
            else
                ++positive_count;
        }

        // Write the counts to the message buffers.
        send(_negatives, negative_count);
        send(_positives, positive_count);

        // Set the agent to the completed state.
        done();
    }

private:
    // Source message buffer to read from.
    ISource<int>& _source;

    // Holds the number of negative and positive numbers that the agent receives.
    single_assignment<size_t> _negatives;
    single_assignment<size_t> _positives;
};

```

Although this example makes basic use of control flow in an agent, it demonstrates the serial nature of control-flow-based programming. Each message must be processed sequentially, even though multiple messages might be available in the input message buffer. The dataflow model enables both branches of the conditional statement to evaluate concurrently. The dataflow model also enables you to create more complex messaging networks that act on data as it becomes available.

## Creating a Basic Dataflow Agent

This section shows how to convert the `control_flow_agent` class to use the dataflow model to perform the same task.

The dataflow agent works by creating a network of message buffers, each of which serves a specific purpose. Certain message blocks use a filter function to accept or reject a message on the basis of its payload. A filter function ensures that a message block receives only certain values.

### To convert the control-flow agent to a dataflow agent

1. Copy the body of the `control_flow_agent` class to another class, for example, `dataflow_agent`.  
Alternatively, you can rename the `control_flow_agent` class.
2. Remove the body of the loop that calls `receive` from the `run` method.

```
void run()
{
    // Counts the number of negative and positive values that
    // the agent receives.
    size_t negative_count = 0;
    size_t positive_count = 0;

    // Write the counts to the message buffers.
    send(_negatives, negative_count);
    send(_positives, positive_count);

    // Set the agent to the completed state.
    done();
}
```

1. In the `run` method, after the initialization of the variables `negative_count` and `positive_count`, add a `countdown_event` object that tracks the count of active operations.

```
// Tracks the count of active operations.
countdown_event active;
// An event that is set by the sentinel.
event received_sentinel;
```

The `countdown_event` class is shown later in this topic.

1. Create the message buffer objects that will participate in the dataflow network.

```

//
// Create the members of the dataflow network.
//

// Increments the active counter.
transformer<int, int> increment_active(
    [&active](int value) -> int {
        active.add_count();
        return value;
    });

// Increments the count of negative values.
call<int> negatives(
    [&](int value) {
        ++negative_count;
        // Decrement the active counter.
        active.signal();
    },
    [](int value) -> bool {
        return value < 0;
    });

// Increments the count of positive values.
call<int> positives(
    [&](int value) {
        ++positive_count;
        // Decrement the active counter.
        active.signal();
    },
    [](int value) -> bool {
        return value > 0;
    });

// Receives only the sentinel value of 0.
call<int> sentinel(
    [&](int value) {
        // Decrement the active counter.
        active.signal();
        // Set the sentinel event.
        received_sentinel.set();
    },
    [](int value) -> bool {
        return value == 0;
    });

// Connects the _source message buffer to the rest of the network.
unbounded_buffer<int> connector;

```

## 1. Connect the message buffers to form a network.

```

//
// Connect the network.
//

// Connect the internal nodes of the network.
connector.link_target(&negatives);
connector.link_target(&positives);
connector.link_target(&sentinel);
increment_active.link_target(&connector);

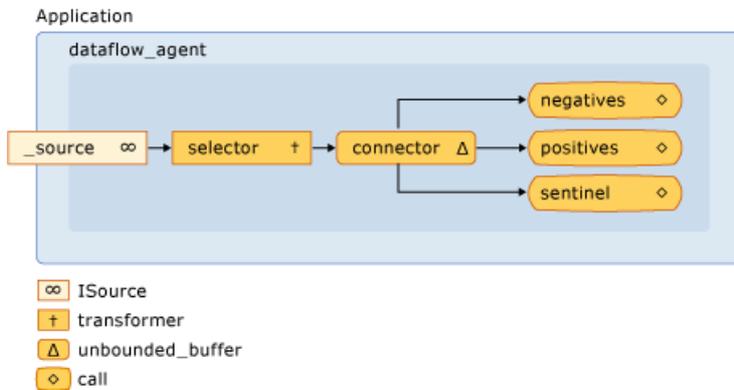
// Connect the _source buffer to the internal network to
// begin data flow.
_source.link_target(&increment_active);

```

1. Wait for the `event` and `countdown_event` objects to be set. These events signal that the agent has received the sentinel value and that all operations have finished.

```
// Wait for the sentinel event and for all operations to finish.
received_sentinel.wait();
active.wait();
```

The following diagram shows the complete dataflow network for the `dataflow_agent` class:



The following table describes the members of the network.

MEMBER	DESCRIPTION
<code>increment_active</code>	A <code>concurrency::transformer</code> object that increments the active event counter and passes the input value to the rest of the network.
<code>negatives</code> , <code>positives</code>	<code>concurrency::call</code> objects that increment the count of numbers and decrements the active event counter. The objects each use a filter to accept either negative numbers or positive numbers.
<code>sentinel</code>	A <code>concurrency::call</code> object that accepts only the sentinel value of zero and decrements the active event counter.
<code>connector</code>	A <code>concurrency::unbounded_buffer</code> object that connects the source message buffer to the internal network.

Because the `run` method is called on a separate thread, other threads can send messages to the network before the network is fully connected. The `_source` data member is an `unbounded_buffer` object that buffers all input that is sent from the application to the agent. To make sure that the network processes all input messages, the agent first links the internal nodes of the network and then links the start of that network, `connector`, to the `_source` data member. This guarantees that messages do not get processed as the network is being formed.

Because the network in this example is based on dataflow, rather than on control-flow, the network must communicate to the agent that it has finished processing each input value and that the sentinel node has received its value. This example uses a `countdown_event` object to signal that all input values have been processed and a `concurrency::event` object to indicate that the sentinel node has received its value. The `countdown_event` class uses an `event` object to signal when a counter value reaches zero. The head of the dataflow network increments the counter every time that it receives a value. Every terminal node of the network decrements the counter after it processes the input value. After the agent forms the dataflow network, it waits for the sentinel node to set the `event` object and for the `countdown_event` object to signal that its counter has reached zero.

The following example shows the `control_flow_agent`, `dataflow_agent`, and `countdown_event` classes. The `wmain`

function creates a `control_flow_agent` and a `dataflow_agent` object and uses the `send_values` function to send a series of random values to the agents.

```
// dataflow-agent.cpp
// compile with: /EHsc
#include <windows.h>
#include <agents.h>
#include <iostream>
#include <random>

using namespace concurrency;
using namespace std;

// A basic agent that uses control-flow to regulate the order of program
// execution. This agent reads numbers from a message buffer and counts the
// number of positive and negative values.
class control_flow_agent : public agent
{
public:
    explicit control_flow_agent(ISource<int>& source)
        : _source(source)
    {
    }

    // Retrieves the count of negative numbers that the agent received.
    size_t negatives()
    {
        return receive(_negatives);
    }

    // Retrieves the count of positive numbers that the agent received.
    size_t positives()
    {
        return receive(_positives);
    }

protected:
    void run()
    {
        // Counts the number of negative and positive values that
        // the agent receives.
        size_t negative_count = 0;
        size_t positive_count = 0;

        // Read from the source buffer until we receive
        // the sentinel value of 0.
        int value = 0;
        while ((value = receive(_source)) != 0)
        {
            // Send negative values to the first target and
            // non-negative values to the second target.
            if (value < 0)
                ++negative_count;
            else
                ++positive_count;
        }

        // Write the counts to the message buffers.
        send(_negatives, negative_count);
        send(_positives, positive_count);

        // Set the agent to the completed state.
        done();
    }

private:
    // Source message buffer to read from.
    ISource<int>& _source;
};
```

```

    // Holds the number of negative and positive numbers that the agent receives.
    single_assignment<size_t> _negatives;
    single_assignment<size_t> _positives;
};

// A synchronization primitive that is signaled when its
// count reaches zero.
class countdown_event
{
public:
    countdown_event(unsigned int count = 0L)
        : _current(static_cast<long>(count))
    {
        // Set the event if the initial count is zero.
        if (_current == 0L)
            _event.set();
    }

    // Decrements the event counter.
    void signal() {
        if(InterlockedDecrement(&_current) == 0L) {
            _event.set();
        }
    }

    // Increments the event counter.
    void add_count() {
        if(InterlockedIncrement(&_current) == 1L) {
            _event.reset();
        }
    }

    // Blocks the current context until the event is set.
    void wait() {
        _event.wait();
    }

private:
    // The current count.
    volatile long _current;
    // The event that is set when the counter reaches zero.
    event _event;

    // Disable copy constructor.
    countdown_event(const countdown_event&);
    // Disable assignment.
    countdown_event const & operator=(countdown_event const&);
};

// A basic agent that resembles control_flow_agent, but uses uses dataflow to
// perform computations when data becomes available.
class dataflow_agent : public agent
{
public:
    dataflow_agent(ISource<int>& source)
        : _source(source)
    {
    }

    // Retrieves the count of negative numbers that the agent received.
    size_t negatives()
    {
        return receive(_negatives);
    }

    // Retrieves the count of positive numbers that the agent received.
    size_t positives()
    {

```

```

    return receive(_positives);
}

protected:
void run()
{
    // Counts the number of negative and positive values that
    // the agent receives.
    size_t negative_count = 0;
    size_t positive_count = 0;

    // Tracks the count of active operations.
    countdown_event active;
    // An event that is set by the sentinel.
    event received_sentinel;

    //
    // Create the members of the dataflow network.
    //

    // Increments the active counter.
    transformer<int, int> increment_active(
        [&active](int value) -> int {
            active.add_count();
            return value;
        });

    // Increments the count of negative values.
    call<int> negatives(
        [&](int value) {
            ++negative_count;
            // Decrement the active counter.
            active.signal();
        },
        [](int value) -> bool {
            return value < 0;
        });

    // Increments the count of positive values.
    call<int> positives(
        [&](int value) {
            ++positive_count;
            // Decrement the active counter.
            active.signal();
        },
        [](int value) -> bool {
            return value > 0;
        });

    // Receives only the sentinel value of 0.
    call<int> sentinel(
        [&](int value) {
            // Decrement the active counter.
            active.signal();
            // Set the sentinel event.
            received_sentinel.set();
        },
        [](int value) -> bool {
            return value == 0;
        });

    // Connects the _source message buffer to the rest of the network.
    unbounded_buffer<int> connector;

    //
    // Connect the network.
    //

    // Connect the internal nodes of the network.

```

```

connector.link_target(&negatives);
connector.link_target(&positives);
connector.link_target(&sentinel);
increment_active.link_target(&connector);

// Connect the _source buffer to the internal network to
// begin data flow.
_source.link_target(&increment_active);

// Wait for the sentinel event and for all operations to finish.
received_sentinel.wait();
active.wait();

// Write the counts to the message buffers.
send(_negatives, negative_count);
send(_positives, positive_count);

// Set the agent to the completed state.
done();
}

private:
// Source message buffer to read from.
ISource<int>& _source;

// Holds the number of negative and positive numbers that the agent receives.
single_assignment<size_t> _negatives;
single_assignment<size_t> _positives;
};

// Sends a number of random values to the provided message buffer.
void send_values(ITarget<int>& source, int sentinel, size_t count)
{
// Send a series of random numbers to the source buffer.
mt19937 rnd(42);
for (size_t i = 0; i < count; ++i)
{
// Generate a random number that is not equal to the sentinel value.
int n;
while ((n = rnd()) == sentinel);

send(source, n);
}
// Send the sentinel value.
send(source, sentinel);
}

int wmain()
{
// Signals to the agent that there are no more values to process.
const int sentinel = 0;
// The number of samples to send to each agent.
const size_t count = 1000000;

// The source buffer that the application writes numbers to and
// the agents read numbers from.
unbounded_buffer<int> source;

//
// Use a control-flow agent to process a series of random numbers.
//
wcout << L"Control-flow agent:" << endl;

// Create and start the agent.
control_flow_agent cf_agent(source);
cf_agent.start();

// Send values to the agent.
send_values(source, sentinel, count);

```

```

// Wait for the agent to finish.
agent::wait(&cf_agent);

// Print the count of negative and positive numbers.
wcout << L"There are " << cf_agent.negatives()
    << L" negative numbers."<< endl;
wcout << L"There are " << cf_agent.positives()
    << L" positive numbers."<< endl;

//
// Perform the same task, but this time with a dataflow agent.
//
wcout << L"Dataflow agent:" << endl;

// Create and start the agent.
dataflow_agent df_agent(source);
df_agent.start();

// Send values to the agent.
send_values(source, sentinel, count);

// Wait for the agent to finish.
agent::wait(&df_agent);

// Print the count of negative and positive numbers.
wcout << L"There are " << df_agent.negatives()
    << L" negative numbers."<< endl;
wcout << L"There are " << df_agent.positives()
    << L" positive numbers."<< endl;
}

```

This example produces the following sample output:

```

Control-flow agent:
There are 500523 negative numbers.
There are 499477 positive numbers.
Dataflow agent:
There are 500523 negative numbers.
There are 499477 positive numbers.

```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `dataflow-agent.cpp` and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc dataflow-agent.cpp**

[\[Top\]](#)

## Creating a Message-Logging Agent

The following example shows the `log_agent` class, which resembles the `dataflow_agent` class. The `log_agent` class implements an asynchronous logging agent that writes log messages to a file and to the console. The `log_agent` class enables the application to categorize messages as informational, warning, or error. It also enables the application to specify whether each log category is written to a file, the console, or both. This example writes all log messages to a file and only error messages to the console.

```

// log-filter.cpp
// compile with: /EHsc
#include <windows.h>
#include <agents.h>
#include <sstream>

```

```

#include <fstream>
#include <iostream>

using namespace concurrency;
using namespace std;

// A synchronization primitive that is signaled when its
// count reaches zero.
class countdown_event
{
public:
    countdown_event(unsigned int count = 0L)
        : _current(static_cast<long>(count))
    {
        // Set the event if the initial count is zero.
        if (_current == 0L)
        {
            _event.set();
        }
    }

    // Decrements the event counter.
    void signal()
    {
        if(InterlockedDecrement(&_current) == 0L)
        {
            _event.set();
        }
    }

    // Increments the event counter.
    void add_count()
    {
        if(InterlockedIncrement(&_current) == 1L)
        {
            _event.reset();
        }
    }

    // Blocks the current context until the event is set.
    void wait()
    {
        _event.wait();
    }

private:
    // The current count.
    volatile long _current;
    // The event that is set when the counter reaches zero.
    event _event;

    // Disable copy constructor.
    countdown_event(const countdown_event&);
    // Disable assignment.
    countdown_event const & operator=(countdown_event const&);
};

// Defines message types for the logger.
enum log_message_type
{
    log_info     = 0x1,
    log_warning  = 0x2,
    log_error    = 0x4,
};

// An asynchronous logging agent that writes log messages to
// file and to the console.
class log_agent : public agent
{

```

```

    // Holds a message string and its logging type.
    struct log_message
    {
        wstring message;
        log_message_type type;
    };

public:
    log_agent(const wstring& file_path, log_message_type file_messages, log_message_type console_messages)
        : _file(file_path)
        , _file_messages(file_messages)
        , _console_messages(console_messages)
        , _active(0)
    {
        if (_file.bad())
        {
            throw invalid_argument("Unable to open log file.");
        }
    }

    // Writes the provided message to the log.
    void log(const wstring& message, log_message_type type)
    {
        // Increment the active message count.
        _active.add_count();

        // Send the message to the network.
        log_message msg = { message, type };
        send(_log_buffer, msg);
    }

    void close()
    {
        // Signal that the agent is now closed.
        _closed.set();
    }

protected:

    void run()
    {
        //
        // Create the dataflow network.
        //

        // Writes a log message to file.
        call<log_message> writer([this](log_message msg)
        {
            if ((msg.type & _file_messages) != 0)
            {
                // Write the message to the file.
                write_to_stream(msg, _file);
            }
            if ((msg.type & _console_messages) != 0)
            {
                // Write the message to the console.
                write_to_stream(msg, wcout);
            }
            // Decrement the active counter.
            _active.signal();
        });

        // Connect _log_buffer to the internal network to begin data flow.
        _log_buffer.link_target(&writer);

        // Wait for the closed event to be signaled.
        _closed.wait();

        // Wait for all messages to be processed

```

```

    // wait for all messages to be processed.
    _active.wait();

    // Close the log file and flush the console.
    _file.close();
    wcout.flush();

    // Set the agent to the completed state.
    done();
}

private:
    // Writes a logging message to the specified output stream.
    void write_to_stream(const log_message& msg, wostream& stream)
    {
        // Write the message to the stream.
        wstringstream ss;

        switch (msg.type)
        {
            case log_info:
                ss << L"info: ";
                break;
            case log_warning:
                ss << L"warning: ";
                break;
            case log_error:
                ss << L"error: ";
        }

        ss << msg.message << endl;
        stream << ss.str();
    }

private:
    // The file stream to write messages to.
    wofstream _file;

    // The log message types that are written to file.
    log_message_type _file_messages;

    // The log message types that are written to the console.
    log_message_type _console_messages;

    // The head of the network. Propagates logging messages
    // to the rest of the network.
    unbounded_buffer<log_message> _log_buffer;

    // Counts the number of active messages in the network.
    countdown_event _active;

    // Signals that the agent has been closed.
    event _closed;
};

int wmain()
{
    // Union of all log message types.
    log_message_type log_all = log_message_type(log_info | log_warning | log_error);

    // Create a logging agent that writes all log messages to file and error
    // messages to the console.
    log_agent logger(L"log.txt", log_all, log_error);

    // Start the agent.
    logger.start();

    // Log a few messages.

```

```
logger.log(L"===Logging started.===", log_info);

logger.log(L"This is a sample warning message.", log_warning);
logger.log(L"This is a sample error message.", log_error);

logger.log(L"===Logging finished.===", log_info);

// Close the logger and wait for the agent to finish.
logger.close();
agent::wait(&logger);
}
```

This example writes the following output to the console.

```
error: This is a sample error message.
```

This example also produces the log.txt file, which contains the following text.

```
info: ===Logging started.===
warning: This is a sample warning message.
error: This is a sample error message.
info: ===Logging finished.===
```

### Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `log-filter.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc log-filter.cpp
```

[\[Top\]](#)

## See also

[Concurrency Runtime Walkthroughs](#)

# Walkthrough: Creating an Image-Processing Network

4/25/2019 • 19 minutes to read • [Edit Online](#)

This document demonstrates how to create a network of asynchronous message blocks that perform image processing.

The network determines which operations to perform on an image on the basis of its characteristics. This example uses the *dataflow* model to route images through the network. In the dataflow model, independent components of a program communicate with one another by sending messages. When a component receives a message, it can perform some action and then pass the result of that action to another component. Compare this with the *control flow* model, in which an application uses control structures, for example, conditional statements, loops, and so on, to control the order of operations in a program.

A network that is based on dataflow creates a *pipeline* of tasks. Each stage of the pipeline concurrently performs part of the overall task. An analogy to this is an assembly line for automobile manufacturing. As each vehicle passes through the assembly line, one station assembles the frame, another installs the engine, and so on. By enabling multiple vehicles to be assembled simultaneously, the assembly line provides better throughput than assembling complete vehicles one at a time.

## Prerequisites

Read the following documents before you start this walkthrough:

- [Asynchronous Message Blocks](#)
- [How to: Use a Message Block Filter](#)
- [Walkthrough: Creating a Dataflow Agent](#)

We also recommend that you understand the basics of GDI+ before you start this walkthrough.

## Sections

This walkthrough contains the following sections:

- [Defining Image Processing Functionality](#)
- [Creating the Image Processing Network](#)
- [The Complete Example](#)

## Defining Image Processing Functionality

This section shows the support functions that the image processing network uses to work with images that are read from disk.

The following functions, `GetRGB` and `MakeColor`, extract and combine the individual components of the given color, respectively.

```

// Retrieves the red, green, and blue components from the given
// color value.
void GetRGB(DWORD color, BYTE& r, BYTE& g, BYTE& b)
{
    r = static_cast<BYTE>((color & 0x00ff0000) >> 16);
    g = static_cast<BYTE>((color & 0x0000ff00) >> 8);
    b = static_cast<BYTE>((color & 0x000000ff));
}

// Creates a single color value from the provided red, green,
// and blue components.
DWORD MakeColor(BYTE r, BYTE g, BYTE b)
{
    return (r<<16) | (g<<8) | (b);
}

```

The following function, `ProcessImage`, calls the given `std::function` object to transform the color value of each pixel in a GDI+ `Bitmap` object. The `ProcessImage` function uses the `concurrency::parallel_for` algorithm to process each row of the bitmap in parallel.

```

// Calls the provided function for each pixel in a Bitmap object.
void ProcessImage(Bitmap* bmp, const function<void (DWORD*)>& f)
{
    int width = bmp->GetWidth();
    int height = bmp->GetHeight();

    // Lock the bitmap.
    BitmapData bitmapData;
    Rect rect(0, 0, bmp->GetWidth(), bmp->GetHeight());
    bmp->LockBits(&rect, ImageLockModeWrite, PixelFormat32bppRGB, &bitmapData);

    // Get a pointer to the bitmap data.
    DWORD* image_bits = (DWORD*)bitmapData.Scan0;

    // Call the function for each pixel in the image.
    parallel_for (0, height, [&, width](int y)
    {
        for (int x = 0; x < width; ++x)
        {
            // Get the current pixel value.
            DWORD* curr_pixel = image_bits + (y * width) + x;

            // Call the function.
            f(*curr_pixel);
        }
    });

    // Unlock the bitmap.
    bmp->UnlockBits(&bitmapData);
}

```

The following functions, `Grayscale`, `Sepiatone`, `ColorMask`, and `Darken`, call the `ProcessImage` function to transform the color value of each pixel in a `Bitmap` object. Each of these functions uses a lambda expression to define the color transformation of one pixel.

```

// Converts the given image to grayscale.
Bitmap* Grayscale(Bitmap* bmp)
{
    ProcessImage(bmp,
        [(DWORD& color) {
            BYTE r, g, b;
            GetRGB(color, r, g, b);

            // Set each color component to the average of
            // the original components.
            BYTE c = (static_cast<WORD>(r) + g + b) / 3;
            color = MakeColor(c, c, c);
        }
    );
    return bmp;
}

// Applies sepia toning to the provided image.
Bitmap* Sepiatone(Bitmap* bmp)
{
    ProcessImage(bmp,
        [(DWORD& color) {
            BYTE r0, g0, b0;
            GetRGB(color, r0, g0, b0);

            WORD r1 = static_cast<WORD>((r0 * .393) + (g0 * .769) + (b0 * .189));
            WORD g1 = static_cast<WORD>((r0 * .349) + (g0 * .686) + (b0 * .168));
            WORD b1 = static_cast<WORD>((r0 * .272) + (g0 * .534) + (b0 * .131));

            color = MakeColor(min(0xff, r1), min(0xff, g1), min(0xff, b1));
        }
    );
    return bmp;
}

// Applies the given color mask to each pixel in the provided image.
Bitmap* ColorMask(Bitmap* bmp, DWORD mask)
{
    ProcessImage(bmp,
        [mask](DWORD& color) {
            color = color & mask;
        }
    );
    return bmp;
}

// Darkens the provided image by the given amount.
Bitmap* Darken(Bitmap* bmp, unsigned int percent)
{
    if (percent > 100)
        throw invalid_argument("Darken: percent must less than 100.");

    double factor = percent / 100.0;

    ProcessImage(bmp,
        [factor](DWORD& color) {
            BYTE r, g, b;
            GetRGB(color, r, g, b);
            r = static_cast<BYTE>(factor*r);
            g = static_cast<BYTE>(factor*g);
            b = static_cast<BYTE>(factor*b);
            color = MakeColor(r, g, b);
        }
    );
    return bmp;
}

```

The following function, `GetColorDominance`, also calls the `ProcessImage` function. However, instead of changing the value of each color, this function uses `concurrency::combinable` objects to compute whether the red, green, or blue color component dominates the image.

```
// Determines which color component (red, green, or blue) is most dominant
// in the given image and returns a corresponding color mask.
DWORD GetColorDominance(Bitmap* bmp)
{
    // The ProcessImage function processes the image in parallel.
    // The following combinable objects enable the callback function
    // to increment the color counts without using a lock.
    combinable<unsigned int> reds;
    combinable<unsigned int> greens;
    combinable<unsigned int> blues;

    ProcessImage(bmp,
        [&](DWORD& color) {
            BYTE r, g, b;
            GetRGB(color, r, g, b);
            if (r >= g && r >= b)
                reds.local()++;
            else if (g >= r && g >= b)
                greens.local()++;
            else
                blues.local()++;
        }
    );

    // Determine which color is dominant and return the corresponding
    // color mask.

    unsigned int r = reds.combine(plus<unsigned int>());
    unsigned int g = greens.combine(plus<unsigned int>());
    unsigned int b = blues.combine(plus<unsigned int>());

    if (r + r >= g + b)
        return 0x00ff0000;
    else if (g + g >= r + b)
        return 0x0000ff00;
    else
        return 0x000000ff;
}
```

The following function, `GetEncoderClsid`, retrieves the class identifier for the given MIME type of an encoder. The application uses this function to retrieve the encoder for a bitmap.

```

// Retrieves the class identifier for the given MIME type of an encoder.
int GetEncoderClsid(const WCHAR* format, CLSID* pClsid)
{
    UINT  num = 0;          // number of image encoders
    UINT  size = 0;        // size of the image encoder array in bytes

    ImageCodecInfo* pImageCodecInfo = nullptr;

    GetImageEncodersSize(&num, &size);
    if(size == 0)
        return -1; // Failure

    pImageCodecInfo = (ImageCodecInfo*)(malloc(size));
    if(pImageCodecInfo == nullptr)
        return -1; // Failure

    GetImageEncoders(num, size, pImageCodecInfo);

    for(UINT j = 0; j < num; ++j)
    {
        if( wcsncmp(pImageCodecInfo[j].MimeType, format) == 0 )
        {
            *pClsid = pImageCodecInfo[j].Clsid;
            free(pImageCodecInfo);
            return j; // Success
        }
    }

    free(pImageCodecInfo);
    return -1; // Failure
}

```

[\[Top\]](#)

## Creating the Image Processing Network

This section describes how to create a network of asynchronous message blocks that perform image processing on every JPEG (.jpg) image in a given directory. The network performs the following image-processing operations:

1. For any image that is authored by Tom, convert to grayscale.
2. For any image that has red as the dominant color, remove the green and blue components and then darken it.
3. For any other image, apply sepia toning.

The network applies only the first image-processing operation that matches one of these conditions. For example, if an image is authored by Tom and has red as its dominant color, the image is only converted to grayscale.

After the network performs each image-processing operation, it saves the image to disk as a bitmap (.bmp) file.

The following steps show how to create a function that implements this image processing network and applies that network to every JPEG image in a given directory.

### To create the image processing network

1. Create a function, `ProcessImages`, that takes the name of a directory on disk.

```
void ProcessImages(const wstring& directory)
{
}
```

2. In the `ProcessImages` function, create a `countdown_event` variable. The `countdown_event` class is shown later in this walkthrough.

```
// Holds the number of active image processing operations and
// signals to the main thread that processing is complete.
countdown_event active(0);
```

3. Create a `std::map` object that associates a `Bitmap` object with its original file name.

```
// Maps Bitmap objects to their original file names.
map<Bitmap*, wstring> bitmap_file_names;
```

4. Add the following code to define the members of the image-processing network.

```
//
// Create the nodes of the network.
//

// Loads Bitmap objects from disk.
transformer<wstring, Bitmap*> load_bitmap(
    [&](wstring file_name) -> Bitmap* {
        Bitmap* bmp = new Bitmap(file_name.c_str());
        if (bmp != nullptr)
            bitmap_file_names.insert(make_pair(bmp, file_name));
        return bmp;
    }
);

// Holds loaded Bitmap objects.
unbounded_buffer<Bitmap*> loaded_bitmaps;

// Converts images that are authored by Tom to grayscale.
transformer<Bitmap*, Bitmap*> grayscale(
    [](Bitmap* bmp) {
        return Grayscale(bmp);
    },
    nullptr,
    [](Bitmap* bmp) -> bool {
        if (bmp == nullptr)
            return false;

        // Retrieve the artist name from metadata.
        UINT size = bmp->GetPropertyItemSize(PropertyTagArtist);
        if (size == 0)
            // Image does not have the Artist property.
            return false;

        PropertyItem* artistProperty = (PropertyItem*) malloc(size);
        bmp->GetPropertyItem(PropertyTagArtist, size, artistProperty);
        string artist(reinterpret_cast<char*>(artistProperty->value));
        free(artistProperty);

        return (artist.find("Tom ") == 0);
    }
);

// Removes the green and blue color components from images that have red as
// their dominant color.
```

```

transformer<Bitmap*, Bitmap*> colormask(
    [](Bitmap* bmp) {
        return ColorMask(bmp, 0x00ff0000);
    },
    nullptr,
    [](Bitmap* bmp) -> bool {
        if (bmp == nullptr)
            return false;
        return (GetColorDominance(bmp) == 0x00ff0000);
    }
);

// Darkens the color of the provided Bitmap object.
transformer<Bitmap*, Bitmap*> darken([](Bitmap* bmp) {
    return Darken(bmp, 50);
});

// Applies sepia toning to the remaining images.
transformer<Bitmap*, Bitmap*> sepiatone(
    [](Bitmap* bmp) {
        return Sepiatone(bmp);
    },
    nullptr,
    [](Bitmap* bmp) -> bool { return bmp != nullptr; }
);

// Saves Bitmap objects to disk.
transformer<Bitmap*, Bitmap*> save_bitmap([&](Bitmap* bmp) -> Bitmap* {
    // Replace the file extension with .bmp.
    wstring file_name = bitmap_file_names[bmp];
    file_name.replace(file_name.rfind(L'.' ) + 1, 3, L"bmp");

    // Save the processed image.
    CLSID bmpClsid;
    GetEncoderClsid(L"image/bmp", &bmpClsid);
    bmp->Save(file_name.c_str(), &bmpClsid);

    return bmp;
});

// Deletes Bitmap objects.
transformer<Bitmap*, Bitmap*> delete_bitmap([](Bitmap* bmp) -> Bitmap* {
    delete bmp;
    return nullptr;
});

// Decrements the event counter.
call<Bitmap*> decrement([&](Bitmap* _) {
    active.signal();
});

```

5. Add the following code to connect the network.

```

//
// Connect the network.
//

load_bitmap.link_target(&loaded_bitmaps);

loaded_bitmaps.link_target(&grayscale);
loaded_bitmaps.link_target(&colormask);
colormask.link_target(&darken);
loaded_bitmaps.link_target(&sepiatone);
loaded_bitmaps.link_target(&decrement);

grayscale.link_target(&save_bitmap);
darken.link_target(&save_bitmap);
sepiatone.link_target(&save_bitmap);

save_bitmap.link_target(&delete_bitmap);
delete_bitmap.link_target(&decrement);

```

6. Add the following code to send to the head of the network the full path of each JPEG file in the directory.

```

// Traverse all files in the directory.
wstring searchPattern = directory;
searchPattern.append(L"\\*");

WIN32_FIND_DATA fileFindData;
HANDLE hFind = FindFirstFile(searchPattern.c_str(), &fileFindData);
if (hFind == INVALID_HANDLE_VALUE)
    return;
do
{
    if (!(fileFindData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY))
    {
        wstring file = fileFindData.cFileName;

        // Process only JPEG files.
        if (file.rfind(L".jpg") == file.length() - 4)
        {
            // Form the full path to the file.
            wstring full_path(directory);
            full_path.append(L"\\");
            full_path.append(file);

            // Increment the count of work items.
            active.add_count();

            // Send the path name to the network.
            send(load_bitmap, full_path);
        }
    }
}
while (FindNextFile(hFind, &fileFindData) != 0);
FindClose(hFind);

```

7. Wait for the `countdown_event` variable to reach zero.

```

// Wait for all operations to finish.
active.wait();

```

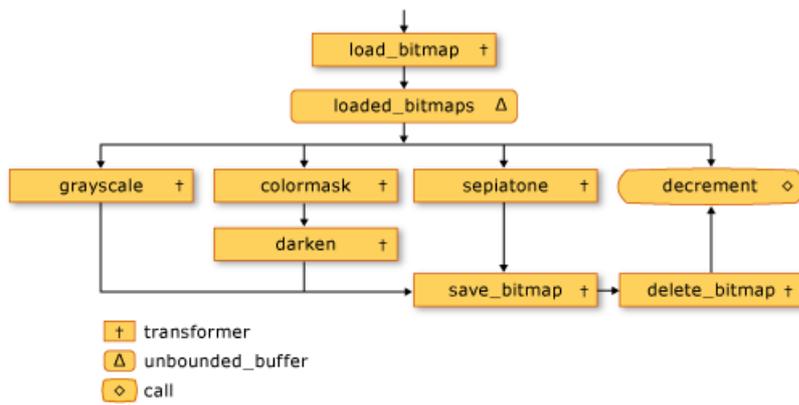
The following table describes the members of the network.

MEMBER	DESCRIPTION
<code>load_bitmap</code>	A <code>concurrency::transformer</code> object that loads a <code>Bitmap</code> object from disk and adds an entry to the <code>map</code> object to associate the image with its original file name.
<code>loaded_bitmaps</code>	A <code>concurrency::unbounded_buffer</code> object that sends the loaded images to the image processing filters.
<code>grayscale</code>	A <code>transformer</code> object that converts images that are authored by Tom to grayscale. It uses the metadata of the image to determine its author.
<code>colormask</code>	A <code>transformer</code> object that removes the green and blue color components from images that have red as the dominant color.
<code>darken</code>	A <code>transformer</code> object that darkens images that have red as the dominant color.
<code>sepiatone</code>	A <code>transformer</code> object that applies sepia toning to images that are not authored by Tom and are not predominantly red.
<code>save_bitmap</code>	A <code>transformer</code> object that saves the processed <code>image</code> to disk as a bitmap. <code>save_bitmap</code> retrieves the original file name from the <code>map</code> object and changes its file name extension to .bmp.
<code>delete_bitmap</code>	A <code>transformer</code> object that frees the memory for the images.
<code>decrement</code>	A <code>concurrency::call</code> object that acts as the terminal node in the network. It decrements the <code>countdown_event</code> object to signal to the main application that an image has been processed.

The `loaded_bitmaps` message buffer is important because, as an `unbounded_buffer` object, it offers `Bitmap` objects to multiple receivers. When a target block accepts a `Bitmap` object, the `unbounded_buffer` object does not offer that `Bitmap` object to any other targets. Therefore, the order in which you link objects to an `unbounded_buffer` object is important. The `grayscale`, `colormask`, and `sepiatone` message blocks each use a filter to accept only certain `Bitmap` objects. The `decrement` message buffer is an important target of the `loaded_bitmaps` message buffer because it accepts all `Bitmap` objects that are rejected by the other message buffers. An `unbounded_buffer` object is required to propagate messages in order. Therefore, an `unbounded_buffer` object blocks until a new target block is linked to it and accepts the message if no current target block accepts that message.

If your application requires that multiple message blocks process the message, instead of just the one message block that first accepts the message, you can use another message block type, such as `overwrite_buffer`. The `overwrite_buffer` class holds one message at a time, but it propagates that message to each of its targets.

The following illustration shows the image processing network:



The `countdown_event` object in this example enables the image processing network to inform the main application when all images have been processed. The `countdown_event` class uses a `concurrency::event` object to signal when a counter value reaches zero. The main application increments the counter every time that it sends a file name to the network. The terminal node of the network decrements the counter after each image has been processed. After the main application traverses the specified directory, it waits for the `countdown_event` object to signal that its counter has reached zero.

The following example shows the `countdown_event` class:

```

// A synchronization primitive that is signaled when its
// count reaches zero.
class countdown_event
{
public:
    countdown_event(unsigned int count = 0)
        : _current(static_cast<long>(count))
    {
        // Set the event if the initial count is zero.
        if (_current == 0L)
            _event.set();
    }

    // Decrements the event counter.
    void signal() {
        if(InterlockedDecrement(&_current) == 0L) {
            _event.set();
        }
    }

    // Increments the event counter.
    void add_count() {
        if(InterlockedIncrement(&_current) == 1L) {
            _event.reset();
        }
    }

    // Blocks the current context until the event is set.
    void wait() {
        _event.wait();
    }

private:
    // The current count.
    volatile long _current;
    // The event that is set when the counter reaches zero.
    event _event;

    // Disable copy constructor.
    countdown_event(const countdown_event&);
    // Disable assignment.
    countdown_event const & operator=(countdown_event const&);
};

```

[\[Top\]](#)

## The Complete Example

The following code shows the complete example. The `wmain` function manages the GDI+ library and calls the `ProcessImages` function to process the JPEG files in the `Sample Pictures` directory.

```

// image-processing-network.cpp
// compile with: /DUNICODE /EHsc image-processing-network.cpp /link gdiplus.lib
#include <windows.h>
#include <gdiplus.h>
#include <iostream>
#include <map>
#include <agents.h>
#include <ppl.h>

using namespace concurrency;
using namespace Gdiplus;
using namespace std;

// Retrieves the red, green, and blue components from the given

```

```

// retrieves the red, green, and blue components from the given
// color value.
void GetRGB(DWORD color, BYTE& r, BYTE& g, BYTE& b)
{
    r = static_cast<BYTE>((color & 0x00ff0000) >> 16);
    g = static_cast<BYTE>((color & 0x0000ff00) >> 8);
    b = static_cast<BYTE>((color & 0x000000ff));
}

// Creates a single color value from the provided red, green,
// and blue components.
DWORD MakeColor(BYTE r, BYTE g, BYTE b)
{
    return (r<<16) | (g<<8) | (b);
}

// Calls the provided function for each pixel in a Bitmap object.
void ProcessImage(Bitmap* bmp, const function<void (DWORD*)>& f)
{
    int width = bmp->GetWidth();
    int height = bmp->GetHeight();

    // Lock the bitmap.
    BitmapData bitmapData;
    Rect rect(0, 0, bmp->GetWidth(), bmp->GetHeight());
    bmp->LockBits(&rect, ImageLockModeWrite, PixelFormat32bppRGB, &bitmapData);

    // Get a pointer to the bitmap data.
    DWORD* image_bits = (DWORD*)bitmapData.Scan0;

    // Call the function for each pixel in the image.
    parallel_for (0, height, [&, width](int y)
    {
        for (int x = 0; x < width; ++x)
        {
            // Get the current pixel value.
            DWORD* curr_pixel = image_bits + (y * width) + x;

            // Call the function.
            f(*curr_pixel);
        }
    });

    // Unlock the bitmap.
    bmp->UnlockBits(&bitmapData);
}

// Converts the given image to grayscale.
Bitmap* Grayscale(Bitmap* bmp)
{
    ProcessImage(bmp,
        [(DWORD& color) {
            BYTE r, g, b;
            GetRGB(color, r, g, b);

            // Set each color component to the average of
            // the original components.
            BYTE c = (static_cast<WORD>(r) + g + b) / 3;
            color = MakeColor(c, c, c);
        }
    );
    return bmp;
}

// Applies sepia toning to the provided image.
Bitmap* Sepiatone(Bitmap* bmp)
{
    ProcessImage(bmp,
        [(DWORD& color) {
            BYTE r, g, b;

```

```

        BYTE r0, g0, b0;
        GetRGB(color, r0, g0, b0);

        WORD r1 = static_cast<WORD>((r0 * .393) + (g0 * .769) + (b0 * .189));
        WORD g1 = static_cast<WORD>((r0 * .349) + (g0 * .686) + (b0 * .168));
        WORD b1 = static_cast<WORD>((r0 * .272) + (g0 * .534) + (b0 * .131));

        color = MakeColor(min(0xff, r1), min(0xff, g1), min(0xff, b1));
    }
);
return bmp;
}

// Applies the given color mask to each pixel in the provided image.
Bitmap* ColorMask(Bitmap* bmp, DWORD mask)
{
    ProcessImage(bmp,
        [mask](DWORD& color) {
            color = color & mask;
        }
    );
    return bmp;
}

// Darkens the provided image by the given amount.
Bitmap* Darken(Bitmap* bmp, unsigned int percent)
{
    if (percent > 100)
        throw invalid_argument("Darken: percent must less than 100.");

    double factor = percent / 100.0;

    ProcessImage(bmp,
        [factor](DWORD& color) {
            BYTE r, g, b;
            GetRGB(color, r, g, b);
            r = static_cast<BYTE>(factor*r);
            g = static_cast<BYTE>(factor*g);
            b = static_cast<BYTE>(factor*b);
            color = MakeColor(r, g, b);
        }
    );
    return bmp;
}

// Determines which color component (red, green, or blue) is most dominant
// in the given image and returns a corresponding color mask.
DWORD GetColorDominance(Bitmap* bmp)
{
    // The ProcessImage function processes the image in parallel.
    // The following combinable objects enable the callback function
    // to increment the color counts without using a lock.
    combinable<unsigned int> reds;
    combinable<unsigned int> greens;
    combinable<unsigned int> blues;

    ProcessImage(bmp,
        [&](DWORD& color) {
            BYTE r, g, b;
            GetRGB(color, r, g, b);
            if (r >= g && r >= b)
                reds.local()++;
            else if (g >= r && g >= b)
                greens.local()++;
            else
                blues.local()++;
        }
    );
}

```

```

// Determine which color is dominant and return the corresponding
// color mask.

unsigned int r = reds.combine(plus<unsigned int>());
unsigned int g = greens.combine(plus<unsigned int>());
unsigned int b = blues.combine(plus<unsigned int>());

if (r + r >= g + b)
    return 0x00ff0000;
else if (g + g >= r + b)
    return 0x0000ff00;
else
    return 0x000000ff;
}

// Retrieves the class identifier for the given MIME type of an encoder.
int GetEncoderClsid(const WCHAR* format, CLSID* pClsid)
{
    UINT num = 0;           // number of image encoders
    UINT size = 0;         // size of the image encoder array in bytes

    ImageCodecInfo* pImageCodecInfo = nullptr;

    GetImageEncodersSize(&num, &size);
    if(size == 0)
        return -1; // Failure

    pImageCodecInfo = (ImageCodecInfo*)(malloc(size));
    if(pImageCodecInfo == nullptr)
        return -1; // Failure

    GetImageEncoders(num, size, pImageCodecInfo);

    for(UINT j = 0; j < num; ++j)
    {
        if( wcsncmp(pImageCodecInfo[j].MimeType, format) == 0 )
        {
            *pClsid = pImageCodecInfo[j].Clsid;
            free(pImageCodecInfo);
            return j; // Success
        }
    }

    free(pImageCodecInfo);
    return -1; // Failure
}

// A synchronization primitive that is signaled when its
// count reaches zero.
class countdown_event
{
public:
    countdown_event(unsigned int count = 0)
        : _current(static_cast<long>(count))
    {
        // Set the event if the initial count is zero.
        if (_current == 0L)
            _event.set();
    }

    // Decrements the event counter.
    void signal() {
        if(InterlockedDecrement(&_current) == 0L) {
            _event.set();
        }
    }

    // Increments the event counter.
    void add_count() {

```

```

        if(InterlockedIncrement(&_current) == 1L) {
            _event.reset();
        }
    }

    // Blocks the current context until the event is set.
    void wait() {
        _event.wait();
    }

private:
    // The current count.
    volatile long _current;
    // The event that is set when the counter reaches zero.
    event _event;

    // Disable copy constructor.
    countdown_event(const countdown_event&);
    // Disable assignment.
    countdown_event const & operator=(countdown_event const&);
};

// Demonstrates how to set up a message network that performs a series of
// image processing operations on each JPEG image in the given directory and
// saves each altered image as a Windows bitmap.
void ProcessImages(const wstring& directory)
{
    // Holds the number of active image processing operations and
    // signals to the main thread that processing is complete.
    countdown_event active(0);

    // Maps Bitmap objects to their original file names.
    map<Bitmap*, wstring> bitmap_file_names;

    //
    // Create the nodes of the network.
    //

    // Loads Bitmap objects from disk.
    transformer<wstring, Bitmap*> load_bitmap(
        [&](wstring file_name) -> Bitmap* {
            Bitmap* bmp = new Bitmap(file_name.c_str());
            if (bmp != nullptr)
                bitmap_file_names.insert(make_pair(bmp, file_name));
            return bmp;
        }
    );

    // Holds loaded Bitmap objects.
    unbounded_buffer<Bitmap*> loaded_bitmaps;

    // Converts images that are authored by Tom to grayscale.
    transformer<Bitmap*, Bitmap*> grayscale(
        [](Bitmap* bmp) {
            return Grayscale(bmp);
        },
        nullptr,
        [](Bitmap* bmp) -> bool {
            if (bmp == nullptr)
                return false;

            // Retrieve the artist name from metadata.
            UINT size = bmp->GetPropertyItemSize(PropertyTagArtist);
            if (size == 0)
                // Image does not have the Artist property.
                return false;

            PropertyItem* artistProperty = (PropertyItem*) malloc(size);
            bmp->GetPropertyItem(PropertyTagArtist, size, artistProperty);

```

```

        string artist(reinterpret_cast<char*>(artistProperty->value));
        free(artistProperty);

        return (artist.find("Tom ") == 0);
    }
};

// Removes the green and blue color components from images that have red as
// their dominant color.
transformer<Bitmap*, Bitmap*> colormask(
    [](Bitmap* bmp) {
        return ColorMask(bmp, 0x00ff0000);
    },
    nullptr,
    [](Bitmap* bmp) -> bool {
        if (bmp == nullptr)
            return false;
        return (GetColorDominance(bmp) == 0x00ff0000);
    }
);

// Darkens the color of the provided Bitmap object.
transformer<Bitmap*, Bitmap*> darken([](Bitmap* bmp) {
    return Darken(bmp, 50);
});

// Applies sepia toning to the remaining images.
transformer<Bitmap*, Bitmap*> sepiaTone(
    [](Bitmap* bmp) {
        return SepiaTone(bmp);
    },
    nullptr,
    [](Bitmap* bmp) -> bool { return bmp != nullptr; }
);

// Saves Bitmap objects to disk.
transformer<Bitmap*, Bitmap*> save_bitmap([&](Bitmap* bmp) -> Bitmap* {
    // Replace the file extension with .bmp.
    wstring file_name = bitmap_file_names[bmp];
    file_name.replace(file_name.rfind(L'.') + 1, 3, L"bmp");

    // Save the processed image.
    CLSID bmpClsid;
    GetEncoderClsid(L"image/bmp", &bmpClsid);
    bmp->Save(file_name.c_str(), &bmpClsid);

    return bmp;
});

// Deletes Bitmap objects.
transformer<Bitmap*, Bitmap*> delete_bitmap([](Bitmap* bmp) -> Bitmap* {
    delete bmp;
    return nullptr;
});

// Decrements the event counter.
call<Bitmap*> decrement([&](Bitmap* _) {
    active.signal();
});

//
// Connect the network.
//

load_bitmap.link_target(&loaded_bitmaps);

loaded_bitmaps.link_target(&grayscale);
loaded_bitmaps.link_target(&colormask);
colormask.link_target(&darken);

```

```

loaded_bitmaps.link_target(&sepiatone);
loaded_bitmaps.link_target(&decrement);

grayscale.link_target(&save_bitmap);
darken.link_target(&save_bitmap);
sepiatone.link_target(&save_bitmap);

save_bitmap.link_target(&delete_bitmap);
delete_bitmap.link_target(&decrement);

// Traverse all files in the directory.
wstring searchPattern = directory;
searchPattern.append(L"\\*");

WIN32_FIND_DATA fileFindData;
HANDLE hFind = FindFirstFile(searchPattern.c_str(), &fileFindData);
if (hFind == INVALID_HANDLE_VALUE)
    return;
do
{
    if (!(fileFindData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY))
    {
        wstring file = fileFindData.cFileName;

        // Process only JPEG files.
        if (file.rfind(L".jpg") == file.length() - 4)
        {
            // Form the full path to the file.
            wstring full_path(directory);
            full_path.append(L"\\");
            full_path.append(file);

            // Increment the count of work items.
            active.add_count();

            // Send the path name to the network.
            send(load_bitmap, full_path);
        }
    }
}
while (FindNextFile(hFind, &fileFindData) != 0);
FindClose(hFind);

// Wait for all operations to finish.
active.wait();
}

int wmain()
{
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR          gdiplusToken;

    // Initialize GDI+.
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, nullptr);

    // Perform image processing.
    // TODO: Change this path if necessary.
    ProcessImages(L"C:\\Users\\Public\\Pictures\\Sample Pictures");

    // Shutdown GDI+.
    GdiplusShutdown(gdiplusToken);
}

```

The following illustration shows sample output. Each source image is above its corresponding modified image.



`Lighthouse` is authored by Tom Alphin and therefore is converted to grayscale. `Chrysanthemum`, `Desert`, `Koala`, and `Tulips` have red as the dominant color and therefore have the blue and green color components removed and are darkened. `Hydrangeas`, `Jellyfish`, and `Penguins` match the default criteria and therefore are sepia toned.

[\[Top\]](#)

### Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `image-processing-network.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /DUNICODE /EHsc image-processing-network.cpp /link gdiplus.lib
```

## See also

[Concurrency Runtime Walkthroughs](#)

# Walkthrough: Implementing Futures

4/25/2019 • 5 minutes to read • [Edit Online](#)

This topic shows how to implement futures in your application. The topic demonstrates how to combine existing functionality in the Concurrency Runtime into something that does more.

## IMPORTANT

This topic illustrates the concept of futures for demonstration purposes. We recommend that you use `std::future` or `concurrency::task` when you require an asynchronous task that computes a value for later use.

A *task* is a computation that can be decomposed into additional, more fine-grained, computations. A *future* is an asynchronous task that computes a value for later use.

To implement futures, this topic defines the `async_future` class. The `async_future` class uses these components of the Concurrency Runtime: the `concurrency::task_group` class and the `concurrency::single_assignment` class. The `async_future` class uses the `task_group` class to compute a value asynchronously and the `single_assignment` class to store the result of the computation. The constructor of the `async_future` class takes a work function that computes the result, and the `get` method retrieves the result.

## To implement the `async_future` class

1. Declare a template class named `async_future` that is parameterized on the type of the resulting computation. Add `public` and `private` sections to this class.

```
template <typename T>
class async_future
{
public:
private:
};
```

1. In the `private` section of the `async_future` class, declare a `task_group` and a `single_assignment` data member.

```
// Executes the asynchronous work function.
task_group _tasks;

// Stores the result of the asynchronous work function.
single_assignment<T> _value;
```

1. In the `public` section of the `async_future` class, implement the constructor. The constructor is a template that is parameterized on the work function that computes the result. The constructor asynchronously executes the work function in the `task_group` data member and uses the `concurrency::send` function to write the result to the `single_assignment` data member.

```

template <class Functor>
explicit async_future(Functor&& fn)
{
    // Execute the work function in a task group and send the result
    // to the single_assignment object.
    _tasks.run([fn, this]() {
        send(_value, fn());
    });
}

```

1. In the `public` section of the `async_future` class, implement the destructor. The destructor waits for the task to finish.

```

~async_future()
{
    // Wait for the task to finish.
    _tasks.wait();
}

```

1. In the `public` section of the `async_future` class, implement the `get` method. This method uses the [concurrency::receive](#) function to retrieve the result of the work function.

```

// Retrieves the result of the work function.
// This method blocks if the async_future object is still
// computing the value.
T get()
{
    return receive(_value);
}

```

## Example

### Description

The following example shows the complete `async_future` class and an example of its usage. The `wmain` function creates a `std::vector` object that contains 10,000 random integer values. It then uses `async_future` objects to find the smallest and largest values that are contained in the `vector` object.

### Code

```

// futures.cpp
// compile with: /EHsc
#include <ppl.h>
#include <agents.h>
#include <vector>
#include <algorithm>
#include <iostream>
#include <numeric>
#include <random>

using namespace concurrency;
using namespace std;

template <typename T>
class async_future
{
public:
    template <class Functor>
    explicit async_future(Functor&& fn)
    {

```

```

    // Execute the work function in a task group and send the result
    // to the single_assignment object.
    _tasks.run([fn, this]() {
        send(_value, fn());
    });
}

~async_future()
{
    // Wait for the task to finish.
    _tasks.wait();
}

// Retrieves the result of the work function.
// This method blocks if the async_future object is still
// computing the value.
T get()
{
    return receive(_value);
}

private:
    // Executes the asynchronous work function.
    task_group _tasks;

    // Stores the result of the asynchronous work function.
    single_assignment<T> _value;
};

int wmain()
{
    // Create a vector of 10000 integers, where each element
    // is between 0 and 9999.
    mt19937 gen(2);
    vector<int> values(10000);
    generate(begin(values), end(values), [&gen]{ return gen()%10000; });

    // Create a async_future object that finds the smallest value in the
    // vector.
    async_future<int> min_value([&]() -> int {
        int smallest = INT_MAX;
        for_each(begin(values), end(values), [&](int value) {
            if (value < smallest)
            {
                smallest = value;
            }
        });
        return smallest;
    });

    // Create a async_future object that finds the largest value in the
    // vector.
    async_future<int> max_value([&]() -> int {
        int largest = INT_MIN;
        for_each(begin(values), end(values), [&](int value) {
            if (value > largest)
            {
                largest = value;
            }
        });
        return largest;
    });

    // Calculate the average value of the vector while the async_future objects
    // work in the background.
    int sum = accumulate(begin(values), end(values), 0);
    int average = sum / values.size();

    // Print the smallest, largest, and average values.

```

```

wcout << L"smallest: " << min_value.get() << endl
      << L"largest: " << max_value.get() << endl
      << L"average: " << average << endl;
}

```

## Comments

This example produces the following output:

```

smallest: 0
largest: 9999
average: 4981

```

The example uses the `async_future::get` method to retrieve the results of the computation. The `async_future::get` method waits for the computation to finish if the computation is still active.

## Robust Programming

To extend the `async_future` class to handle exceptions that are thrown by the work function, modify the `async_future::get` method to call the `concurrency::task_group::wait` method. The `task_group::wait` method throws any exceptions that were generated by the work function.

The following example shows the modified version of the `async_future` class. The `wmain` function uses a `try-catch` block to print the result of the `async_future` object or to print the value of the exception that is generated by the work function.

```

// futures-with-eh.cpp
// compile with: /EHsc
#include <ppl.h>
#include <agents.h>
#include <vector>
#include <algorithm>
#include <iostream>

using namespace concurrency;
using namespace std;

template <typename T>
class async_future
{
public:
    template <class Functor>
    explicit async_future(Functor&& fn)
    {
        // Execute the work function in a task group and send the result
        // to the single_assignment object.
        _tasks.run([fn, this]() {
            send(_value, fn());
        });
    }

    ~async_future()
    {
        // Wait for the task to finish.
        _tasks.wait();
    }

    // Retrieves the result of the work function.
    // This method blocks if the async_future object is still
    // computing the value.
    T get()
    {
        // Wait for the task to finish.

```

```

// Wait for the task to finish.
// The wait method throws any exceptions that were generated
// by the work function.
_tasks.wait();

// Return the result of the computation.
return receive(_value);
}

private:
// Executes the asynchronous work function.
task_group _tasks;

// Stores the result of the asynchronous work function.
single_assignment<T> _value;
};

int wmain()
{
// For illustration, create a async_future with a work
// function that throws an exception.
async_future<int> f([]() -> int {
    throw exception("error");
});

// Try to read from the async_future object.
try
{
    int value = f.get();
    wcout << L"f contains value: " << value << endl;
}
catch (const exception& e)
{
    wcout << L"caught exception: " << e.what() << endl;
}
}
}

```

This example produces the following output:

```
caught exception: error
```

For more information about the exception handling model in the Concurrency Runtime, see [Exception Handling](#).

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `futures.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc futures.cpp
```

## See also

[Concurrency Runtime Walkthroughs](#)

[Exception Handling](#)

[task\\_group Class](#)

[single\\_assignment Class](#)

# Walkthrough: Using join to Prevent Deadlock

4/25/2019 • 10 minutes to read • [Edit Online](#)

This topic uses the dining philosophers problem to illustrate how to use the `concurrency::join` class to prevent deadlock in your application. In a software application, *deadlock* occurs when two or more processes each hold a resource and mutually wait for another process to release some other resource.

The dining philosophers problem is a specific example of the general set of problems that may occur when a set of resources is shared among multiple concurrent processes.

## Prerequisites

Read the following topics before you start this walkthrough:

- [Asynchronous Agents](#)
- [Walkthrough: Creating an Agent-Based Application](#)
- [Asynchronous Message Blocks](#)
- [Message Passing Functions](#)
- [Synchronization Data Structures](#)

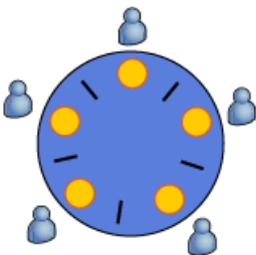
## Sections

This walkthrough contains the following sections:

- [The Dining Philosophers Problem](#)
- [A Naïve Implementation](#)
- [Using join to Prevent Deadlock](#)

## The Dining Philosophers Problem

The dining philosophers problem illustrates how deadlock occurs in an application. In this problem, five philosophers sit at a round table. Every philosopher alternates between thinking and eating. Every philosopher must share a chopstick with the neighbor to the left and another chopstick with the neighbor to the right. The following illustration shows this layout.



To eat, a philosopher must hold two chopsticks. If every philosopher holds just one chopstick and is waiting for another one, then no philosopher can eat and all starve.

[\[Top\]](#)

# A Naïve Implementation

The following example shows a naïve implementation of the dining philosophers problem. The `philosopher` class, which derives from `concurrency::agent`, enables each philosopher to act independently. The example uses a shared array of `concurrency::critical_section` objects to give each `philosopher` object exclusive access to a pair of chopsticks.

To relate the implementation to the illustration, the `philosopher` class represents one philosopher. An `int` variable represents each chopstick. The `critical_section` objects serve as holders on which the chopsticks rest. The `run` method simulates the life of the philosopher. The `think` method simulates the act of thinking and the `eat` method simulates the act of eating.

A `philosopher` object locks both `critical_section` objects to simulate the removal of the chopsticks from the holders before it calls the `eat` method. After the call to `eat`, the `philosopher` object returns the chopsticks to the holders by setting the `critical_section` objects back to the unlocked state.

The `pickup_chopsticks` method illustrates where deadlock can occur. If every `philosopher` object gains access to one of the locks, then no `philosopher` object can continue because the other lock is controlled by another `philosopher` object.

## Example

### Description

### Code

```
// philosophers-deadlock.cpp
// compile with: /EHsc
#include <agents.h>
#include <string>
#include <array>
#include <iostream>
#include <algorithm>
#include <random>

using namespace concurrency;
using namespace std;

// Defines a single chopstick.
typedef int chopstick;

// The total number of philosophers.
const int philosopher_count = 5;

// The number of times each philosopher should eat.
const int eat_count = 50;

// A shared array of critical sections. Each critical section
// guards access to a single chopstick.
critical_section locks[philosopher_count];

// Implements the logic for a single dining philosopher.
class philosopher : public agent
{
public:
    explicit philosopher(chopstick& left, chopstick& right, const wstring& name)
        : _left(left)
        , _right(right)
        , _name(name)
        , _random_generator(42)
    {
        send(_times_eaten, 0);
    }
};
```

```

    // Retrieves the number of times the philosopher has eaten.
    int times_eaten()
    {
        return receive(_times_eaten);
    }

    // Retrieves the name of the philosopher.
    wstring name() const
    {
        return _name;
    }

protected:
    // Performs the main logic of the dining philosopher algorithm.
    void run()
    {
        // Repeat the thinks/eat cycle a set number of times.
        for (int n = 0; n < eat_count; ++n)
        {
            think();

            pickup_chopsticks();
            eat();
            send(_times_eaten, n+1);
            putdown_chopsticks();
        }

        done();
    }

    // Gains access to the chopsticks.
    void pickup_chopsticks()
    {
        // Deadlock occurs here if each philosopher gains access to one
        // of the chopsticks and mutually waits for another to release
        // the other chopstick.
        locks[_left].lock();
        locks[_right].lock();
    }

    // Releases the chopsticks for others.
    void putdown_chopsticks()
    {
        locks[_right].unlock();
        locks[_left].unlock();
    }

    // Simulates thinking for a brief period of time.
    void think()
    {
        random_wait(100);
    }

    // Simulates eating for a brief period of time.
    void eat()
    {
        random_wait(100);
    }

private:
    // Yields the current context for a random period of time.
    void random_wait(unsigned int max)
    {
        concurrency::wait(_random_generator()%max);
    }

private:
    // Index of the left chopstick in the chopstick array

```

```

// Index of the left chopstick in the chopstick array.
chopstick& _left;
// Index of the right chopstick in the chopstick array.
chopstick& _right;

// The name of the philosopher.
wstring _name;
// Stores the number of times the philosopher has eaten.
overwrite_buffer<int> _times_eaten;

// A random number generator.
mt19937 _random_generator;
};

int wmain()
{
    // Create an array of index values for the chopsticks.
    array<chopstick, philosopher_count> chopsticks = {0, 1, 2, 3, 4};

    // Create an array of philosophers. Each pair of neighboring
    // philosophers shares one of the chopsticks.
    array<philosopher, philosopher_count> philosophers = {
        philosopher(chopsticks[0], chopsticks[1], L"aristotle"),
        philosopher(chopsticks[1], chopsticks[2], L"descartes"),
        philosopher(chopsticks[2], chopsticks[3], L"hobbes"),
        philosopher(chopsticks[3], chopsticks[4], L"socrates"),
        philosopher(chopsticks[4], chopsticks[0], L"plato"),
    };

    // Begin the simulation.
    for_each (begin(philosophers), end(philosophers), [](philosopher& p) {
        p.start();
    });

    // Wait for each philosopher to finish and print his name and the number
    // of times he has eaten.
    for_each (begin(philosophers), end(philosophers), [](philosopher& p) {
        agent::wait(&p);
        wcout << p.name() << L" ate " << p.times_eaten() << L" times." << endl;
    });
}

```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `philosophers-deadlock.cpp` and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc philosophers-deadlock.cpp**

[\[Top\]](#)

## Using join to Prevent Deadlock

This section shows how to use message buffers and message-passing functions to eliminate the chance of deadlock.

To relate this example to the earlier one, the `philosopher` class replaces each `critical_section` object by using a `concurrency::unbounded_buffer` object and a `join` object. The `join` object serves as an arbiter that provides the chopsticks to the philosopher.

This example uses the `unbounded_buffer` class because when a target receives a message from an `unbounded_buffer` object, the message is removed from the message queue. This enables an `unbounded_buffer` object that holds a message to indicate that the chopstick is available. An `unbounded_buffer` object that holds no

message indicates that the chopstick is being used.

This example uses a non-greedy `join` object because a non-greedy join gives each `philosopher` object access to both chopsticks only when both `unbounded_buffer` objects contain a message. A greedy join would not prevent deadlock because a greedy join accepts messages as soon as they become available. Deadlock can occur if all greedy `join` objects receive one of the messages but wait forever for the other to become available.

For more information about greedy and non-greedy joins, and the differences between the various message buffer types, see [Asynchronous Message Blocks](#).

#### To prevent deadlock in this example

1. Remove the following code from the example.

```
// A shared array of critical sections. Each critical section
// guards access to a single chopstick.
critical_section locks[philosopher_count];
```

1. Change the type of the `_left` and `_right` data members of the `philosopher` class to `unbounded_buffer`.

```
// Message buffer for the left chopstick.
unbounded_buffer<chopstick>& _left;
// Message buffer for the right chopstick.
unbounded_buffer<chopstick>& _right;
```

1. Modify the `philosopher` constructor to take `unbounded_buffer` objects as its parameters.

```
explicit philosopher(unbounded_buffer<chopstick>& left,
                    unbounded_buffer<chopstick>& right, const wstring& name)
    : _left(left)
    , _right(right)
    , _name(name)
    , _random_generator(42)
{
    send(_times_eaten, 0);
}
```

1. Modify the `pickup_chopsticks` method to use a non-greedy `join` object to receive messages from the message buffers for both chopsticks.

```
// Gains access to the chopsticks.
vector<int> pickup_chopsticks()
{
    // Create a non-greedy join object and link it to the left and right
    // chopstick.
    join<chopstick, non_greedy> j(2);
    _left.link_target(&j);
    _right.link_target(&j);

    // Receive from the join object. This resolves the deadlock situation
    // because a non-greedy join removes the messages only when a message
    // is available from each of its sources.
    return receive(&j);
}
```

1. Modify the `putdown_chopsticks` method to release access to the chopsticks by sending a message to the message buffers for both chopsticks.

```
// Releases the chopsticks for others.
void putdown_chopsticks(int left, int right)
{
    // Add the values of the messages back to the message queue.
    asend(&_left, left);
    asend(&_right, right);
}
```

1. Modify the `run` method to hold the results of the `pickup_chopsticks` method and to pass those results to the `putdown_chopsticks` method.

```
// Performs the main logic of the dining philosopher algorithm.
void run()
{
    // Repeat the thinks/eat cycle a set number of times.
    for (int n = 0; n < eat_count; ++n)
    {
        think();

        vector<int> v = pickup_chopsticks();

        eat();

        send(_times_eaten, n+1);

        putdown_chopsticks(v[0], v[1]);
    }

    done();
}
```

1. Modify the declaration of the `chopsticks` variable in the `wmain` function to be an array of `unbounded_buffer` objects that each hold one message.

```
// Create an array of message buffers to hold the chopsticks.
array<unbounded_buffer<chopstick>, philosopher_count> chopsticks;

// Send a value to each message buffer in the array.
// The value of the message is not important. A buffer that contains
// any message indicates that the chopstick is available.
for_each (begin(chopsticks), end(chopsticks),
    [](unbounded_buffer<chopstick>& c) {
        send(c, 1);
    });
```

## Example

### Description

The following shows the completed example that uses non-greedy `join` objects to eliminate the risk of deadlock.

### Code

```
// philosophers-join.cpp
// compile with: /EHsc
#include <agents.h>
#include <string>
#include <array>
#include <iostream>
#include <algorithm>
#include <random>
```

```

using namespace concurrency;
using namespace std;

// Defines a single chopstick.
typedef int chopstick;

// The total number of philosophers.
const int philosopher_count = 5;

// The number of times each philosopher should eat.
const int eat_count = 50;

// Implements the logic for a single dining philosopher.
class philosopher : public agent
{
public:
    explicit philosopher(unbounded_buffer<chopstick>& left,
        unbounded_buffer<chopstick>& right, const wstring& name)
        : _left(left)
        , _right(right)
        , _name(name)
        , _random_generator(42)
    {
        send(_times_eaten, 0);
    }

    // Retrieves the number of times the philosopher has eaten.
    int times_eaten()
    {
        return receive(_times_eaten);
    }

    // Retrieves the name of the philosopher.
    wstring name() const
    {
        return _name;
    }

protected:
    // Performs the main logic of the dining philosopher algorithm.
    void run()
    {
        // Repeat the thinks/eat cycle a set number of times.
        for (int n = 0; n < eat_count; ++n)
        {
            think();

            vector<int> v = pickup_chopsticks();

            eat();

            send(_times_eaten, n+1);

            putdown_chopsticks(v[0], v[1]);
        }

        done();
    }

    // Gains access to the chopsticks.
    vector<int> pickup_chopsticks()
    {
        // Create a non-greedy join object and link it to the left and right
        // chopstick.
        join<chopstick, non_greedy> j(2);
        _left.link_target(&j);
        _right.link_target(&j);
    }
};

```

```

    // Receive from the join object. This resolves the deadlock situation
    // because a non-greedy join removes the messages only when a message
    // is available from each of its sources.
    return receive(&j);
}

// Releases the chopsticks for others.
void putdown_chopsticks(int left, int right)
{
    // Add the values of the messages back to the message queue.
    asend(&_left, left);
    asend(&_right, right);
}

// Simulates thinking for a brief period of time.
void think()
{
    random_wait(100);
}

// Simulates eating for a brief period of time.
void eat()
{
    random_wait(100);
}

private:
    // Yields the current context for a random period of time.
    void random_wait(unsigned int max)
    {
        concurrency::wait(_random_generator()%max);
    }

private:
    // Message buffer for the left chopstick.
    unbounded_buffer<chopstick>& _left;
    // Message buffer for the right chopstick.
    unbounded_buffer<chopstick>& _right;

    // The name of the philosopher.
    wstring _name;
    // Stores the number of times the philosopher has eaten.
    overwrite_buffer<int> _times_eaten;

    // A random number generator.
    mt19937 _random_generator;
};

int wmain()
{
    // Create an array of message buffers to hold the chopsticks.
    array<unbounded_buffer<chopstick>, philosopher_count> chopsticks;

    // Send a value to each message buffer in the array.
    // The value of the message is not important. A buffer that contains
    // any message indicates that the chopstick is available.
    for_each (begin(chopsticks), end(chopsticks),
        [](unbounded_buffer<chopstick>& c) {
            send(c, 1);
        });

    // Create an array of philosophers. Each pair of neighboring
    // philosophers shares one of the chopsticks.
    array<philosopher, philosopher_count> philosophers = {
        philosopher(chopsticks[0], chopsticks[1], L"aristotle"),
        philosopher(chopsticks[1], chopsticks[2], L"descartes"),
        philosopher(chopsticks[2], chopsticks[3], L"hobbes"),
        philosopher(chopsticks[3], chopsticks[4], L"socrates"),
        philosopher(chopsticks[4], chopsticks[0], L"plato")
    };
}

```

```

        philosopher(chopsticks[4], chopsticks[0], L plato );
    };

    // Begin the simulation.
    for_each (begin(philosophers), end(philosophers), [](philosopher& p) {
        p.start();
    });

    // Wait for each philosopher to finish and print his name and the number
    // of times he has eaten.
    for_each (begin(philosophers), end(philosophers), [](philosopher& p) {
        agent::wait(&p);
        wcout << p.name() << L" ate " << p.times_eaten() << L" times." << endl;
    });
}

```

## Comments

This example produces the following output.

```

aristotle ate 50 times.
descartes ate 50 times.
hobbes ate 50 times.
socrates ate 50 times.
plato ate 50 times.

```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `philosophers-join.cpp` and then run the following command in a Visual Studio Command Prompt window.

**cl.exe /EHsc philosophers-join.cpp**

[\[Top\]](#)

## See also

- [Concurrency Runtime Walkthroughs](#)
- [Asynchronous Agents Library](#)
- [Asynchronous Agents](#)
- [Asynchronous Message Blocks](#)
- [Message Passing Functions](#)
- [Synchronization Data Structures](#)

# Walkthrough: Removing Work from a User-Interface Thread

4/25/2019 • 13 minutes to read • [Edit Online](#)

This document demonstrates how to use the Concurrency Runtime to move the work that is performed by the user-interface (UI) thread in a Microsoft Foundation Classes (MFC) application to a worker thread. This document also demonstrates how to improve the performance of a lengthy drawing operation.

Removing work from the UI thread by offloading blocking operations, for example, drawing, to worker threads can improve the responsiveness of your application. This walkthrough uses a drawing routine that generates the Mandelbrot fractal to demonstrate a lengthy blocking operation. The generation of the Mandelbrot fractal is also a good candidate for parallelization because the computation of each pixel is independent of all other computations.

## Prerequisites

Read the following topics before you start this walkthrough:

- [Task Parallelism](#)
- [Asynchronous Message Blocks](#)
- [Message Passing Functions](#)
- [Parallel Algorithms](#)
- [Cancellation in the PPL](#)

We also recommend that you understand the basics of MFC application development and GDI+ before you start this walkthrough. For more information about MFC, see [MFC Desktop Applications](#). For more information about GDI+, see [GDI+](#).

## Sections

This walkthrough contains the following sections:

- [Creating the MFC Application](#)
- [Implementing the Serial Version of the Mandelbrot Application](#)
- [Removing Work from the User-Interface Thread](#)
- [Improving Drawing Performance](#)
- [Adding Support for Cancellation](#)

## Creating the MFC Application

This section describes how to create the basic MFC application.

### To create a Visual C++ MFC application

1. Use the **MFC Application Wizard** to create an MFC application with all the default settings. See [Walkthrough: Using the New MFC Shell Controls](#) for instructions on how to open the wizard for your version of Visual Studio.

2. Type a name for the project, for example, `Mandelbrot`, and then click **OK** to display the **MFC Application Wizard**.
3. In the **Application Type** pane, select **Single document**. Ensure that the **Document/View architecture support** check box is cleared.
4. Click **Finish** to create the project and close the **MFC Application Wizard**.

Verify that the application was created successfully by building and running it. To build the application, on the **Build** menu, click **Build Solution**. If the application builds successfully, run the application by clicking **Start Debugging** on the **Debug** menu.

## Implementing the Serial Version of the Mandelbrot Application

This section describes how to draw the Mandelbrot fractal. This version draws the Mandelbrot fractal to a GDI+ `Bitmap` object and then copies the contents of that bitmap to the client window.

**To implement the serial version of the Mandelbrot application**

1. In `stdafx.h`, add the following `#include` directive:

```
#include <memory>
```

2. In `ChildView.h`, after the `pragma` directive, define the `BitmapPtr` type. The `BitmapPtr` type enables a pointer to a `Bitmap` object to be shared by multiple components. The `Bitmap` object is deleted when it is no longer referenced by any component.

```
typedef std::shared_ptr<Gdiplus::Bitmap> BitmapPtr;
```

3. In `ChildView.h`, add the following code to the `protected` section of the `CChildView` class:

```
protected:  
    // Draws the Mandelbrot fractal to the specified Bitmap object.  
    void DrawMandelbrot(BitmapPtr);  
  
protected:  
    ULONG_PTR m_gdiplusToken;
```

4. In `ChildView.cpp`, comment out or remove the following lines.

```
//#ifdef _DEBUG  
//#define new DEBUG_NEW  
//#endif
```

In Debug builds, this step prevents the application from using the `DEBUG_NEW` allocator, which is incompatible with GDI+.

5. In `ChildView.cpp`, add a `using` directive to the `Gdiplus` namespace.

```
using namespace Gdiplus;
```

6. Add the following code to the constructor and destructor of the `CChildView` class to initialize and shut down GDI+.

```

CChildView::CChildView()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    GdiplusStartup(&m_gdiplusToken, &gdiplusStartupInput, NULL);
}

CChildView::~CChildView()
{
    // Shutdown GDI+.
    GdiplusShutdown(m_gdiplusToken);
}

```

7. Implement the `CChildView::DrawMandelbrot` method. This method draws the Mandelbrot fractal to the specified `Bitmap` object.

```

// Draws the Mandelbrot fractal to the specified Bitmap object.
void CChildView::DrawMandelbrot(BitmapPtr pBitmap)
{
    if (pBitmap == NULL)
        return;

    // Get the size of the bitmap.
    const UINT width = pBitmap->GetWidth();
    const UINT height = pBitmap->GetHeight();

    // Return if either width or height is zero.
    if (width == 0 || height == 0)
        return;

    // Lock the bitmap into system memory.
    BitmapData bitmapData;
    Rect rectBmp(0, 0, width, height);
    pBitmap->LockBits(&rectBmp, ImageLockModeWrite, PixelFormat32bppRGB,
        &bitmapData);

    // Obtain a pointer to the bitmap bits.
    int* bits = reinterpret_cast<int*>(bitmapData.Scan0);

    // Real and imaginary bounds of the complex plane.
    double re_min = -2.1;
    double re_max = 1.0;
    double im_min = -1.3;
    double im_max = 1.3;

    // Factors for mapping from image coordinates to coordinates on the complex plane.
    double re_factor = (re_max - re_min) / (width - 1);
    double im_factor = (im_max - im_min) / (height - 1);

    // The maximum number of iterations to perform on each point.
    const UINT max_iterations = 1000;

    // Compute whether each point lies in the Mandelbrot set.
    for (UINT row = 0u; row < height; ++row)
    {
        // Obtain a pointer to the bitmap bits for the current row.
        int *destPixel = bits + (row * width);

        // Convert from image coordinate to coordinate on the complex plane.
        double y0 = im_max - (row * im_factor);

        for (UINT col = 0u; col < width; ++col)
        {
            // Convert from image coordinate to coordinate on the complex plane.
            double x0 = re_min + col * re_factor;

```

```

double x = x0;
double y = y0;

UINT iter = 0;
double x_sq, y_sq;
while (iter < max_iterations && ((x_sq = x*x) + (y_sq = y*y) < 4))
{
    double temp = x_sq - y_sq + x0;
    y = 2 * x * y + y0;
    x = temp;
    ++iter;
}

// If the point is in the set (or approximately close to it), color
// the pixel black.
if(iter == max_iterations)
{
    *destPixel = 0;
}
// Otherwise, select a color that is based on the current iteration.
else
{
    BYTE red = static_cast<BYTE>((iter % 64) * 4);
    *destPixel = red<<16;
}

// Move to the next point.
++destPixel;
}
}

// Unlock the bitmap from system memory.
pBitmap->UnlockBits(&bitmapData);
}

```

8. Implement the `CChildView::OnPaint` method. This method calls `CChildView::DrawMandelbrot` and then copies the contents of the `Bitmap` object to the window.

```

void CChildView::OnPaint()
{
    CPaintDC dc(this); // device context for painting

    // Get the size of the client area of the window.
    RECT rc;
    GetClientRect(&rc);

    // Create a Bitmap object that has the width and height of
    // the client area.
    BitmapPtr pBitmap(new Bitmap(rc.right, rc.bottom));

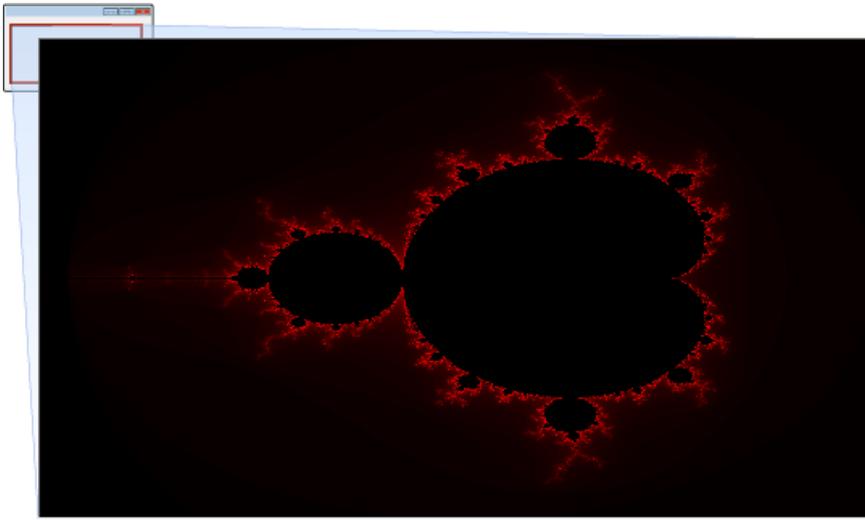
    if (pBitmap != NULL)
    {
        // Draw the Mandelbrot fractal to the bitmap.
        DrawMandelbrot(pBitmap);

        // Draw the bitmap to the client area.
        Graphics g(dc);
        g.DrawImage(pBitmap.get(), 0, 0);
    }
}

```

9. Verify that the application was updated successfully by building and running it.

The following illustration shows the results of the Mandelbrot application.



Because the computation for each pixel is computationally expensive, the UI thread cannot process additional messages until the overall computation finishes. This could decrease responsiveness in the application. However, you can relieve this problem by removing work from the UI thread.

[\[Top\]](#)

## Removing Work from the UI Thread

This section shows how to remove the drawing work from the UI thread in the Mandelbrot application. By moving drawing work from the UI thread to a worker thread, the UI thread can process messages as the worker thread generates the image in the background.

The Concurrency Runtime provides three ways to run tasks: [task groups](#), [asynchronous agents](#), and [lightweight tasks](#). Although you can use any one of these mechanisms to remove work from the UI thread, this example uses a [concurrency::task\\_group](#) object because task groups support cancellation. This walkthrough later uses cancellation to reduce the amount of work that is performed when the client window is resized, and to perform cleanup when the window is destroyed.

This example also uses a [concurrency::unbounded\\_buffer](#) object to enable the UI thread and the worker thread to communicate with each other. After the worker thread produces the image, it sends a pointer to the `Bitmap` object to the `unbounded_buffer` object and then posts a paint message to the UI thread. The UI thread then receives from the `unbounded_buffer` object the `Bitmap` object and draws it to the client window.

### To remove the drawing work from the UI thread

1. In `stdafx.h`, add the following `#include` directives:

```
#include <agents.h>
#include <ppl.h>
```

2. In `ChildView.h`, add `task_group` and `unbounded_buffer` member variables to the `protected` section of the `CChildView` class. The `task_group` object holds the tasks that perform drawing; the `unbounded_buffer` object holds the completed Mandelbrot image.

```
concurrency::task_group m_DrawingTasks;
concurrency::unbounded_buffer<BitmapPtr> m_MandelbrotImages;
```

3. In `ChildView.cpp`, add a `using` directive to the `concurrency` namespace.

```
using namespace concurrency;
```

4. In the `CChildView::DrawMandelbrot` method, after the call to `Bitmap::UnlockBits`, call the `concurrency::send` function to pass the `Bitmap` object to the UI thread. Then post a paint message to the UI thread and invalidate the client area.

```
// Unlock the bitmap from system memory.
pBitmap->UnlockBits(&bitmapData);

// Add the Bitmap object to image queue.
send(m_MandelbrotImages, pBitmap);

// Post a paint message to the UI thread.
PostMessage(WM_PAINT);
// Invalidate the client area.
InvalidateRect(NULL, FALSE);
```

5. Update the `CChildView::OnPaint` method to receive the updated `Bitmap` object and draw the image to the client window.

```
void CChildView::OnPaint()
{
    CPaintDC dc(this); // device context for painting

    // If the unbounded_buffer object contains a Bitmap object,
    // draw the image to the client area.
    BitmapPtr pBitmap;
    if (try_receive(m_MandelbrotImages, pBitmap))
    {
        if (pBitmap != NULL)
        {
            // Draw the bitmap to the client area.
            Graphics g(dc);
            g.DrawImage(pBitmap.get(), 0, 0);
        }
    }
    // Draw the image on a worker thread if the image is not available.
    else
    {
        RECT rc;
        GetClientRect(&rc);
        m_DrawingTasks.run([rc, this]() {
            DrawMandelbrot(BitmapPtr(new Bitmap(rc.right, rc.bottom)));
        });
    }
}
```

The `CChildView::OnPaint` method creates a task to generate the Mandelbrot image if one does not exist in the message buffer. The message buffer will not contain a `Bitmap` object in cases such as the initial paint message and when another window is moved in front of the client window.

6. Verify that the application was updated successfully by building and running it.

The UI is now more responsive because the drawing work is performed in the background.

[\[Top\]](#)

## Improving Drawing Performance

The generation of the Mandelbrot fractal is a good candidate for parallelization because the computation of each pixel is independent of all other computations. To parallelize the drawing procedure, convert the outer `for` loop in the `CChildView::DrawMandelbrot` method to a call to the `concurrency::parallel_for` algorithm, as follows.

```
// Compute whether each point lies in the Mandelbrot set.
parallel_for (0u, height, [&](UINT row)
{
    // Loop body omitted for brevity.
});
```

Because the computation of each bitmap element is independent, you do not have to synchronize the drawing operations that access the bitmap memory. This enables performance to scale as the number of available processors increases.

[\[Top\]](#)

## Adding Support for Cancellation

This section describes how to handle window resizing and how to cancel any active drawing tasks when the window is destroyed.

The document [Cancellation in the PPL](#) explains how cancellation works in the runtime. Cancellation is cooperative; therefore, it does not occur immediately. To stop a canceled task, the runtime throws an internal exception during a subsequent call from the task into the runtime. The previous section shows how to use the `parallel_for` algorithm to improve the performance of the drawing task. The call to `parallel_for` enables the runtime to stop the task, and therefore enables cancellation to work.

### Canceling Active Tasks

The Mandelbrot application creates `Bitmap` objects whose dimensions match the size of the client window. Every time the client window is resized, the application creates an additional background task to generate an image for the new window size. The application does not require these intermediate images; it requires only the image for the final window size. To prevent the application from performing this additional work, you can cancel any active drawing tasks in the message handlers for the `WM_SIZE` and `WM_SIZING` messages and then reschedule drawing work after the window is resized.

To cancel active drawing tasks when the window is resized, the application calls the `concurrency::task_group::cancel` method in the handlers for the `WM_SIZING` and `WM_SIZE` messages. The handler for the `WM_SIZE` message also calls the `concurrency::task_group::wait` method to wait for all active tasks to complete and then reschedules the drawing task for the updated window size.

When the client window is destroyed, it is good practice to cancel any active drawing tasks. Canceling any active drawing tasks makes sure that worker threads do not post messages to the UI thread after the client window is destroyed. The application cancels any active drawing tasks in the handler for the `WM_DESTROY` message.

### Responding to Cancellation

The `CChildView::DrawMandelbrot` method, which performs the drawing task, must respond to cancellation. Because the runtime uses exception handling to cancel tasks, the `CChildView::DrawMandelbrot` method must use an exception-safe mechanism to guarantee that all resources are correctly cleaned-up. This example uses the *Resource Acquisition Is Initialization* (RAII) pattern to guarantee that the bitmap bits are unlocked when the task is canceled.

To add support for cancellation in the Mandelbrot application

1. In `ChildView.h`, in the `protected` section of the `CChildView` class, add declarations for the `OnSize`, `OnSizing`, and `OnDestroy` message map functions.

```
afx_msg void OnPaint();
afx_msg void OnSize(UINT, int, int);
afx_msg void OnSizing(UINT, LPRECT);
afx_msg void OnDestroy();
DECLARE_MESSAGE_MAP()
```

2. In ChildView.cpp, modify the message map to contain handlers for the `WM_SIZE`, `WM_SIZING`, and `WM_DESTROY` messages.

```
BEGIN_MESSAGE_MAP(CChildView, CWnd)
    ON_WM_PAINT()
    ON_WM_SIZE()
    ON_WM_SIZING()
    ON_WM_DESTROY()
END_MESSAGE_MAP()
```

3. Implement the `CChildView::OnSizing` method. This method cancels any existing drawing tasks.

```
void CChildView::OnSizing(UINT nSide, LPRECT lpRect)
{
    // The window size is changing; cancel any existing drawing tasks.
    m_DrawingTasks.cancel();
}
```

4. Implement the `CChildView::OnSize` method. This method cancels any existing drawing tasks and creates a new drawing task for the updated client window size.

```
void CChildView::OnSize(UINT nType, int cx, int cy)
{
    // The window size has changed; cancel any existing drawing tasks.
    m_DrawingTasks.cancel();
    // Wait for any existing tasks to finish.
    m_DrawingTasks.wait();

    // If the new size is non-zero, create a task to draw the Mandelbrot
    // image on a separate thread.
    if (cx != 0 && cy != 0)
    {
        m_DrawingTasks.run([cx,cy,this]() {
            DrawMandelbrot(BitmapPtr(new Bitmap(cx, cy)));
        });
    }
}
```

5. Implement the `CChildView::OnDestroy` method. This method cancels any existing drawing tasks.

```
void CChildView::OnDestroy()
{
    // The window is being destroyed; cancel any existing drawing tasks.
    m_DrawingTasks.cancel();
    // Wait for any existing tasks to finish.
    m_DrawingTasks.wait();
}
```

6. In ChildView.cpp, define the `scope_guard` class, which implements the RAII pattern.

```

// Implements the Resource Acquisition Is Initialization (RAII) pattern
// by calling the specified function after leaving scope.
class scope_guard
{
public:
    explicit scope_guard(std::function<void()> f)
        : m_f(std::move(f)) { }

    // Dismisses the action.
    void dismiss() {
        m_f = nullptr;
    }

    ~scope_guard() {
        // Call the function.
        if (m_f) {
            try {
                m_f();
            }
            catch (...) {
                terminate();
            }
        }
    }
}

private:
    // The function to call when leaving scope.
    std::function<void()> m_f;

    // Hide copy constructor and assignment operator.
    scope_guard(const scope_guard&);
    scope_guard& operator=(const scope_guard&);
};

```

7. Add the following code to the `CChildView::DrawMandelbrot` method after the call to `Bitmap::LockBits`:

```

// Create a scope_guard object that unlocks the bitmap bits when it
// leaves scope. This ensures that the bitmap is properly handled
// when the task is canceled.
scope_guard guard([&pBitmap, &bitmapData] {
    // Unlock the bitmap from system memory.
    pBitmap->UnlockBits(&bitmapData);
});

```

This code handles cancellation by creating a `scope_guard` object. When the object leaves scope, it unlocks the bitmap bits.

8. Modify the end of the `CChildView::DrawMandelbrot` method to dismiss the `scope_guard` object after the bitmap bits are unlocked, but before any messages are sent to the UI thread. This ensures that the UI thread is not updated before the bitmap bits are unlocked.

```
// Unlock the bitmap from system memory.
pBitmap->UnlockBits(&bitmapData);

// Dismiss the scope guard because the bitmap has been
// properly unlocked.
guard.dismiss();

// Add the Bitmap object to image queue.
send(m_MandelbrotImages, pBitmap);

// Post a paint message to the UI thread.
PostMessage(WM_PAINT);
// Invalidate the client area.
InvalidateRect(NULL, FALSE);
```

9. Verify that the application was updated successfully by building and running it.

When you resize the window, drawing work is performed only for the final window size. Any active drawing tasks are also canceled when the window is destroyed.

[\[Top\]](#)

## See also

[Concurrency Runtime Walkthroughs](#)

[Task Parallelism](#)

[Asynchronous Message Blocks](#)

[Message Passing Functions](#)

[Parallel Algorithms](#)

[Cancellation in the PPL](#)

[MFC Desktop Applications](#)

# Walkthrough: Using the Concurrency Runtime in a COM-Enabled Application

4/25/2019 • 14 minutes to read • [Edit Online](#)

This document demonstrates how to use the Concurrency Runtime in an application that uses the Component Object Model (COM).

## Prerequisites

Read the following documents before you start this walkthrough:

- [Task Parallelism](#)
- [Parallel Algorithms](#)
- [Asynchronous Agents](#)
- [Exception Handling](#)

For more information about COM, see [Component Object Model \(COM\)](#).

## Managing the Lifetime of the COM Library

Although the use of COM with the Concurrency Runtime follows the same principles as any other concurrency mechanism, the following guidelines can help you use these libraries together effectively.

- A thread must call `CoInitializeEx` before it uses the COM library.
- A thread can call `CoInitializeEx` multiple times as long as it provides the same arguments to every call.
- For each call to `CoInitializeEx`, a thread must also call `CoUninitialize`. In other words, calls to `CoInitializeEx` and `CoUninitialize` must be balanced.
- To switch from one thread apartment to another, a thread must completely free the COM library before it calls `CoInitializeEx` with the new threading specification.

Other COM principles apply when you use COM with the Concurrency Runtime. For example, an application that creates an object in a single-threaded apartment (STA) and marshals that object to another apartment must also provide a message loop to process incoming messages. Also remember that marshaling objects between apartments can decrease performance.

### Using COM with the Parallel Patterns Library

When you use COM with a component in the Parallel Patterns Library (PPL), for example, a task group or parallel algorithm, call `CoInitializeEx` before you use the COM library during each task or iteration, and call `CoUninitialize` before each task or iteration finishes. The following example shows how to manage the lifetime of the COM library with a `concurrency::structured_task_group` object.

```

structured_task_group tasks;

// Create and run a task.
auto task = make_task([] {
    // Initialize the COM library on the current thread.
    CoInitializeEx(NULL, COINIT_MULTITHREADED);

    // TODO: Perform task here.

    // Free the COM library.
    CoUninitialize();
});
tasks.run(task);

// TODO: Run additional tasks here.

// Wait for the tasks to finish.
tasks.wait();

```

You must make sure that the COM library is correctly freed when a task or parallel algorithm is canceled or when the task body throws an exception. To guarantee that the task calls `CoUninitialize` before it exits, use a `try-finally` block or the *Resource Acquisition Is Initialization* (RAII) pattern. The following example uses a `try-finally` block to free the COM library when the task completes or is canceled, or when an exception is thrown.

```

structured_task_group tasks;

// Create and run a task.
auto task = make_task([] {
    bool coinit = false;
    __try {
        // Initialize the COM library on the current thread.
        CoInitializeEx(NULL, COINIT_MULTITHREADED);
        coinit = true;

        // TODO: Perform task here.
    }
    __finally {
        // Free the COM library.
        if (coinit)
            CoUninitialize();
    }
});
tasks.run(task);

// TODO: Run additional tasks here.

// Wait for the tasks to finish.
tasks.wait();

```

The following example uses the RAII pattern to define the `CCoInitializer` class, which manages the lifetime of the COM library in a given scope.

```

// An exception-safe wrapper class that manages the lifetime
// of the COM library in a given scope.
class CCoInitializer
{
public:
    explicit CCoInitializer(DWORD dwCoInit = COINIT_APARTMENTTHREADED)
        : _coinitialized(false)
    {
        // Initialize the COM library on the current thread.
        HRESULT hr = CoInitializeEx(NULL, dwCoInit);
        if (FAILED(hr))
            throw hr;
        _coinitialized = true;
    }
    ~CCoInitializer()
    {
        // Free the COM library.
        if (_coinitialized)
            CoUninitialize();
    }
private:
    // Flags whether COM was properly initialized.
    bool _coinitialized;

    // Hide copy constructor and assignment operator.
    CCoInitializer(const CCoInitializer&);
    CCoInitializer& operator=(const CCoInitializer&);
};

```

You can use the `CCoInitializer` class to automatically free the COM library when the task exits, as follows.

```

structured_task_group tasks;

// Create and run a task.
auto task = make_task([] {
    // Enable COM for the lifetime of the task.
    CCoInitializer coinit(COINIT_MULTITHREADED);

    // TODO: Perform task here.

    // The CCoInitializer object frees the COM library
    // when the task exits.
});
tasks.run(task);

// TODO: Run additional tasks here.

// Wait for the tasks to finish.
tasks.wait();

```

For more information about cancellation in the Concurrency Runtime, see [Cancellation in the PPL](#).

### Using COM with Asynchronous Agents

When you use COM with asynchronous agents, call `CoInitializeEx` before you use the COM library in the `concurrency::agent::run` method for your agent. Then call `CoUninitialize` before the `run` method returns. Do not use COM management routines in the constructor or destructor of your agent, and do not override the `concurrency::agent::start` or `concurrency::agent::done` methods because these methods are called from a different thread than the `run` method.

The following example shows a basic agent class, named `CCoAgent`, which manages the COM library in the `run` method.

```

class CCoAgent : public agent
{
protected:
    void run()
    {
        // Initialize the COM library on the current thread.
        CoInitializeEx(NULL, COINIT_MULTITHREADED);

        // TODO: Perform work here.

        // Free the COM library.
        CoUninitialize();

        // Set the agent to the finished state.
        done();
    }
};

```

A complete example is provided later in this walkthrough.

### Using COM with Lightweight Tasks

The document [Task Scheduler](#) describes the role of lightweight tasks in the Concurrency Runtime. You can use COM with a lightweight task just as you would with any thread routine that you pass to the `CreateThread` function in the Windows API. This is shown in the following example.

```

// A basic lightweight task that you schedule directly from a
// Scheduler or ScheduleGroup object.
void ThreadProc(void* data)
{
    // Initialize the COM library on the current thread.
    CoInitializeEx(NULL, COINIT_MULTITHREADED);

    // TODO: Perform work here.

    // Free the COM library.
    CoUninitialize();
}

```

## An Example of a COM-Enabled Application

This section shows a complete COM-enabled application that uses the `IScriptControl` interface to execute a script that computes the  $n^{\text{th}}$  Fibonacci number. This example first calls the script from the main thread, and then uses the PPL and agents to call the script concurrently.

Consider the following helper function, `RunScriptProcedure`, which calls a procedure in an `IScriptControl` object.

```

// Calls a procedure in an IScriptControl object.
template<size_t ArgCount>
_variant_t RunScriptProcedure(IScriptControlPtr pScriptControl,
    _bstr_t& procedureName, array<_variant_t, ArgCount>& arguments)
{
    // Create a 1-dimensional, 0-based safe array.
    SAFEARRAYBOUND rgsabound[] = { ArgCount, 0 };
    CComSafeArray<VARIANT> sa(rgsabound, 1U);

    // Copy the arguments to the safe array.
    LONG lIndex = 0;
    for_each(begin(arguments), end(arguments), [&](_variant_t& arg) {
        HRESULT hr = sa.SetAt(lIndex, arg);
        if (FAILED(hr))
            throw hr;
        ++lIndex;
    });

    // Call the procedure in the script.
    return pScriptControl->Run(procedureName, &sa.m_psa);
}

```

The `wmain` function creates an `IScriptControl` object, adds script code to it that computes the  $n^{\text{th}}$  Fibonacci number, and then calls the `RunScriptProcedure` function to run that script.

```

int wmain()
{
    HRESULT hr;

    // Enable COM on this thread for the lifetime of the program.
    CCoInitializer coinit(COINIT_MULTITHREADED);

    // Create the script control.
    IScriptControlPtr pScriptControl(__uuidof(ScriptControl));

    // Set script control properties.
    pScriptControl->Language = "JScript";
    pScriptControl->AllowUI = TRUE;

    // Add script code that computes the nth Fibonacci number.
    hr = pScriptControl->AddCode(
        "function fib(n) { if (n<2) return n; else return fib(n-1) + fib(n-2); }" );
    if (FAILED(hr))
        return hr;

    // Test the script control by computing the 15th Fibonacci number.
    wcout << endl << L"Main Thread:" << endl;
    LONG lValue = 15;
    array<_variant_t, 1> args = { _variant_t(lValue) };
    _variant_t result = RunScriptProcedure(
        pScriptControl,
        _bstr_t("fib"),
        args);
    // Print the result.
    wcout << L"fib(" << lValue << L") = " << result.lVal << endl;

    return S_OK;
}

```

### Calling the Script from the PPL

The following function, `ParallelFibonacci`, uses the `concurrency::parallel_for` algorithm to call the script in parallel. This function uses the `CCoInitializer` class to manage the lifetime of the COM library during every iteration of the task.

```

// Computes multiple Fibonacci numbers in parallel by using
// the parallel_for algorithm.
HRESULT ParallelFibonacci(IScriptControlPtr pScriptControl)
{
    try {
        parallel_for(10L, 20L, [&pScriptControl](LONG lIndex)
        {
            // Enable COM for the lifetime of the task.
            CCoInitializer coininit(COINIT_MULTITHREADED);

            // Call the helper function to run the script procedure.
            array<_variant_t, 1> args = { _variant_t(lIndex) };
            _variant_t result = RunScriptProcedure(
                pScriptControl,
                _bstr_t("fib"),
                args);

            // Print the result.
            wstringstream ss;
            ss << L"fib(" << lIndex << L") = " << result.lVal << endl;
            wcout << ss.str();
        });
    }
    catch (HRESULT hr) {
        return hr;
    }
    return S_OK;
}

```

To use the `ParallelFibonacci` function with the example, add the following code before the `wmain` function returns.

```

// Use the parallel_for algorithm to compute multiple
// Fibonacci numbers in parallel.
wcout << endl << L"Parallel Fibonacci:" << endl;
if (FAILED(hr = ParallelFibonacci(pScriptControl)))
    return hr;

```

### Calling the Script from an Agent

The following example shows the `FibonacciScriptAgent` class, which calls a script procedure to compute the  $n^{\text{th}}$  Fibonacci number. The `FibonacciScriptAgent` class uses message passing to receive, from the main program, input values to the script function. The `run` method manages the lifetime of the COM library throughout the task.

```

// A basic agent that calls a script procedure to compute the
// nth Fibonacci number.
class FibonacciScriptAgent : public agent
{
public:
    FibonacciScriptAgent(IScriptControlPtr pScriptControl, ISource<LONG>& source)
        : _pScriptControl(pScriptControl)
        , _source(source) { }

public:
    // Retrieves the result code.
    HRESULT GetHRESULT()
    {
        return receive(_result);
    }

protected:
    void run()
    {
        // Enable COM for the lifetime of the task.
    }
}

```

```

// Initialize the COM library on the current thread.
CoInitializeEx(NULL, COINIT_MULTITHREADED);

// Read values from the message buffer until
// we receive the sentinel value.
LONG lValue;
while ((lValue = receive(_source)) != Sentinel)
{
    try {
        // Call the helper function to run the script procedure.
        array<_variant_t, 1> args = { _variant_t(lValue) };
        _variant_t result = RunScriptProcedure(
            _pScriptControl,
            _bstr_t("fib"),
            args);

        // Print the result.
        wstringstream ss;
        ss << L"fib(" << lValue << L") = " << result.lVal << endl;
        wcout << ss.str();
    }
    catch (HRESULT hr) {
        send(_result, hr);
        break;
    }
}

// Set the result code (does nothing if a value is already set).
send(_result, S_OK);

// Free the COM library.
CoUninitialize();

// Set the agent to the finished state.
done();
}

public:
    // Signals the agent to terminate.
    static const LONG Sentinel = 0L;

private:
    // The IScriptControl object that contains the script procedure.
    IScriptControlPtr _pScriptControl;
    // Message buffer from which to read arguments to the
    // script procedure.
    ISource<LONG>& _source;
    // The result code for the overall operation.
    single_assignment<HRESULT> _result;
};

```

The following function, `AgentFibonacci`, creates several `FibonacciScriptAgent` objects and uses message passing to send several input values to those objects.

```

// Computes multiple Fibonacci numbers in parallel by using
// asynchronous agents.
HRESULT AgentFibonacci(IScriptControlPtr pScriptControl)
{
    // Message buffer to hold arguments to the script procedure.
    unbounded_buffer<LONG> values;

    // Create several agents.
    array<agent*, 3> agents =
    {
        new FibonacciScriptAgent(pScriptControl, values),
        new FibonacciScriptAgent(pScriptControl, values),
        new FibonacciScriptAgent(pScriptControl, values),
    };

    // Start each agent.
    for_each(begin(agents), end(agents), [](agent* a) {
        a->start();
    });

    // Send a few values to the agents.
    send(values, 30L);
    send(values, 22L);
    send(values, 10L);
    send(values, 12L);
    // Send a sentinel value to each agent.
    for_each(begin(agents), end(agents), [&values](agent*) {
        send(values, FibonacciScriptAgent::Sentinel);
    });

    // Wait for all agents to finish.
    agent::wait_for_all(3, &agents[0]);

    // Determine the result code.
    HRESULT hr = S_OK;
    for_each(begin(agents), end(agents), [&hr](agent* a) {
        HRESULT hrTemp;
        if (FAILED(hrTemp =
            reinterpret_cast<FibonacciScriptAgent*>(a)->GetHRESULT()))
        {
            hr = hrTemp;
        }
    });

    // Clean up.
    for_each(begin(agents), end(agents), [](agent* a) {
        delete a;
    });

    return hr;
}

```

To use the `AgentFibonacci` function with the example, add the following code before the `wmain` function returns.

```

// Use asynchronous agents to compute multiple
// Fibonacci numbers in parallel.
wcout << endl << L"Agent Fibonacci:" << endl;
if (FAILED(hr = AgentFibonacci(pScriptControl)))
    return hr;

```

## The Complete Example

The following code shows the complete example, which uses parallel algorithms and asynchronous agents to call a script procedure that computes Fibonacci numbers.

```

// parallel-scripts.cpp
// compile with: /EHsc

#include <agents.h>
#include <ppl.h>
#include <array>
#include <sstream>
#include <iostream>
#include <atlsafe.h>

// TODO: Change this path if necessary.
#import "C:\windows\system32\msscript.ocx"

using namespace concurrency;
using namespace MSScriptControl;
using namespace std;

// An exception-safe wrapper class that manages the lifetime
// of the COM library in a given scope.
class CCoInitializer
{
public:
    explicit CCoInitializer(DWORD dwCoInit = COINIT_APARTMENTTHREADED)
        : _coinitialized(false)
    {
        // Initialize the COM library on the current thread.
        HRESULT hr = CoInitializeEx(NULL, dwCoInit);
        if (FAILED(hr))
            throw hr;
        _coinitialized = true;
    }
    ~CCoInitializer()
    {
        // Free the COM library.
        if (_coinitialized)
            CoUninitialize();
    }
private:
    // Flags whether COM was properly initialized.
    bool _coinitialized;

    // Hide copy constructor and assignment operator.
    CCoInitializer(const CCoInitializer&);
    CCoInitializer& operator=(const CCoInitializer&);
};

// Calls a procedure in an IScriptControl object.
template<size_t ArgCount>
_variant_t RunScriptProcedure(IScriptControlPtr pScriptControl,
    _bstr_t& procedureName, array<_variant_t, ArgCount>& arguments)
{
    // Create a 1-dimensional, 0-based safe array.
    SAFEARRAYBOUND rgsabound[] = { ArgCount, 0 };
    CComSafeArray<VARIANT> sa(rgsabound, 1U);

    // Copy the arguments to the safe array.
    LONG lIndex = 0;
    for_each(begin(arguments), end(arguments), [&]( _variant_t& arg ) {
        HRESULT hr = sa.SetAt(lIndex, arg);
        if (FAILED(hr))
            throw hr;
        ++lIndex;
    });

    // Call the procedure in the script.
    return pScriptControl->Run(procedureName, &sa.m_psa);
}

```

```

// Computes multiple Fibonacci numbers in parallel by using
// the parallel_for algorithm.
HRESULT ParallelFibonacci(IScriptControlPtr pScriptControl)
{
    try {
        parallel_for(10L, 20L, [&pScriptControl](LONG lIndex)
        {
            // Enable COM for the lifetime of the task.
            CoInitializer coininit(COINIT_MULTITHREADED);

            // Call the helper function to run the script procedure.
            array<_variant_t, 1> args = { _variant_t(lIndex) };
            _variant_t result = RunScriptProcedure(
                pScriptControl,
                _bstr_t("fib"),
                args);

            // Print the result.
            wstringstream ss;
            ss << L"fib(" << lIndex << L") = " << result.lVal << endl;
            wcout << ss.str();
        });
    }
    catch (HRESULT hr) {
        return hr;
    }
    return S_OK;
}

// A basic agent that calls a script procedure to compute the
// nth Fibonacci number.
class FibonacciScriptAgent : public agent
{
public:
    FibonacciScriptAgent(IScriptControlPtr pScriptControl, ISource<LONG>& source)
        : _pScriptControl(pScriptControl)
        , _source(source) { }

public:
    // Retrieves the result code.
    HRESULT GetHRESULT()
    {
        return receive(_result);
    }

protected:
    void run()
    {
        // Initialize the COM library on the current thread.
        CoInitializeEx(NULL, COINIT_MULTITHREADED);

        // Read values from the message buffer until
        // we receive the sentinel value.
        LONG lValue;
        while ((lValue = receive(_source)) != Sentinel)
        {
            try {
                // Call the helper function to run the script procedure.
                array<_variant_t, 1> args = { _variant_t(lValue) };
                _variant_t result = RunScriptProcedure(
                    _pScriptControl,
                    _bstr_t("fib"),
                    args);

                // Print the result.
                wstringstream ss;
                ss << L"fib(" << lValue << L") = " << result.lVal << endl;
                wcout << ss.str();
            }

```

```

        catch (HRESULT hr) {
            send(_result, hr);
            break;
        }
    }

    // Set the result code (does nothing if a value is already set).
    send(_result, S_OK);

    // Free the COM library.
    CoUninitialize();

    // Set the agent to the finished state.
    done();
}

public:
    // Signals the agent to terminate.
    static const LONG Sentinel = 0L;

private:
    // The IScriptControl object that contains the script procedure.
    IScriptControlPtr _pScriptControl;
    // Message buffer from which to read arguments to the
    // script procedure.
    ISource<LONG>& _source;
    // The result code for the overall operation.
    single_assignment<HRESULT> _result;
};

// Computes multiple Fibonacci numbers in parallel by using
// asynchronous agents.
HRESULT AgentFibonacci(IScriptControlPtr pScriptControl)
{
    // Message buffer to hold arguments to the script procedure.
    unbounded_buffer<LONG> values;

    // Create several agents.
    array<agent*, 3> agents =
    {
        new FibonacciScriptAgent(pScriptControl, values),
        new FibonacciScriptAgent(pScriptControl, values),
        new FibonacciScriptAgent(pScriptControl, values),
    };

    // Start each agent.
    for_each(begin(agents), end(agents), [](agent* a) {
        a->start();
    });

    // Send a few values to the agents.
    send(values, 30L);
    send(values, 22L);
    send(values, 10L);
    send(values, 12L);
    // Send a sentinel value to each agent.
    for_each(begin(agents), end(agents), [&values](agent* a) {
        send(values, FibonacciScriptAgent::Sentinel);
    });

    // Wait for all agents to finish.
    agent::wait_for_all(3, &agents[0]);

    // Determine the result code.
    HRESULT hr = S_OK;
    for_each(begin(agents), end(agents), [&hr](agent* a) {
        HRESULT hrTemp;
        if (FAILED(hrTemp =
            reinterpret_cast<FibonacciScriptAgent*>(a)->GetHRESULT()))

```

```

        {
            hr = hrTemp;
        }
    });

    // Clean up.
    for_each(begin(agents), end(agents), [](agent* a) {
        delete a;
    });

    return hr;
}

int wmain()
{
    HRESULT hr;

    // Enable COM on this thread for the lifetime of the program.
    CCoInitializer coinit(COINIT_MULTITHREADED);

    // Create the script control.
    IScriptControlPtr pScriptControl(__uuidof(ScriptControl));

    // Set script control properties.
    pScriptControl->Language = "JScript";
    pScriptControl->AllowUI = TRUE;

    // Add script code that computes the nth Fibonacci number.
    hr = pScriptControl->AddCode(
        "function fib(n) { if (n<2) return n; else return fib(n-1) + fib(n-2); }" );
    if (FAILED(hr))
        return hr;

    // Test the script control by computing the 15th Fibonacci number.
    wcout << L"Main Thread:" << endl;
    long n = 15;
    array<variant_t, 1> args = { variant_t(n) };
    _variant_t result = RunScriptProcedure(
        pScriptControl,
        _bstr_t("fib"),
        args);
    // Print the result.
    wcout << L"fib(" << n << L") = " << result.lVal << endl;

    // Use the parallel_for algorithm to compute multiple
    // Fibonacci numbers in parallel.
    wcout << endl << L"Parallel Fibonacci:" << endl;
    if (FAILED(hr = ParallelFibonacci(pScriptControl)))
        return hr;

    // Use asynchronous agents to compute multiple
    // Fibonacci numbers in parallel.
    wcout << endl << L"Agent Fibonacci:" << endl;
    if (FAILED(hr = AgentFibonacci(pScriptControl)))
        return hr;

    return S_OK;
}

```

The example produces the following sample output.

```
Main Thread:  
fib(15) = 610
```

```
Parallel Fibonacci:  
fib(15) = 610  
fib(10) = 55  
fib(16) = 987  
fib(18) = 2584  
fib(11) = 89  
fib(17) = 1597  
fib(19) = 4181  
fib(12) = 144  
fib(13) = 233  
fib(14) = 377
```

```
Agent Fibonacci:  
fib(30) = 832040  
fib(22) = 17711  
fib(10) = 55  
fib(12) = 144
```

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste it in a file that is named `parallel-scripts.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc parallel-scripts.cpp /link ole32.lib
```

## See also

[Concurrency Runtime Walkthroughs](#)

[Task Parallelism](#)

[Parallel Algorithms](#)

[Asynchronous Agents](#)

[Exception Handling](#)

[Cancellation in the PPL](#)

[Task Scheduler](#)

# Walkthrough: Adapting Existing Code to Use Lightweight Tasks

4/25/2019 • 3 minutes to read • [Edit Online](#)

This topic shows how to adapt existing code that uses the Windows API to create and execute a thread to use a lightweight task.

A *lightweight task* is a task that you schedule directly from a [concurrency::Scheduler](#) or [concurrency::ScheduleGroup](#) object. Lightweight tasks are useful when you adapt existing code to use the scheduling functionality of the Concurrency Runtime.

## Prerequisites

Before you start this walkthrough, read the topic [Task Scheduler](#).

## Example

### Description

The following example illustrates typical usage of the Windows API to create and execute a thread. This example uses the [CreateThread](#) function to call the `MyThreadFunction` on a separate thread.

### Code

```
// windows-threads.cpp
#include <windows.h>
#include <tchar.h>
#include <strsafe.h>

#define BUF_SIZE 255

DWORD WINAPI MyThreadFunction(LPVOID param);

// Data structure for threads to use.
typedef struct MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;

int _tmain()
{
    // Allocate memory for thread data.
    PMYDATA pData = (PMYDATA) HeapAlloc(GetProcessHeap(),
        HEAP_ZERO_MEMORY, sizeof(MYDATA));

    if( pData == NULL )
    {
        ExitProcess(2);
    }

    // Set the values of the thread data.
    pData->val1 = 50;
    pData->val2 = 100;

    // Create the thread to begin execution on its own.
    DWORD dwThreadId;
    HANDLE hThread = CreateThread(
        NULL, // default security attributes
```

```

    0,                // use default stack size
    MyThreadFunction, // thread function name
    pData,           // argument to thread function
    0,               // use default creation flags
    &dwThreadId);    // returns the thread identifier

if (hThread == NULL)
{
    ExitProcess(3);
}

// Wait for the thread to finish.
WaitForSingleObject(hThread, INFINITE);

// Close the thread handle and free memory allocation.
CloseHandle(hThread);
HeapFree(GetProcessHeap(), 0, pData);

return 0;
}

DWORD WINAPI MyThreadFunction(LPVOID lpParam)
{
    PMYDATA pData = (PMYDATA)lpParam;

    // Use thread-safe functions to print the parameter values.

    TCHAR msgBuf[BUF_SIZE];
    StringCchPrintf(msgBuf, BUF_SIZE, TEXT("Parameters = %d, %d\n"),
        pData->val1, pData->val2);

    size_t cchStringSize;
    StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);

    DWORD dwChars;
    WriteConsole(GetStdHandle(STD_OUTPUT_HANDLE), msgBuf, (DWORD)cchStringSize, &dwChars, NULL);

    return 0;
}

```

## Comments

This example produces the following output.

```
Parameters = 50, 100
```

The following steps show how to adapt the code example to use the Concurrency Runtime to perform the same task.

### To adapt the example to use a lightweight task

1. Add a `#include` directive for the header file `concrct.h`.

```
#include <concrct.h>
```

1. Add a `using` directive for the `concurrency` namespace.

```
using namespace concurrency;
```

1. Change the declaration of `MyThreadFunction` to use the `__cdecl` calling convention and to return `void`.

```
void __cdecl MyThreadFunction(LPVOID param);
```

1. Modify the `MyData` structure to include a `concurrency::event` object that signals to the main application that the task has finished.

```
typedef struct MyData {  
    int val1;  
    int val2;  
    event signal;  
} MYDATA, *PMYDATA;
```

1. Replace the call to `CreateThread` with a call to the `concurrency::CurrentScheduler::ScheduleTask` method.

```
CurrentScheduler::ScheduleTask(MyThreadFunction, pData);
```

1. Replace the call to `WaitForSingleObject` with a call to the `concurrency::event::wait` method to wait for the task to finish.

```
// Wait for the task to finish.  
pData->signal.wait();
```

1. Remove the call to `CloseHandle`.
2. Change the signature of the definition of `MyThreadFunction` to match step 3.

```
void __cdecl MyThreadFunction(LPVOID lpParam)
```

9. At the end of the `MyThreadFunction` function, call the `concurrency::event::set` method to signal to the main application that the task has finished.

```
pData->signal.set();
```

10. Remove the `return` statement from `MyThreadFunction`.

## Example

### Description

The following completed example shows code that uses a lightweight task to call the `MyThreadFunction` function.

### Code

```

// migration-lwt.cpp
// compile with: /EHsc
#include <windows.h>
#include <tchar.h>
#include <strsafe.h>
#include <concr.h>

using namespace concurrency;

#define BUF_SIZE 255

void __cdecl MyThreadFunction(LPVOID param);

// Data structure for threads to use.
typedef struct MyData {
    int val1;
    int val2;
    event signal;
} MYDATA, *PMYDATA;

int _tmain()
{
    // Allocate memory for thread data.
    PMYDATA pData = (PMYDATA) HeapAlloc(GetProcessHeap(),
        HEAP_ZERO_MEMORY, sizeof(MYDATA));

    if( pData == NULL )
    {
        ExitProcess(2);
    }

    // Set the values of the thread data.
    pData->val1 = 50;
    pData->val2 = 100;

    // Create the thread to begin execution on its own.
    CurrentScheduler::ScheduleTask(MyThreadFunction, pData);

    // Wait for the task to finish.
    pData->signal.wait();

    // Free memory allocation.
    HeapFree(GetProcessHeap(), 0, pData);

    return 0;
}

void __cdecl MyThreadFunction(LPVOID lpParam)
{
    PMYDATA pData = (PMYDATA)lpParam;

    // Use thread-safe functions to print the parameter values.

    TCHAR msgBuf[BUF_SIZE];
    StringCchPrintf(msgBuf, BUF_SIZE, TEXT("Parameters = %d, %d\n"),
        pData->val1, pData->val2);

    size_t cchStringSize;
    StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);

    DWORD dwChars;
    WriteConsole(GetStdHandle(STD_OUTPUT_HANDLE), msgBuf, (DWORD)cchStringSize, &dwChars, NULL);

    pData->signal.set();
}

```

## Comments

## See also

[Task Scheduler](#)  
[Scheduler Class](#)

# Walkthrough: Creating a Custom Message Block

4/25/2019 • 23 minutes to read • [Edit Online](#)

This document describes how to create a custom message block type that orders incoming messages by priority.

Although the built-in message block types provide a wide-range of functionality, you can create your own message block type and customize it to meet the requirements of your application. For a description of the built-in message block types that are provided by the Asynchronous Agents Library, see [Asynchronous Message Blocks](#).

## Prerequisites

Read the following documents before you start this walkthrough:

- [Asynchronous Message Blocks](#)
- [Message Passing Functions](#)

## Sections

This walkthrough contains the following sections:

- [Designing a Custom Message Block](#)
- [Defining the `priority\_buffer` Class](#)
- [The Complete Example](#)

## Designing a Custom Message Block

Message blocks participate in the act of sending and receiving messages. A message block that sends messages is known as a *source block*. A message block that receives messages is known as a *target block*. A message block that both sends and receives messages is known as a *propagator block*. The Agents Library uses the abstract class `concurrency::ISource` to represent source blocks and the abstract class `concurrency::ITarget` to represent target blocks. Message block types that act as sources derive from `ISource`; message block types that act as targets derive from `ITarget`.

Although you can derive your message block type directly from `ISource` and `ITarget`, the Agents Library defines three base classes that perform much of the functionality that is common to all message block types, for example, handling errors and connecting message blocks together in a concurrency-safe manner. The `concurrency::source_block` class derives from `ISource` and sends messages to other blocks. The `concurrency::target_block` class derives from `ITarget` and receives messages from other blocks. The `concurrency::propagator_block` class derives from `ISource` and `ITarget` and sends messages to other blocks and it receives messages from other blocks. We recommend that you use these three base classes to handle infrastructure details so that you can focus on the behavior of your message block.

The `source_block`, `target_block`, and `propagator_block` classes are templates that are parameterized on a type that manages the connections, or links, between source and target blocks and on a type that manages how messages are processed. The Agents Library defines two types that perform link management, `concurrency::single_link_registry` and `concurrency::multi_link_registry`. The `single_link_registry` class enables a message block to be linked to one source or to one target. The `multi_link_registry` class enables a message block to be linked to multiple sources or multiple targets. The Agents Library defines one class that performs message management, `concurrency::ordered_message_processor`. The `ordered_message_processor` class enables message

blocks to process messages in the order in which it receives them.

To better understand how message blocks relate to their sources and targets, consider the following example. This example shows the declaration of the `concurrency::transformer` class.

```
template<
    class _Input,
    class _Output
>
class transformer : public propagator_block<
    single_link_registry<ITarget<_Output>>,
    multi_link_registry<ISource<_Input>>
>;
```

The `transformer` class derives from `propagator_block`, and therefore acts as both a source block and as a target block. It accepts messages of type `_Input` and sends messages of type `_Output`. The `transformer` class specifies `single_link_registry` as the link manager for any target blocks and `multi_link_registry` as the link manager for any source blocks. Therefore, a `transformer` object can have up to one target and an unlimited number of sources.

A class that derives from `source_block` must implement six methods: `propagate_to_any_targets`, `accept_message`, `reserve_message`, `consume_message`, `release_message`, and `resume_propagation`. A class that derives from `target_block` must implement the `propagate_message` method and can optionally implement the `send_message` method. Deriving from `propagator_block` is functionally equivalent to deriving from both `source_block` and `target_block`.

The `propagate_to_any_targets` method is called by the runtime to asynchronously or synchronously process any incoming messages and propagate out any outgoing messages. The `accept_message` method is called by target blocks to accept messages. Many message block types, such as `unbounded_buffer`, send messages only to the first target that would receive it. Therefore, it transfers ownership of the message to the target. Other message block types, such as `concurrency::overwrite_buffer`, offer messages to each of its target blocks. Therefore, `overwrite_buffer` creates a copy of the message for each of its targets.

The `reserve_message`, `consume_message`, `release_message`, and `resume_propagation` methods enable message blocks to participate in message reservation. Target blocks call the `reserve_message` method when they are offered a message and have to reserve the message for later use. After a target block reserves a message, it can call the `consume_message` method to consume that message or the `release_message` method to cancel the reservation. As with the `accept_message` method, the implementation of `consume_message` can either transfer ownership of the message or return a copy of the message. After a target block either consumes or releases a reserved message, the runtime calls the `resume_propagation` method. Typically, this method continues message propagation, starting with the next message in the queue.

The runtime calls the `propagate_message` method to asynchronously transfer a message from another block to the current one. The `send_message` method resembles `propagate_message`, except that it synchronously, instead of asynchronously, sends the message to the target blocks. The default implementation of `send_message` rejects all incoming messages. The runtime does not call either of these methods if the message does not pass the optional filter function that is associated with the target block. For more information about message filters, see [Asynchronous Message Blocks](#).

[\[Top\]](#)

## Defining the `priority_buffer` Class

The `priority_buffer` class is a custom message block type that orders incoming messages first by priority, and then by the order in which messages are received. The `priority_buffer` class resembles the `concurrency::unbounded_buffer` class because it holds a queue of messages, and also because it acts as both a

source and a target message block and can have both multiple sources and multiple targets. However, `unbounded_buffer` bases message propagation only on the order in which it receives messages from its sources.

The `priority_buffer` class receives messages of type `std::tuple` that contain `PriorityType` and `Type` elements. `PriorityType` refers to the type that holds the priority of each message; `Type` refers to the data portion of the message. The `priority_buffer` class sends messages of type `Type`. The `priority_buffer` class also manages two message queues: a `std::priority_queue` object for incoming messages and a `std::queue` object for outgoing messages. Ordering messages by priority is useful when a `priority_buffer` object receives multiple messages simultaneously or when it receives multiple messages before any messages are read by consumers.

In addition to the seven methods that a class that derives from `propagator_block` must implement, the `priority_buffer` class also overrides the `link_target_notification` and `send_message` methods. The `priority_buffer` class also defines two public helper methods, `enqueue` and `dequeue`, and a private helper method, `propagate_priority_order`.

The following procedure describes how to implement the `priority_buffer` class.

#### To define the `priority_buffer` class

1. Create a C++ header file and name it `priority_buffer.h`. Alternatively, you can use an existing header file that is part of your project.
2. In `priority_buffer.h`, add the following code.

```
#pragma once
#include <agents.h>
#include <queue>
```

1. In the `std` namespace, define specializations of `std::less` and `std::greater` that act on `concurrency::message` objects.

```

namespace std
{
    // A specialization of less that tests whether the priority element of a
    // message is less than the priority element of another message.
    template<class Type, class PriorityType>
    struct less<concurrency::message<tuple<PriorityType,Type>>*>
    {
        typedef concurrency::message<tuple<PriorityType, Type>> MessageType;

        bool operator()(const MessageType* left, const MessageType* right) const
        {
            // apply operator< to the first element (the priority)
            // of the tuple payload.
            return (get<0>(left->payload) < get<0>(right->payload));
        }
    };

    // A specialization of less that tests whether the priority element of a
    // message is greater than the priority element of another message.
    template<class Type, class PriorityType>
    struct greater<concurrency::message<tuple<PriorityType, Type>>*>
    {
        typedef concurrency::message<std::tuple<PriorityType,Type>> MessageType;

        bool operator()(const MessageType* left, const MessageType* right) const
        {
            // apply operator> to the first element (the priority)
            // of the tuple payload.
            return (get<0>(left->payload) > get<0>(right->payload));
        }
    };
}

```

The `priority_buffer` class stores `message` objects in a `priority_queue` object. These type specializations enable the priority queue to sort messages according to their priority. The priority is the first element of the `tuple` object.

1. In the `concurrencyex` namespace, declare the `priority_buffer` class.

```

namespace concurrencyex
{
    template<class Type,
            typename PriorityType = int,
            typename Pr = std::less<message<std::tuple<PriorityType, Type>>*>>
    class priority_buffer : public
        concurrency::propagator_block<concurrency::multi_link_registry<concurrency::ITarget<Type>>,
            concurrency::multi_link_registry<concurrency::ISource<std::tuple<PriorityType, Type>>>>
    {
    public:
    protected:
    private:
    };
}

```

The `priority_buffer` class derives from `propagator_block`. Therefore, it can both send and receive messages. The `priority_buffer` class can have multiple targets that receive messages of type `Type`. It can also have multiple sources that send messages of type `tuple<PriorityType, Type>`.

1. In the `private` section of the `priority_buffer` class, add the following member variables.

```

// Stores incoming messages.
// The type parameter Pr specifies how to order messages by priority.
std::priority_queue<
    concurrency::message<_Source_type>*,
    std::vector<concurrency::message<_Source_type>*>,
    Pr
> _input_messages;

// Synchronizes access to the input message queue.
concurrency::critical_section _input_lock;

// Stores outgoing messages.
std::queue<concurrency::message<_Target_type>*> _output_messages;

```

The `priority_queue` object holds incoming messages; the `queue` object holds outgoing messages. A `priority_buffer` object can receive multiple messages simultaneously; the `critical_section` object synchronizes access to the queue of input messages.

1. In the `private` section, define the copy constructor and the assignment operator. This prevents `priority_queue` objects from being assignable.

```

// Hide assignment operator and copy constructor.
priority_buffer const &operator =(priority_buffer const&);
priority_buffer(priority_buffer const &);

```

1. In the `public` section, define the constructors that are common to many message block types. Also define the destructor.

```

// Constructs a priority_buffer message block.
priority_buffer()
{
    initialize_source_and_target();
}

// Constructs a priority_buffer message block with the given filter function.
priority_buffer(filter_method const& filter)
{
    initialize_source_and_target();
    register_filter(filter);
}

// Constructs a priority_buffer message block that uses the provided
// Scheduler object to propagate messages.
priority_buffer(concurrency::Scheduler& scheduler)
{
    initialize_source_and_target(&scheduler);
}

// Constructs a priority_buffer message block with the given filter function
// and uses the provided Scheduler object to propagate messages.
priority_buffer(concurrency::Scheduler& scheduler, filter_method const& filter)
{
    initialize_source_and_target(&scheduler);
    register_filter(filter);
}

// Constructs a priority_buffer message block that uses the provided
// SchedulerGroup object to propagate messages.
priority_buffer(concurrency::ScheduleGroup& schedule_group)
{
    initialize_source_and_target(NULL, &schedule_group);
}

// Constructs a priority_buffer message block with the given filter function
// and uses the provided SchedulerGroup object to propagate messages.
priority_buffer(concurrency::ScheduleGroup& schedule_group, filter_method const& filter)
{
    initialize_source_and_target(NULL, &schedule_group);
    register_filter(filter);
}

// Destroys the message block.
~priority_buffer()
{
    // Remove all links.
    remove_network_links();
}

```

1. In the `public` section, define the methods `enqueue` and `dequeue`. These helper methods provide an alternative way to send messages to and receive messages from a `priority_buffer` object.

```

// Sends an item to the message block.
bool enqueue(Type const& item)
{
    return concurrency::asend<Type>(this, item);
}

// Receives an item from the message block.
Type dequeue()
{
    return receive<Type>(this);
}

```

9. In the `protected` section, define the `propagate_to_any_targets` method.

```
// Transfers the message at the front of the input queue to the output queue
// and propagates out all messages in the output queue.
virtual void propagate_to_any_targets(concurrency::message<_Target_type>*)
{
    // Retrieve the message from the front of the input queue.
    concurrency::message<_Source_type>* input_message = NULL;
    {
        concurrency::critical_section::scoped_lock lock(_input_lock);
        if (_input_messages.size() > 0)
        {
            input_message = _input_messages.top();
            _input_messages.pop();
        }
    }

    // Move the message to the output queue.
    if (input_message != NULL)
    {
        // The payload of the output message does not contain the
        // priority of the message.
        concurrency::message<_Target_type>* output_message =
            new concurrency::message<_Target_type>(get<1>(input_message->payload));
        _output_messages.push(output_message);

        // Free the memory for the input message.
        delete input_message;

        // Do not propagate messages if the new message is not the head message.
        // In this case, the head message is reserved by another message block.
        if (_output_messages.front()->msg_id() != output_message->msg_id())
        {
            return;
        }
    }

    // Propagate out the output messages.
    propagate_priority_order();
}
```

The `propagate_to_any_targets` method transfers the message that is at the front of the input queue to the output queue and propagates out all messages in the output queue.

10. In the `protected` section, define the `accept_message` method.

```
// Accepts a message that was offered by this block by transferring ownership
// to the caller.
virtual concurrency::message<_Target_type>* accept_message(concurrency::runtime_object_identity msg_id)
{
    concurrency::message<_Target_type>* message = NULL;

    // Transfer ownership if the provided message identifier matches
    // the identifier of the front of the output message queue.
    if (!_output_messages.empty() &&
        _output_messages.front()->msg_id() == msg_id)
    {
        message = _output_messages.front();
        _output_messages.pop();
    }

    return message;
}
```

When a target block calls the `accept_message` method, the `priority_buffer` class transfers ownership of the message to the first target block that accepts it. (This resembles the behavior of `unbounded_buffer`.)

11. In the `protected` section, define the `reserve_message` method.

```
// Reserves a message that was previously offered by this block.
virtual bool reserve_message(concurrency::runtime_object_identity msg_id)
{
    // Allow the message to be reserved if the provided message identifier
    // is the message identifier of the front of the message queue.
    return (!_output_messages.empty() &&
        _output_messages.front()->msg_id() == msg_id);
}
```

The `priority_buffer` class permits a target block to reserve a message when the provided message identifier matches the identifier of the message that is at the front of the queue. In other words, a target can reserve the message if the `priority_buffer` object has not yet received an additional message and has not yet propagated out the current one.

12. In the `protected` section, define the `consume_message` method.

```
// Transfers the message that was previously offered by this block
// to the caller. The caller of this method is the target block that
// reserved the message.
virtual concurrency::message<Type>* consume_message(concurrency::runtime_object_identity msg_id)
{
    // Transfer ownership of the message to the caller.
    return accept_message(msg_id);
}
```

A target block calls `consume_message` to transfer ownership of the message that it reserved.

13. In the `protected` section, define the `release_message` method.

```
// Releases a previous message reservation.
virtual void release_message(concurrency::runtime_object_identity msg_id)
{
    // The head message must be the one that is reserved.
    if (_output_messages.empty() ||
        _output_messages.front()->msg_id() != msg_id)
    {
        throw message_not_found();
    }
}
```

A target block calls `release_message` to cancel its reservation to a message.

14. In the `protected` section, define the `resume_propagation` method.

```
// Resumes propagation after a reservation has been released.
virtual void resume_propagation()
{
    // Propagate out any messages in the output queue.
    if (_output_messages.size() > 0)
    {
        async_send(NULL);
    }
}
```

The runtime calls `resume_propagation` after a target block either consumes or releases a reserved message. This method propagates out any messages that are in the output queue.

15. In the `protected` section, define the `link_target_notification` method.

```
// Notifies this block that a new target has been linked to it.
virtual void link_target_notification(concurrency::ITarget<_Target_type>*)
{
    // Do not propagate messages if a target block reserves
    // the message at the front of the queue.
    if (_M_pReservedFor != NULL)
    {
        return;
    }

    // Propagate out any messages that are in the output queue.
    propagate_priority_order();
}
```

The `_M_pReservedFor` member variable is defined by the base class, `source_block`. This member variable points to the target block, if any, that is holding a reservation to the message that is at the front of the output queue. The runtime calls `link_target_notification` when a new target is linked to the `priority_buffer` object. This method propagates out any messages that are in the output queue if no target is holding a reservation.

16. In the `private` section, define the `propagate_priority_order` method.

```

// Propagates messages in priority order.
void propagate_priority_order()
{
    // Cancel propagation if another block reserves the head message.
    if (_M_pReservedFor != NULL)
    {
        return;
    }

    // Propagate out all output messages.
    // Because this block preserves message ordering, stop propagation
    // if any of the messages are not accepted by a target block.
    while (!_output_messages.empty())
    {
        // Get the next message.
        concurrency::message<_Target_type> * message = _output_messages.front();

        concurrency::message_status status = declined;

        // Traverse each target in the order in which they are connected.
        for (target_iterator iter = _M_connectedTargets.begin();
            *iter != NULL;
            ++iter)
        {
            // Propagate the message to the target.
            concurrency::ITarget<_Target_type>* target = *iter;
            status = target->propagate(message, this);

            // If the target accepts the message then ownership of message has
            // changed. Do not propagate this message to any other target.
            if (status == accepted)
            {
                break;
            }

            // If the target only reserved this message, we must wait until the
            // target accepts the message.
            if (_M_pReservedFor != NULL)
            {
                break;
            }
        }

        // If status is anything other than accepted, then the head message
        // was not propagated out. To preserve the order in which output
        // messages are propagated, we must stop propagation until the head
        // message is accepted.
        if (status != accepted)
        {
            break;
        }
    }
}

```

This method propagates out all messages from the output queue. Every message in the queue is offered to every target block until one of the target blocks accepts the message. The `priority_buffer` class preserves the order of the outgoing messages. Therefore, the first message in the output queue must be accepted by a target block before this method offers any other message to the target blocks.

17. In the `protected` section, define the `propagate_message` method.

```

// Asynchronously passes a message from an ISource block to this block.
// This method is typically called by propagator_block::propagate.
virtual concurrency::message_status propagate_message(concurrency::message<_Source_type>* message,
    concurrency::ISource<_Source_type>* source)
{
    // Accept the message from the source block.
    message = source->accept(message->msg_id(), this);

    if (message != NULL)
    {
        // Insert the message into the input queue. The type parameter Pr
        // defines how to order messages by priority.
        {
            concurrency::critical_section::scoped_lock lock(_input_lock);
            _input_messages.push(message);
        }

        // Asynchronously send the message to the target blocks.
        async_send(NULL);
        return accepted;
    }
    else
    {
        return missed;
    }
}

```

The `propagate_message` method enables the `priority_buffer` class to act as a message receiver, or target. This method receives the message that is offered by the provided source block and inserts that message into the priority queue. The `propagate_message` method then asynchronously sends all output messages to the target blocks.

The runtime calls this method when you call the `concurrency::asend` function or when the message block is connected to other message blocks.

18. In the `protected` section, define the `send_message` method.

```

// Synchronously passes a message from an ISource block to this block.
// This method is typically called by propagator_block::send.
virtual concurrency::message_status send_message(concurrency::message<_Source_type>* message,
    concurrency::ISource<_Source_type>* source)
{
    // Accept the message from the source block.
    message = source->accept(message->msg_id(), this);

    if (message != NULL)
    {
        // Insert the message into the input queue. The type parameter Pr
        // defines how to order messages by priority.
        {
            concurrency::critical_section::scoped_lock lock(_input_lock);
            _input_messages.push(message);
        }

        // Synchronously send the message to the target blocks.
        sync_send(NULL);
        return accepted;
    }
    else
    {
        return missed;
    }
}

```

The `send_message` method resembles `propagate_message`. However it sends the output messages synchronously instead of asynchronously.

The runtime calls this method during a synchronous send operation, such as when you call the `concurrency::send` function.

The `priority_buffer` class contains constructor overloads that are typical in many message block types. Some constructor overloads take `concurrency::Scheduler` or `concurrency::ScheduleGroup` objects, which enable the message block to be managed by a specific task scheduler. Other constructor overloads take a filter function. Filter functions enable message blocks to accept or reject a message on the basis of its payload. For more information about message filters, see [Asynchronous Message Blocks](#). For more information about task schedulers, see [Task Scheduler](#).

Because the `priority_buffer` class orders messages by priority and then by the order in which messages are received, this class is most useful when it receives messages asynchronously, for example, when you call the `concurrency::asend` function or when the message block is connected to other message blocks.

[\[Top\]](#)

## The Complete Example

The following example shows the complete definition of the `priority_buffer` class.

```
// priority_buffer.h
#pragma once
#include <agents.h>
#include <queue>

namespace std
{
    // A specialization of less that tests whether the priority element of a
    // message is less than the priority element of another message.
    template<class Type, class PriorityType>
    struct less<concurrency::message<tuple<PriorityType,Type>>>
    {
        typedef concurrency::message<tuple<PriorityType, Type>> MessageType;

        bool operator()(const MessageType* left, const MessageType* right) const
        {
            // apply operator< to the first element (the priority)
            // of the tuple payload.
            return (get<0>(left->payload) < get<0>(right->payload));
        }
    };

    // A specialization of less that tests whether the priority element of a
    // message is greater than the priority element of another message.
    template<class Type, class PriorityType>
    struct greater<concurrency::message<tuple<PriorityType, Type>>>
    {
        typedef concurrency::message<std::tuple<PriorityType,Type>> MessageType;

        bool operator()(const MessageType* left, const MessageType* right) const
        {
            // apply operator> to the first element (the priority)
            // of the tuple payload.
            return (get<0>(left->payload) > get<0>(right->payload));
        }
    };
}

namespace concurrencyex
{
    // A message block type that orders incoming messages first by priority
```

```

// A message block type that orders incoming messages first by priority,
// and then by the order in which messages are received.
template<class Type,
        typename PriorityType = int,
        typename Pr = std::less<message<std::tuple<PriorityType, Type>>*>>
class priority_buffer : public
concurrency::propagator_block<concurrency::multi_link_registry<concurrency::ITarget<Type>>,
        concurrency::multi_link_registry<concurrency::ISource<std::tuple<PriorityType, Type>>>>
{
public:
    // Constructs a priority_buffer message block.
    priority_buffer()
    {
        initialize_source_and_target();
    }

    // Constructs a priority_buffer message block with the given filter function.
    priority_buffer(filter_method const& filter)
    {
        initialize_source_and_target();
        register_filter(filter);
    }

    // Constructs a priority_buffer message block that uses the provided
    // Scheduler object to propagate messages.
    priority_buffer(concurrency::Scheduler& scheduler)
    {
        initialize_source_and_target(&scheduler);
    }

    // Constructs a priority_buffer message block with the given filter function
    // and uses the provided Scheduler object to propagate messages.
    priority_buffer(concurrency::Scheduler& scheduler, filter_method const& filter)
    {
        initialize_source_and_target(&scheduler);
        register_filter(filter);
    }

    // Constructs a priority_buffer message block that uses the provided
    // SchedulerGroup object to propagate messages.
    priority_buffer(concurrency::ScheduleGroup& schedule_group)
    {
        initialize_source_and_target(NULL, &schedule_group);
    }

    // Constructs a priority_buffer message block with the given filter function
    // and uses the provided SchedulerGroup object to propagate messages.
    priority_buffer(concurrency::ScheduleGroup& schedule_group, filter_method const& filter)
    {
        initialize_source_and_target(NULL, &schedule_group);
        register_filter(filter);
    }

    // Destroys the message block.
    ~priority_buffer()
    {
        // Remove all links.
        remove_network_links();
    }

    // Sends an item to the message block.
    bool enqueue(Type const& item)
    {
        return concurrency::asend<Type>(this, item);
    }

    // Receives an item from the message block.
    Type dequeue()
    {

```

```

    return receive<Type>(this);
}

protected:
    // Asynchronously passes a message from an ISource block to this block.
    // This method is typically called by propagator_block::propagate.
    virtual concurrency::message_status propagate_message(concurrency::message<_Source_type>* message,
        concurrency::ISource<_Source_type>* source)
    {
        // Accept the message from the source block.
        message = source->accept(message->msg_id(), this);

        if (message != NULL)
        {
            // Insert the message into the input queue. The type parameter Pr
            // defines how to order messages by priority.
            {
                concurrency::critical_section::scoped_lock lock(_input_lock);
                _input_messages.push(message);
            }

            // Asynchronously send the message to the target blocks.
            async_send(NULL);
            return accepted;
        }
        else
        {
            return missed;
        }
    }

    // Synchronously passes a message from an ISource block to this block.
    // This method is typically called by propagator_block::send.
    virtual concurrency::message_status send_message(concurrency::message<_Source_type>* message,
        concurrency::ISource<_Source_type>* source)
    {
        // Accept the message from the source block.
        message = source->accept(message->msg_id(), this);

        if (message != NULL)
        {
            // Insert the message into the input queue. The type parameter Pr
            // defines how to order messages by priority.
            {
                concurrency::critical_section::scoped_lock lock(_input_lock);
                _input_messages.push(message);
            }

            // Synchronously send the message to the target blocks.
            sync_send(NULL);
            return accepted;
        }
        else
        {
            return missed;
        }
    }

    // Accepts a message that was offered by this block by transferring ownership
    // to the caller.
    virtual concurrency::message<_Target_type>* accept_message(concurrency::runtime_object_identity
msg_id)
    {
        concurrency::message<_Target_type>* message = NULL;

        // Transfer ownership if the provided message identifier matches
        // the identifier of the front of the output message queue.
        if (!_output_messages.empty() &&
            _output_messages.front()->msg_id() == msg_id)

```

```

    {
        message = _output_messages.front();
        _output_messages.pop();
    }

    return message;
}

// Reserves a message that was previously offered by this block.
virtual bool reserve_message(concurrency::runtime_object_identity msg_id)
{
    // Allow the message to be reserved if the provided message identifier
    // is the message identifier of the front of the message queue.
    return (!_output_messages.empty() &&
        _output_messages.front()->msg_id() == msg_id);
}

// Transfers the message that was previously offered by this block
// to the caller. The caller of this method is the target block that
// reserved the message.
virtual concurrency::message<Type>* consume_message(concurrency::runtime_object_identity msg_id)
{
    // Transfer ownership of the message to the caller.
    return accept_message(msg_id);
}

// Releases a previous message reservation.
virtual void release_message(concurrency::runtime_object_identity msg_id)
{
    // The head message must be the one that is reserved.
    if (_output_messages.empty() ||
        _output_messages.front()->msg_id() != msg_id)
    {
        throw message_not_found();
    }
}

// Resumes propagation after a reservation has been released.
virtual void resume_propagation()
{
    // Propagate out any messages in the output queue.
    if (_output_messages.size() > 0)
    {
        async_send(NULL);
    }
}

// Notifies this block that a new target has been linked to it.
virtual void link_target_notification(concurrency::ITarget<Target_type>*)
{
    // Do not propagate messages if a target block reserves
    // the message at the front of the queue.
    if (_M_pReservedFor != NULL)
    {
        return;
    }

    // Propagate out any messages that are in the output queue.
    propagate_priority_order();
}

// Transfers the message at the front of the input queue to the output queue
// and propagates out all messages in the output queue.
virtual void propagate_to_any_targets(concurrency::message<Target_type>*)
{
    // Retrieve the message from the front of the input queue.
    concurrency::message<Source_type>* input_message = NULL;
    {
        concurrency::critical_section::scoped_lock lock(_input_lock);
    }
}

```

```

        if (_input_messages.size() > 0)
        {
            input_message = _input_messages.top();
            _input_messages.pop();
        }
    }

    // Move the message to the output queue.
    if (input_message != NULL)
    {
        // The payload of the output message does not contain the
        // priority of the message.
        concurrency::message<Target_type>* output_message =
            new concurrency::message<Target_type>(get<1>(input_message->payload));
        _output_messages.push(output_message);

        // Free the memory for the input message.
        delete input_message;

        // Do not propagate messages if the new message is not the head message.
        // In this case, the head message is reserved by another message block.
        if (_output_messages.front()->msg_id() != output_message->msg_id())
        {
            return;
        }
    }

    // Propagate out the output messages.
    propagate_priority_order();
}

private:

// Propagates messages in priority order.
void propagate_priority_order()
{
    // Cancel propagation if another block reserves the head message.
    if (_M_pReservedFor != NULL)
    {
        return;
    }

    // Propagate out all output messages.
    // Because this block preserves message ordering, stop propagation
    // if any of the messages are not accepted by a target block.
    while (!_output_messages.empty())
    {
        // Get the next message.
        concurrency::message<Target_type> * message = _output_messages.front();

        concurrency::message_status status = declined;

        // Traverse each target in the order in which they are connected.
        for (target_iterator iter = _M_connectedTargets.begin();
            *iter != NULL;
            ++iter)
        {
            // Propagate the message to the target.
            concurrency::ITarget<Target_type>* target = *iter;
            status = target->propagate(message, this);

            // If the target accepts the message then ownership of message has
            // changed. Do not propagate this message to any other target.
            if (status == accepted)
            {
                break;
            }
        }

        // If the target only reserved this message, we must wait until the

```

```

        // target accepts the message.
        if (_M_pReservedFor != NULL)
        {
            break;
        }
    }

    // If status is anything other than accepted, then the head message
    // was not propagated out. To preserve the order in which output
    // messages are propagated, we must stop propagation until the head
    // message is accepted.
    if (status != accepted)
    {
        break;
    }
}

private:

    // Stores incoming messages.
    // The type parameter Pr specifies how to order messages by priority.
    std::priority_queue<
        concurrency::message<_Source_type>*,
        std::vector<concurrency::message<_Source_type>*>,
        Pr
    > _input_messages;

    // Synchronizes access to the input message queue.
    concurrency::critical_section _input_lock;

    // Stores outgoing messages.
    std::queue<concurrency::message<_Target_type>*> _output_messages;

private:
    // Hide assignment operator and copy constructor.
    priority_buffer const &operator =(priority_buffer const&);
    priority_buffer(priority_buffer const &);
};
}

```

The following example concurrently performs a number of `asend` and `concurrency::receive` operations on a `priority_buffer` object.

```

// priority_buffer.cpp
// compile with: /EHsc
#include <ppl.h>
#include <iostream>
#include "priority_buffer.h"

using namespace concurrency;
using namespace concurrencyex;
using namespace std;

int wmain()
{
    // Concurrently perform a number of asend and receive operations
    // on a priority_buffer object.

    priority_buffer<int> pb;

    parallel_invoke(
        [&pb] { for (int i = 0; i < 25; ++i) asend(pb, make_tuple(2, 36)); },
        [&pb] { for (int i = 0; i < 25; ++i) asend(pb, make_tuple(0, 12)); },
        [&pb] { for (int i = 0; i < 25; ++i) asend(pb, make_tuple(1, 24)); },
        [&pb] {
            for (int i = 0; i < 75; ++i) {
                wcout << receive(pb) << L' ';
                if ((i+1) % 25 == 0)
                    wcout << endl;
            }
        }
    );
}

```

This example produces the following sample output.

```

36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36 36
24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24 24
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12

```

The `priority_buffer` class orders messages first by priority and then by the order in which it receives messages. In this example, messages with greater numerical priority are inserted towards the front of the queue.

[\[Top\]](#)

## Compiling the Code

Copy the example code and paste it in a Visual Studio project, or paste the definition of the `priority_buffer` class in a file that is named `priority_buffer.h` and the test program in a file that is named `priority_buffer.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe /EHsc priority_buffer.cpp
```

## See also

[Concurrency Runtime Walkthroughs](#)

[Asynchronous Message Blocks](#)

[Message Passing Functions](#)

# Concurrency Runtime Best Practices

3/4/2019 • 2 minutes to read • [Edit Online](#)

This section describes best practices that can help you make effective use of the Concurrency Runtime. These best practices apply to the [Parallel Patterns Library \(PPL\)](#), the [Asynchronous Agents Library](#), and the [Task Scheduler](#).

## In This Section

### [Best Practices in the Parallel Patterns Library](#)

Describes the best practices to follow when you use the Parallel Patterns Library (PPL).

### [Best Practices in the Asynchronous Agents Library](#)

Describes the best practices to follow when you use the Asynchronous Agents Library.

### [General Best Practices in the Concurrency Runtime](#)

Describes best practices that apply to multiple areas of the Concurrency Runtime.

## Related Sections

### [Concurrency Runtime](#)

Introduces the Concurrency Runtime, a concurrency framework for C++.

### [Parallel Patterns Library \(PPL\)](#)

Describes how to use various parallel patterns, for example, parallel algorithms, in your applications.

### [Asynchronous Agents Library](#)

Describes how to use asynchronous agents in your applications.

### [Synchronization Data Structures](#)

Describes the various synchronization primitives that the Concurrency Runtime provides.

### [Task Scheduler](#)

Describes how to use the Task Scheduler to adjust the performance of your applications.

# Best Practices in the Parallel Patterns Library

3/4/2019 • 22 minutes to read • [Edit Online](#)

This document describes how best to make effective use of the Parallel Patterns Library (PPL). The PPL provides general-purpose containers, objects, and algorithms for performing fine-grained parallelism.

For more information about the PPL, see [Parallel Patterns Library \(PPL\)](#).

## Sections

This document contains the following sections:

- [Do Not Parallelize Small Loop Bodies](#)
- [Express Parallelism at the Highest Possible Level](#)
- [Use `parallel\_invoke` to Solve Divide-and-Conquer Problems](#)
- [Use Cancellation or Exception Handling to Break from a Parallel Loop](#)
- [Understand how Cancellation and Exception Handling Affect Object Destruction](#)
- [Do Not Block Repeatedly in a Parallel Loop](#)
- [Do Not Perform Blocking Operations When You Cancel Parallel Work](#)
- [Do Not Write to Shared Data in a Parallel Loop](#)
- [When Possible, Avoid False Sharing](#)
- [Make Sure That Variables Are Valid Throughout the Lifetime of a Task](#)

## Do Not Parallelize Small Loop Bodies

The parallelization of relatively small loop bodies can cause the associated scheduling overhead to outweigh the benefits of parallel processing. Consider the following example, which adds each pair of elements in two arrays.

```

// small-loops.cpp
// compile with: /EHsc
#include <ppl.h>
#include <iostream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create three arrays that each have the same size.
    const size_t size = 100000;
    int a[size], b[size], c[size];

    // Initialize the arrays a and b.
    for (size_t i = 0; i < size; ++i)
    {
        a[i] = i;
        b[i] = i * 2;
    }

    // Add each pair of elements in arrays a and b in parallel
    // and store the result in array c.
    parallel_for<size_t>(0, size, [&a,&b,&c](size_t i) {
        c[i] = a[i] + b[i];
    });

    // TODO: Do something with array c.
}

```

The workload for each parallel loop iteration is too small to benefit from the overhead for parallel processing. You can improve the performance of this loop by performing more work in the loop body or by performing the loop serially.

[\[Top\]](#)

## Express Parallelism at the Highest Possible Level

When you parallelize code only at the low level, you can introduce a fork-join construct that does not scale as the number of processors increases. A *fork-join* construct is a construct where one task divides its work into smaller parallel subtasks and waits for those subtasks to finish. Each subtask can recursively divide itself into additional subtasks.

Although the fork-join model can be useful for solving a variety of problems, there are situations where the synchronization overhead can decrease scalability. For example, consider the following serial code that processes image data.

```

// Calls the provided function for each pixel in a Bitmap object.
void ProcessImage(Bitmap* bmp, const function<void (DWORD*)>& f)
{
    int width = bmp->GetWidth();
    int height = bmp->GetHeight();

    // Lock the bitmap.
    BitmapData bitmapData;
    Rect rect(0, 0, bmp->GetWidth(), bmp->GetHeight());
    bmp->LockBits(&rect, ImageLockModeWrite, PixelFormat32bppRGB, &bitmapData);

    // Get a pointer to the bitmap data.
    DWORD* image_bits = (DWORD*)bitmapData.Scan0;

    // Call the function for each pixel in the image.
    for (int y = 0; y < height; ++y)
    {
        for (int x = 0; x < width; ++x)
        {
            // Get the current pixel value.
            DWORD* curr_pixel = image_bits + (y * width) + x;

            // Call the function.
            f(*curr_pixel);
        }
    }

    // Unlock the bitmap.
    bmp->UnlockBits(&bitmapData);
}

```

Because each loop iteration is independent, you can parallelize much of the work, as shown in the following example. This example uses the [concurrency::parallel\\_for](#) algorithm to parallelize the outer loop.

```

// Calls the provided function for each pixel in a Bitmap object.
void ProcessImage(Bitmap* bmp, const function<void (DWORD*)>& f)
{
    int width = bmp->GetWidth();
    int height = bmp->GetHeight();

    // Lock the bitmap.
    BitmapData bitmapData;
    Rect rect(0, 0, bmp->GetWidth(), bmp->GetHeight());
    bmp->LockBits(&rect, ImageLockModeWrite, PixelFormat32bppRGB, &bitmapData);

    // Get a pointer to the bitmap data.
    DWORD* image_bits = (DWORD*)bitmapData.Scan0;

    // Call the function for each pixel in the image.
    parallel_for (0, height, [&, width](int y)
    {
        for (int x = 0; x < width; ++x)
        {
            // Get the current pixel value.
            DWORD* curr_pixel = image_bits + (y * width) + x;

            // Call the function.
            f(*curr_pixel);
        }
    });

    // Unlock the bitmap.
    bmp->UnlockBits(&bitmapData);
}

```

The following example illustrates a fork-join construct by calling the `ProcessImage` function in a loop. Each call to `ProcessImage` does not return until each subtask finishes.

```
// Processes each bitmap in the provided vector.
void ProcessImages(vector<Bitmap*> bitmaps, const function<void (DWORD*)>& f)
{
    for_each(begin(bitmaps), end(bitmaps), [&f](Bitmap* bmp) {
        ProcessImage(bmp, f);
    });
}
```

If each iteration of the parallel loop either performs almost no work, or the work that is performed by the parallel loop is imbalanced, that is, some loop iterations take longer than others, the scheduling overhead that is required to frequently fork and join work can outweigh the benefit to parallel execution. This overhead increases as the number of processors increases.

To reduce the amount of scheduling overhead in this example, you can parallelize outer loops before you parallelize inner loops or use another parallel construct such as pipelining. The following example modifies the `ProcessImages` function to use the `concurrency::parallel_for_each` algorithm to parallelize the outer loop.

```
// Processes each bitmap in the provided vector.
void ProcessImages(vector<Bitmap*> bitmaps, const function<void (DWORD*)>& f)
{
    parallel_for_each(begin(bitmaps), end(bitmaps), [&f](Bitmap* bmp) {
        ProcessImage(bmp, f);
    });
}
```

For a similar example that uses a pipeline to perform image processing in parallel, see [Walkthrough: Creating an Image-Processing Network](#).

[\[Top\]](#)

## Use `parallel_invoke` to Solve Divide-and-Conquer Problems

A *divide-and-conquer* problem is a form of the fork-join construct that uses recursion to break a task into subtasks. In addition to the `concurrency::task_group` and `concurrency::structured_task_group` classes, you can also use the `concurrency::parallel_invoke` algorithm to solve divide-and-conquer problems. The `parallel_invoke` algorithm has a more succinct syntax than task group objects, and is useful when you have a fixed number of parallel tasks.

The following example illustrates the use of the `parallel_invoke` algorithm to implement the bitonic sorting algorithm.

```

// Sorts the given sequence in the specified order.
template <class T>
void parallel_bitonic_sort(T* items, int lo, int n, bool dir)
{
    if (n > 1)
    {
        // Divide the array into two partitions and then sort
        // the partitions in different directions.
        int m = n / 2;

        parallel_invoke(
            [&] { parallel_bitonic_sort(items, lo, m, INCREASING); },
            [&] { parallel_bitonic_sort(items, lo + m, m, DECREASING); }
        );

        // Merge the results.
        parallel_bitonic_merge(items, lo, n, dir);
    }
}

```

To reduce overhead, the `parallel_invoke` algorithm performs the last of the series of tasks on the calling context.

For the complete version of this example, see [How to: Use `parallel\_invoke` to Write a Parallel Sort Routine](#). For more information about the `parallel_invoke` algorithm, see [Parallel Algorithms](#).

[\[Top\]](#)

## Use Cancellation or Exception Handling to Break from a Parallel Loop

The PPL provides two ways to cancel the parallel work that is performed by a task group or parallel algorithm. One way is to use the cancellation mechanism that is provided by the `concurrency::task_group` and `concurrency::structured_task_group` classes. The other way is to throw an exception in the body of a task work function. The cancellation mechanism is more efficient than exception handling at canceling a tree of parallel work. A *parallel work tree* is a group of related task groups in which some task groups contain other task groups. The cancellation mechanism cancels a task group and its child task groups in a top-down manner. Conversely, exception handling works in a bottom-up manner and must cancel each child task group independently as the exception propagates upward.

When you work directly with a task group object, use the `concurrency::task_group::cancel` or `concurrency::structured_task_group::cancel` methods to cancel the work that belongs to that task group. To cancel a parallel algorithm, for example, `parallel_for`, create a parent task group and cancel that task group. For example, consider the following function, `parallel_find_any`, which searches for a value in an array in parallel.

```

// Returns the position in the provided array that contains the given value,
// or -1 if the value is not in the array.
template<typename T>
int parallel_find_any(const T a[], size_t count, const T& what)
{
    // The position of the element in the array.
    // The default value, -1, indicates that the element is not in the array.
    int position = -1;

    // Call parallel_for in the context of a cancellation token to search for the element.
    cancellation_token_source cts;
    run_with_cancellation_token([count, what, &a, &position, &cts]()
    {
        parallel_for(std::size_t(0), count, [what, &a, &position, &cts](int n) {
            if (a[n] == what)
            {
                // Set the return value and cancel the remaining tasks.
                position = n;
                cts.cancel();
            }
        });
    }, cts.get_token());

    return position;
}

```

Because parallel algorithms use task groups, when one of the parallel iterations cancels the parent task group, the overall task is canceled. For the complete version of this example, see [How to: Use Cancellation to Break from a Parallel Loop](#).

Although exception handling is a less efficient way to cancel parallel work than the cancellation mechanism, there are cases where exception handling is appropriate. For example, the following method, `for_all`, recursively performs a work function on each node of a `tree` structure. In this example, the `_children` data member is a `std::list` that contains `tree` objects.

```

// Performs the given work function on the data element of the tree and
// on each child.
template<class Function>
void tree::for_all(Function& action)
{
    // Perform the action on each child.
    parallel_for_each(begin(_children), end(_children), [&](tree& child) {
        child.for_all(action);
    });

    // Perform the action on this node.
    action(*this);
}

```

The caller of the `tree::for_all` method can throw an exception if it does not require the work function to be called on each element of the tree. The following example shows the `search_for_value` function, which searches for a value in the provided `tree` object. The `search_for_value` function uses a work function that throws an exception when the current element of the tree matches the provided value. The `search_for_value` function uses a `try-catch` block to capture the exception and print the result to the console.

```

// Searches for a value in the provided tree object.
template <typename T>
void search_for_value(tree<T>& t, int value)
{
    try
    {
        // Call the for_all method to search for a value. The work function
        // throws an exception when it finds the value.
        t.for_all([value](const tree<T>& node) {
            if (node.get_data() == value)
            {
                throw &node;
            }
        });
    }
    catch (const tree<T>* node)
    {
        // A matching node was found. Print a message to the console.
        wstringstream ss;
        ss << L"Found a node with value " << value << L'.' << endl;
        wcout << ss.str();
        return;
    }

    // A matching node was not found. Print a message to the console.
    wstringstream ss;
    ss << L"Did not find node with value " << value << L'.' << endl;
    wcout << ss.str();
}

```

For the complete version of this example, see [How to: Use Exception Handling to Break from a Parallel Loop](#).

For more general information about the cancellation and exception-handling mechanisms that are provided by the PPL, see [Cancellation in the PPL](#) and [Exception Handling](#).

[\[Top\]](#)

## Understand how Cancellation and Exception Handling Affect Object Destruction

In a tree of parallel work, a task that is canceled prevents child tasks from running. This can cause problems if one of the child tasks performs an operation that is important to your application, such as freeing a resource. In addition, task cancellation can cause an exception to propagate through an object destructor and cause undefined behavior in your application.

In the following example, the `Resource` class describes a resource and the `Container` class describes a container that holds resources. In its destructor, the `Container` class calls the `cleanup` method on two of its `Resource` members in parallel and then calls the `cleanup` method on its third `Resource` member.

```

// parallel-resource-destruction.h
#pragma once
#include <ppl.h>
#include <sstream>
#include <iostream>

// Represents a resource.
class Resource
{
public:
    Resource(const std::wstring& name)
        : _name(name)
    {

```

```

}

// Frees the resource.
void cleanup()
{
    // Print a message as a placeholder.
    std::wstringstream ss;
    ss << _name << L": Freeing..." << std::endl;
    std::wcout << ss.str();
}
private:
    // The name of the resource.
    std::wstring _name;
};

// Represents a container that holds resources.
class Container
{
public:
    Container(const std::wstring& name)
        : _name(name)
        , _resource1(L"Resource 1")
        , _resource2(L"Resource 2")
        , _resource3(L"Resource 3")
    {
    }

    ~Container()
    {
        std::wstringstream ss;
        ss << _name << L": Freeing resources..." << std::endl;
        std::wcout << ss.str();

        // For illustration, assume that cleanup for _resource1
        // and _resource2 can happen concurrently, and that
        // _resource3 must be freed after _resource1 and _resource2.

        concurrency::parallel_invoke(
            [this]() { _resource1.cleanup(); },
            [this]() { _resource2.cleanup(); }
        );

        _resource3.cleanup();
    }

private:
    // The name of the container.
    std::wstring _name;

    // Resources.
    Resource _resource1;
    Resource _resource2;
    Resource _resource3;
};

```

Although this pattern has no problems on its own, consider the following code that runs two tasks in parallel. The first task creates a `Container` object and the second task cancels the overall task. For illustration, the example uses two `concurrency::event` objects to make sure that the cancellation occurs after the `Container` object is created and that the `Container` object is destroyed after the cancellation operation occurs.

```

// parallel-resource-destruction.cpp
// compile with: /EHsc
#include "parallel-resource-destruction.h"

using namespace concurrency;
using namespace std;

static_assert(false, "This example illustrates a non-recommended practice.");

int main()
{
    // Create a task_group that will run two tasks.
    task_group tasks;

    // Used to synchronize the tasks.
    event e1, e2;

    // Run two tasks. The first task creates a Container object. The second task
    // cancels the overall task group. To illustrate the scenario where a child
    // task is not run because its parent task is cancelled, the event objects
    // ensure that the Container object is created before the overall task is
    // cancelled and that the Container object is destroyed after the overall
    // task is cancelled.

    tasks.run([&tasks,&e1,&e2] {
        // Create a Container object.
        Container c(L"Container 1");

        // Allow the second task to continue.
        e2.set();

        // Wait for the task to be cancelled.
        e1.wait();
    });

    tasks.run([&tasks,&e1,&e2] {
        // Wait for the first task to create the Container object.
        e2.wait();

        // Cancel the overall task.
        tasks.cancel();

        // Allow the first task to continue.
        e1.set();
    });

    // Wait for the tasks to complete.
    tasks.wait();

    wcout << L"Exiting program..." << endl;
}

```

This example produces the following output:

```

Container 1: Freeing resources...Exiting program...

```

This code example contains the following issues that may cause it to behave differently than you expect:

- The cancellation of the parent task causes the child task, the call to `concurrency::parallel_invoke`, to also be canceled. Therefore, these two resources are not freed.
- The cancellation of the parent task causes the child task to throw an internal exception. Because the `Container` destructor does not handle this exception, the exception is propagated upward and the third resource is not freed.

- The exception that is thrown by the child task propagates through the `Container` destructor. Throwing from a destructor puts the application in an undefined state.

We recommend that you do not perform critical operations, such as the freeing of resources, in tasks unless you can guarantee that these tasks will not be canceled. We also recommend that you do not use runtime functionality that can throw in the destructor of your types.

[\[Top\]](#)

## Do Not Block Repeatedly in a Parallel Loop

A parallel loop such as `concurrency::parallel_for` or `concurrency::parallel_for_each` that is dominated by blocking operations may cause the runtime to create many threads over a short time.

The Concurrency Runtime performs additional work when a task finishes or cooperatively blocks or yields. When one parallel loop iteration blocks, the runtime might begin another iteration. When there are no available idle threads, the runtime creates a new thread.

When the body of a parallel loop occasionally blocks, this mechanism helps maximize the overall task throughput. However, when many iterations block, the runtime may create many threads to run the additional work. This could lead to low-memory conditions or poor utilization of hardware resources.

Consider the following example that calls the `concurrency::send` function in each iteration of a `parallel_for` loop. Because `send` blocks cooperatively, the runtime creates a new thread to run additional work every time `send` is called.

```
// repeated-blocking.cpp
// compile with: /EHsc
#include <ppl.h>
#include <agents.h>

using namespace concurrency;

static_assert(false, "This example illustrates a non-recommended practice.");

int main()
{
    // Create a message buffer.
    overwrite_buffer<int> buffer;

    // Repeatedly send data to the buffer in a parallel loop.
    parallel_for(0, 1000, [&buffer](int i) {

        // The send function blocks cooperatively.
        // We discourage the use of repeated blocking in a parallel
        // loop because it can cause the runtime to create
        // a large number of threads over a short period of time.
        send(buffer, i);
    });
}
```

We recommend that you refactor your code to avoid this pattern. In this example, you can avoid the creation of additional threads by calling `send` in a serial `for` loop.

[\[Top\]](#)

## Do Not Perform Blocking Operations When You Cancel Parallel Work

When possible, do not perform blocking operations before you call the `concurrency::task_group::cancel` or

`concurrency::structured_task_group::cancel` method to cancel parallel work.

When a task performs a cooperative blocking operation, the runtime can perform other work while the first task waits for data. The runtime reschedules the waiting task when it unblocks. The runtime typically reschedules tasks that were more recently unblocked before it reschedules tasks that were less recently unblocked. Therefore, the runtime could schedule unnecessary work during the blocking operation, which leads to decreased performance. Accordingly, when you perform a blocking operation before you cancel parallel work, the blocking operation can delay the call to `cancel`. This causes other tasks to perform unnecessary work.

Consider the following example that defines the `parallel_find_answer` function, which searches for an element of the provided array that satisfies the provided predicate function. When the predicate function returns **true**, the parallel work function creates an `Answer` object and cancels the overall task.

```

// blocking-cancel.cpp
// compile with: /c /EHsc
#include <windows.h>
#include <ppl.h>

using namespace concurrency;

// Encapsulates the result of a search operation.
template<typename T>
class Answer
{
public:
    explicit Answer(const T& data)
        : _data(data)
    {
    }

    T get_data() const
    {
        return _data;
    }

    // TODO: Add other methods as needed.

private:
    T _data;

    // TODO: Add other data members as needed.
};

// Searches for an element of the provided array that satisfies the provided
// predicate function.
template<typename T, class Predicate>
Answer<T>* parallel_find_answer(const T a[], size_t count, const Predicate& pred)
{
    // The result of the search.
    Answer<T>* answer = nullptr;
    // Ensures that only one task produces an answer.
    volatile long first_result = 0;

    // Use parallel_for and a task group to search for the element.
    structured_task_group tasks;
    tasks.run_and_wait([&]
    {
        // Declare the type alias for use in the inner lambda function.
        typedef T T;

        parallel_for<size_t>(0, count, [&](const T& n) {
            if (pred(a[n]) && InterlockedExchange(&first_result, 1) == 0)
            {
                // Create an object that holds the answer.
                answer = new Answer<T>(a[n]);
                // Cancel the overall task.
                tasks.cancel();
            }
        });
    });

    return answer;
}

```

The `new` operator performs a heap allocation, which might block. The runtime performs other work only when the task performs a cooperative blocking call, such as a call to `concurrency::critical_section::lock`.

The following example shows how to prevent unnecessary work, and thereby improve performance. This example cancels the task group before it allocates the storage for the `Answer` object.

```

// Searches for an element of the provided array that satisfies the provided
// predicate function.
template<typename T, class Predicate>
Answer<T>* parallel_find_answer(const T a[], size_t count, const Predicate& pred)
{
    // The result of the search.
    Answer<T>* answer = nullptr;
    // Ensures that only one task produces an answer.
    volatile long first_result = 0;

    // Use parallel_for and a task group to search for the element.
    structured_task_group tasks;
    tasks.run_and_wait([&]
    {
        // Declare the type alias for use in the inner lambda function.
        typedef T T;

        parallel_for<size_t>(0, count, [&](const T& n) {
            if (pred(a[n]) && InterlockedExchange(&first_result, 1) == 0)
            {
                // Cancel the overall task.
                tasks.cancel();
                // Create an object that holds the answer.
                answer = new Answer<T>(a[n]);
            }
        });
    });

    return answer;
}

```

[\[Top\]](#)

## Do Not Write to Shared Data in a Parallel Loop

The Concurrency Runtime provides several data structures, for example, [concurrency::critical\\_section](#), that synchronize concurrent access to shared data. These data structures are useful in many cases, for example, when multiple tasks infrequently require shared access to a resource.

Consider the following example that uses the [concurrency::parallel\\_for\\_each](#) algorithm and a `critical_section` object to compute the count of prime numbers in a `std::array` object. This example does not scale because each thread must wait to access the shared variable `prime_sum`.

```

critical_section cs;
prime_sum = 0;
parallel_for_each(begin(a), end(a), [&](int i) {
    cs.lock();
    prime_sum += (is_prime(i) ? i : 0);
    cs.unlock();
});

```

This example can also lead to poor performance because the frequent locking operation effectively serializes the loop. In addition, when a Concurrency Runtime object performs a blocking operation, the scheduler might create an additional thread to perform other work while the first thread waits for data. If the runtime creates many threads because many tasks are waiting for shared data, the application can perform poorly or enter a low-resource state.

The PPL defines the [concurrency::combinable](#) class, which helps you eliminate shared state by providing access to shared resources in a lock-free manner. The `combinable` class provides thread-local storage that lets you perform fine-grained computations and then merge those computations into a final result. You can think of a `combinable`

object as a reduction variable.

The following example modifies the previous one by using a `combinable` object instead of a `critical_section` object to compute the sum. This example scales because each thread holds its own local copy of the sum. This example uses the `concurrency::combinable::combine` method to merge the local computations into the final result.

```
combinable<int> sum;
parallel_for_each(begin(a), end(a), [&](int i) {
    sum.local() += (is_prime(i) ? i : 0);
});
prime_sum = sum.combine(plus<int>());
```

For the complete version of this example, see [How to: Use combinable to Improve Performance](#). For more information about the `combinable` class, see [Parallel Containers and Objects](#).

[\[Top\]](#)

## When Possible, Avoid False Sharing

*False sharing* occurs when multiple concurrent tasks that are running on separate processors write to variables that are located on the same cache line. When one task writes to one of the variables, the cache line for both variables is invalidated. Each processor must reload the cache line every time that the cache line is invalidated. Therefore, false sharing can cause decreased performance in your application.

The following basic example shows two concurrent tasks that each increment a shared counter variable.

```
volatile long count = 0L;
concurrency::parallel_invoke(
    [&count] {
        for(int i = 0; i < 100000000; ++i)
            InterlockedIncrement(&count);
    },
    [&count] {
        for(int i = 0; i < 100000000; ++i)
            InterlockedIncrement(&count);
    }
);
```

To eliminate the sharing of data between the two tasks, you can modify the example to use two counter variables. This example computes the final counter value after the tasks finish. However, this example illustrates false sharing because the variables `count1` and `count2` are likely to be located on the same cache line.

```
long count1 = 0L;
long count2 = 0L;
concurrency::parallel_invoke(
    [&count1] {
        for(int i = 0; i < 100000000; ++i)
            ++count1;
    },
    [&count2] {
        for(int i = 0; i < 100000000; ++i)
            ++count2;
    }
);
long count = count1 + count2;
```

One way to eliminate false sharing is to make sure that the counter variables are on separate cache lines. The following example aligns the variables `count1` and `count2` on 64-byte boundaries.

```
__declspec(align(64)) long count1 = 0L;
__declspec(align(64)) long count2 = 0L;
concurrency::parallel_invoke(
    [&count1] {
        for(int i = 0; i < 100000000; ++i)
            ++count1;
    },
    [&count2] {
        for(int i = 0; i < 100000000; ++i)
            ++count2;
    }
);
long count = count1 + count2;
```

This example assumes that the size of the memory cache is 64 or fewer bytes.

We recommend that you use the [concurrency::combinable](#) class when you must share data among tasks. The `combinable` class creates thread-local variables in such a way that false sharing is less likely. For more information about the `combinable` class, see [Parallel Containers and Objects](#).

[\[Top\]](#)

## Make Sure That Variables Are Valid Throughout the Lifetime of a Task

When you provide a lambda expression to a task group or parallel algorithm, the capture clause specifies whether the body of the lambda expression accesses variables in the enclosing scope by value or by reference. When you pass variables to a lambda expression by reference, you must guarantee that the lifetime of that variable persists until the task finishes.

Consider the following example that defines the `object` class and the `perform_action` function. The `perform_action` function creates an `object` variable and performs some action on that variable asynchronously. Because the task is not guaranteed to finish before the `perform_action` function returns, the program will crash or exhibit unspecified behavior if the `object` variable is destroyed when the task is running.

```

// lambda-lifetime.cpp
// compile with: /c /EHsc
#include <ppl.h>

using namespace concurrency;

// A type that performs an action.
class object
{
public:
    void action() const
    {
        // TODO: Details omitted for brevity.
    }
};

// Performs an action asynchronously.
void perform_action(task_group& tasks)
{
    // Create an object variable and perform some action on
    // that variable asynchronously.
    object obj;
    tasks.run([&obj] {
        obj.action();
    });

    // NOTE: The object variable is destroyed here. The program
    // will crash or exhibit unspecified behavior if the task
    // is still running when this function returns.
}

```

Depending on the requirements of your application, you can use one of the following techniques to guarantee that variables remain valid throughout the lifetime of every task.

The following example passes the `object` variable by value to the task. Therefore, the task operates on its own copy of the variable.

```

// Performs an action asynchronously.
void perform_action(task_group& tasks)
{
    // Create an object variable and perform some action on
    // that variable asynchronously.
    object obj;
    tasks.run([obj] {
        obj.action();
    });
}

```

Because the `object` variable is passed by value, any state changes that occur to this variable do not appear in the original copy.

The following example uses the `concurrency::task_group::wait` method to make sure that the task finishes before the `perform_action` function returns.

```
// Performs an action.
void perform_action(task_group& tasks)
{
    // Create an object variable and perform some action on
    // that variable.
    object obj;
    tasks.run([&obj] {
        obj.action();
    });

    // Wait for the task to finish.
    tasks.wait();
}
```

Because the task now finishes before the function returns, the `perform_action` function no longer behaves asynchronously.

The following example modifies the `perform_action` function to take a reference to the `object` variable. The caller must guarantee that the lifetime of the `object` variable is valid until the task finishes.

```
// Performs an action asynchronously.
void perform_action(object& obj, task_group& tasks)
{
    // Perform some action on the object variable.
    tasks.run([&obj] {
        obj.action();
    });
}
```

You can also use a pointer to control the lifetime of an object that you pass to a task group or parallel algorithm.

For more information about lambda expressions, see [Lambda Expressions](#).

[\[Top\]](#)

## See also

[Concurrency Runtime Best Practices](#)

[Parallel Patterns Library \(PPL\)](#)

[Parallel Containers and Objects](#)

[Parallel Algorithms](#)

[Cancellation in the PPL](#)

[Exception Handling](#)

[Walkthrough: Creating an Image-Processing Network](#)

[How to: Use `parallel\_invoke` to Write a Parallel Sort Routine](#)

[How to: Use Cancellation to Break from a Parallel Loop](#)

[How to: Use `combinable` to Improve Performance](#)

[Best Practices in the Asynchronous Agents Library](#)

[General Best Practices in the Concurrency Runtime](#)

# Best Practices in the Asynchronous Agents Library

3/4/2019 • 13 minutes to read • [Edit Online](#)

This document describes how to make effective use of the Asynchronous Agents Library. The Agents Library promotes an actor-based programming model and in-process message passing for coarse-grained dataflow and pipelining tasks.

For more information about the Agents Library, see [Asynchronous Agents Library](#).

## Sections

This document contains the following sections:

- [Use Agents to Isolate State](#)
- [Use a Throttling Mechanism to Limit the Number of Messages in a Data Pipeline](#)
- [Do Not Perform Fine-Grained Work in a Data Pipeline](#)
- [Do Not Pass Large Message Payloads by Value](#)
- [Use `shared\_ptr` in a Data Network When Ownership Is Undefined](#)

## Use Agents to Isolate State

The Agents Library provides alternatives to shared state by letting you connect isolated components through an asynchronous message-passing mechanism. Asynchronous agents are most effective when they isolate their internal state from other components. By isolating state, multiple components do not typically act on shared data. State isolation can enable your application to scale because it reduces contention on shared memory. State isolation also reduces the chance of deadlock and race conditions because components do not have to synchronize access to shared data.

You typically isolate state in an agent by holding data members in the `private` or `protected` sections of the agent class and by using message buffers to communicate state changes. The following example shows the `basic_agent` class, which derives from `concurrency::agent`. The `basic_agent` class uses two message buffers to communicate with external components. One message buffer holds incoming messages; the other message buffer holds outgoing messages.

```

// basic-agent.cpp
// compile with: /c /EHsc
#include <agents.h>

// An agent that uses message buffers to isolate state and communicate
// with other components.
class basic_agent : public concurrency::agent
{
public:
    basic_agent(concurrency::unbounded_buffer<int>& input)
        : _input(input)
    {
    }

    // Retrieves the message buffer that holds output messages.
    concurrency::unbounded_buffer<int>& output()
    {
        return _output;
    }

protected:
    void run()
    {
        while (true)
        {
            // Read from the input message buffer.
            int value = concurrency::receive(_input);

            // TODO: Do something with the value.
            int result = value;

            // Write the result to the output message buffer.
            concurrency::send(_output, result);
        }
        done();
    }

private:
    // Holds incoming messages.
    concurrency::unbounded_buffer<int>& _input;
    // Holds outgoing messages.
    concurrency::unbounded_buffer<int> _output;
};

```

For complete examples about how to define and use agents, see [Walkthrough: Creating an Agent-Based Application](#) and [Walkthrough: Creating a Dataflow Agent](#).

[\[Top\]](#)

## Use a Throttling Mechanism to Limit the Number of Messages in a Data Pipeline

Many message-buffer types, such as `concurrency::unbounded_buffer`, can hold an unlimited number of messages. When a message producer sends messages to a data pipeline faster than the consumer can process these messages, the application can enter a low-memory or out-of-memory state. You can use a throttling mechanism, for example, a semaphore, to limit the number of messages that are concurrently active in a data pipeline.

The following basic example demonstrates how to use a semaphore to limit the number of messages in a data pipeline. The data pipeline uses the `concurrency::wait` function to simulate an operation that takes at least 100 milliseconds. Because the sender produces messages faster than the consumer can process those messages, this example defines the `semaphore` class to enable the application to limit the number of active messages.

```

// message-throttling.cpp
// compile with: /EHsc
#include <windows.h> // for GetTickCount()
#include <atomic>
#include <agents.h>
#include <concrct.h>
#include <concurrent_queue.h>
#include <sstream>
#include <iostream>

using namespace concurrency;
using namespace std;

// A semaphore type that uses cooperative blocking semantics.
class semaphore
{
public:
    explicit semaphore(long long capacity)
        : _semaphore_count(capacity)
    {
    }

    // Acquires access to the semaphore.
    void acquire()
    {
        // The capacity of the semaphore is exceeded when the semaphore count
        // falls below zero. When this happens, add the current context to the
        // back of the wait queue and block the current context.
        if (--_semaphore_count < 0)
        {
            _waiting_contexts.push(Context::CurrentContext());
            Context::Block();
        }
    }

    // Releases access to the semaphore.
    void release()
    {
        // If the semaphore count is negative, unblock the first waiting context.
        if (++_semaphore_count <= 0)
        {
            // A call to acquire might have decremented the counter, but has not
            // yet finished adding the context to the queue.
            // Create a spin loop that waits for the context to become available.
            Context* waiting = NULL;
            while (!_waiting_contexts.try_pop(waiting))
            {
                Context::Yield();
            }

            // Unblock the context.
            waiting->Unblock();
        }
    }

private:
    // The semaphore count.
    atomic<long long> _semaphore_count;

    // A concurrency-safe queue of contexts that must wait to
    // acquire the semaphore.
    concurrent_queue<Context*> _waiting_contexts;
};

// A synchronization primitive that is signaled when its
// count reaches zero.
class countdown_event
{
public:

```

```

countdown_event(long long count)
: _current(count)
{
    // Set the event if the initial count is zero.
    if (_current == 0LL)
        _event.set();
}

// Decrements the event counter.
void signal() {
    if(--_current == 0LL) {
        _event.set();
    }
}

// Increments the event counter.
void add_count() {
    if(++_current == 1LL) {
        _event.reset();
    }
}

// Blocks the current context until the event is set.
void wait() {
    _event.wait();
}

private:
    // The current count.
    atomic<long long> _current;
    // The event that is set when the counter reaches zero.
    event _event;

    // Disable copy constructor.
    countdown_event(const countdown_event&);
    // Disable assignment.
    countdown_event const & operator=(countdown_event const&);
};

int wmain()
{
    // The number of messages to send to the consumer.
    const long long MessageCount = 5;

    // The number of messages that can be active at the same time.
    const long long ActiveMessages = 2;

    // Used to compute the elapsed time.
    DWORD start_time;

    // Computes the elapsed time, rounded-down to the nearest
    // 100 milliseconds.
    auto elapsed = [&start_time] {
        return (GetTickCount() - start_time)/100*100;
    };

    // Limits the number of active messages.
    semaphore s(ActiveMessages);

    // Enables the consumer message buffer to coordinate completion
    // with the main application.
    countdown_event e(MessageCount);

    // Create a data pipeline that has three stages.

    // The first stage of the pipeline prints a message.
    transformer<int, int> print_message([&elapsed](int n) -> int {
        wstringstream ss;
        ss << elapsed() << " received " << n << endl;

```

```

    ss << elapsed() << L": received " << i << endl;
    wcout << ss.str();

    // Send the input to the next pipeline stage.
    return n;
});

// The second stage of the pipeline simulates a
// time-consuming operation.
transformer<int, int> long_operation([](int n) -> int {
    wait(100);

    // Send the input to the next pipeline stage.
    return n;
});

// The third stage of the pipeline releases the semaphore
// and signals to the main application that the message has
// been processed.
call<int> release_and_signal([&](int unused) {
    // Enable the sender to send the next message.
    s.release();

    // Signal that the message has been processed.
    e.signal();
});

// Connect the pipeline.
print_message.link_target(&long_operation);
long_operation.link_target(&release_and_signal);

// Send several messages to the pipeline.
start_time = GetTickCount();
for(auto i = 0; i < MessageCount; ++i)
{
    // Acquire access to the semaphore.
    s.acquire();

    // Print the message to the console.
    wstringstream ss;
    ss << elapsed() << L": sending " << i << L"... " << endl;
    wcout << ss.str();

    // Send the message.
    send(print_message, i);
}

// Wait for the consumer to process all messages.
e.wait();
}
/* Sample output:
0: sending 0...
0: received 0
0: sending 1...
0: received 1
100: sending 2...
100: received 2
200: sending 3...
200: received 3
300: sending 4...
300: received 4
*/

```

The `semaphore` object limits the pipeline to process at most two messages at the same time.

The producer in this example sends relatively few messages to the consumer. Therefore, this example does not demonstrate a potential low-memory or out-of-memory condition. However, this mechanism is useful when a

data pipeline contains a relatively high number of messages.

For more information about how to create the semaphore class that is used in this example, see [How to: Use the Context Class to Implement a Cooperative Semaphore](#).

[\[Top\]](#)

## Do Not Perform Fine-Grained Work in a Data Pipeline

The Agents Library is most useful when the work that is performed by a data pipeline is fairly coarse-grained. For example, one application component might read data from a file or a network connection and occasionally send that data to another component. The protocol that the Agents Library uses to propagate messages causes the message-passing mechanism to have more overhead than the task parallel constructs that are provided by the [Parallel Patterns Library](#) (PPL). Therefore, make sure that the work that is performed by a data pipeline is long enough to offset this overhead.

Although a data pipeline is most effective when its tasks are coarse-grained, each stage of the data pipeline can use PPL constructs such as task groups and parallel algorithms to perform more fine-grained work. For an example of a coarse-grained data network that uses fine-grained parallelism at each processing stage, see [Walkthrough: Creating an Image-Processing Network](#).

[\[Top\]](#)

## Do Not Pass Large Message Payloads by Value

In some cases, the runtime creates a copy of every message that it passes from one message buffer to another message buffer. For example, the [concurrency::overwrite\\_buffer](#) class offers a copy of every message that it receives to each of its targets. The runtime also creates a copy of the message data when you use message-passing functions such as [concurrency::send](#) and [concurrency::receive](#) to write messages to and read messages from a message buffer. Although this mechanism helps eliminate the risk of concurrently writing to shared data, it could lead to poor memory performance when the message payload is relatively large.

You can use pointers or references to improve memory performance when you pass messages that have a large payload. The following example compares passing large messages by value to passing pointers to the same message type. The example defines two agent types, `producer` and `consumer`, that act on `message_data` objects. The example compares the time that is required for the producer to send several `message_data` objects to the consumer to the time that is required for the producer agent to send several pointers to `message_data` objects to the consumer.

```
// message-payloads.cpp
// compile with: /EHsc
#include <Windows.h>
#include <agents.h>
#include <iostream>

using namespace concurrency;
using namespace std;

// Calls the provided work function and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
__int64 time_call(Function&& f)
{
    __int64 begin = GetTickCount();
    f();
    return GetTickCount() - begin;
}

// A message structure that contains large payload data.
struct message_data
```

```

struct message_data
{
    int id;
    string source;
    unsigned char binary_data[32768];
};

// A basic agent that produces values.
template <typename T>
class producer : public agent
{
public:
    explicit producer(ITarget<T>& target, unsigned int message_count)
        : _target(target)
        , _message_count(message_count)
    {
    }
protected:
    void run();

private:
    // The target buffer to write to.
    ITarget<T>& _target;
    // The number of messages to send.
    unsigned int _message_count;
};

// Template specialization for message_data.
template <>
void producer<message_data>::run()
{
    // Send a number of messages to the target buffer.
    while (_message_count > 0)
    {
        message_data message;
        message.id = _message_count;
        message.source = "Application";

        send(_target, message);
        --_message_count;
    }

    // Set the agent to the finished state.
    done();
}

// Template specialization for message_data*.
template <>
void producer<message_data*>::run()
{
    // Send a number of messages to the target buffer.
    while (_message_count > 0)
    {
        message_data* message = new message_data;
        message->id = _message_count;
        message->source = "Application";

        send(_target, message);
        --_message_count;
    }

    // Set the agent to the finished state.
    done();
}

// A basic agent that consumes values.
template <typename T>
class consumer : public agent
{

```

```

public:
    explicit consumer(ISource<T>& source, unsigned int message_count)
        : _source(source)
        , _message_count(message_count)
    {
    }

protected:
    void run();

private:
    // The source buffer to read from.
    ISource<T>& _source;
    // The number of messages to receive.
    unsigned int _message_count;
};

// Template specialization for message_data.
template <>
void consumer<message_data>::run()
{
    // Receive a number of messages from the source buffer.
    while (_message_count > 0)
    {
        message_data message = receive(_source);
        --_message_count;

        // TODO: Do something with the message.
        // ...
    }

    // Set the agent to the finished state.
    done();
}

template <>
void consumer<message_data*>::run()
{
    // Receive a number of messages from the source buffer.
    while (_message_count > 0)
    {
        message_data* message = receive(_source);
        --_message_count;

        // TODO: Do something with the message.
        // ...

        // Release the memory for the message.
        delete message;
    }

    // Set the agent to the finished state.
    done();
}

int wmain()
{
    // The number of values for the producer agent to send.
    const unsigned int count = 10000;

    __int64 elapsed;

    // Run the producer and consumer agents.
    // This version uses message_data as the message payload type.

    wcout << L"Using message_data..." << endl;
    elapsed = time_call([count] {
        // A message buffer that is shared by the agents.
        unbounded_buffer<message_data> buffer;

```

```

// Create and start the producer and consumer agents.
producer<message_data> prod(buffer, count);
consumer<message_data> cons(buffer, count);
prod.start();
cons.start();

// Wait for the agents to finish.
agent::wait(&prod);
agent::wait(&cons);
});
wcout << L"took " << elapsed << L"ms." << endl;

// Run the producer and consumer agents a second time.
// This version uses message_data* as the message payload type.

wcout << L"Using message_data*..." << endl;
elapsed = time_call([count] {
    // A message buffer that is shared by the agents.
    unbounded_buffer<message_data*> buffer;

    // Create and start the producer and consumer agents.
    producer<message_data*> prod(buffer, count);
    consumer<message_data*> cons(buffer, count);
    prod.start();
    cons.start();

    // Wait for the agents to finish.
    agent::wait(&prod);
    agent::wait(&cons);
});
wcout << L"took " << elapsed << L"ms." << endl;
}

```

This example produces the following sample output:

```

Using message_data...
took 437ms.
Using message_data*...
took 47ms.

```

The version that uses pointers performs better because it eliminates the requirement for the runtime to create a full copy of every `message_data` object that it passes from the producer to the consumer.

[\[Top\]](#)

## Use `shared_ptr` in a Data Network When Ownership Is Undefined

When you send messages by pointer through a message-passing pipeline or network, you typically allocate the memory for each message at the front of the network and free that memory at the end of the network. Although this mechanism frequently works well, there are cases in which it is difficult or not possible to use it. For example, consider the case in which the data network contains multiple end nodes. In this case, there is no clear location to free the memory for the messages.

To solve this problem, you can use a mechanism, for example, `std::shared_ptr`, that enables a pointer to be owned by multiple components. When the final `shared_ptr` object that owns a resource is destroyed, the resource is also freed.

The following example demonstrates how to use `shared_ptr` to share pointer values among multiple message buffers. The example connects a `concurrency::overwrite_buffer` object to three `concurrency::call` objects. The `overwrite_buffer` class offers messages to each of its targets. Because there are multiple owners of the data at the

end of the data network, this example uses `shared_ptr` to enable each `call` object to share ownership of the messages.

```
// message-sharing.cpp
// compile with: /EHsc
#include <agents.h>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

// A type that holds a resource.
class resource
{
public:
    resource(int id) : _id(id)
    {
        wcout << L"Creating resource " << _id << L"..." << endl;
    }
    ~resource()
    {
        wcout << L"Destroying resource " << _id << L"..." << endl;
    }

    // Retrieves the identifier for the resource.
    int id() const { return _id; }

    // TODO: Add additional members here.
private:
    // An identifier for the resource.
    int _id;

    // TODO: Add additional members here.
};

int wmain()
{
    // A message buffer that sends messages to each of its targets.
    overwrite_buffer<shared_ptr<resource>> input;

    // Create three call objects that each receive resource objects
    // from the input message buffer.

    call<shared_ptr<resource>> receiver1(
        [](shared_ptr<resource> res) {
            wstringstream ss;
            ss << L"receiver1: received resource " << res->id() << endl;
            wcout << ss.str();
        },
        [](shared_ptr<resource> res) {
            return res != nullptr;
        }
    );

    call<shared_ptr<resource>> receiver2(
        [](shared_ptr<resource> res) {
            wstringstream ss;
            ss << L"receiver2: received resource " << res->id() << endl;
            wcout << ss.str();
        },
        [](shared_ptr<resource> res) {
            return res != nullptr;
        }
    );

    event e;
    call<shared_ptr<resource>> receiver3(
```

```

call<shared_ptr<resource>> receiver3(
    [&e](shared_ptr<resource> res) {
        e.set();
    },
    [](shared_ptr<resource> res) {
        return res == nullptr;
    }
);

// Connect the call objects to the input message buffer.
input.link_target(&receiver1);
input.link_target(&receiver2);
input.link_target(&receiver3);

// Send a few messages through the network.
send(input, make_shared<resource>(42));
send(input, make_shared<resource>(64));
send(input, shared_ptr<resource>(nullptr));

// Wait for the receiver that accepts the nullptr value to
// receive its message.
e.wait();
}

```

This example produces the following sample output:

```

Creating resource 42...
receiver1: received resource 42
Creating resource 64...
receiver2: received resource 42
receiver1: received resource 64
Destroying resource 42...
receiver2: received resource 64
Destroying resource 64...

```

## See also

[Concurrency Runtime Best Practices](#)

[Asynchronous Agents Library](#)

[Walkthrough: Creating an Agent-Based Application](#)

[Walkthrough: Creating a Dataflow Agent](#)

[Walkthrough: Creating an Image-Processing Network](#)

[Best Practices in the Parallel Patterns Library](#)

[General Best Practices in the Concurrency Runtime](#)

# General Best Practices in the Concurrency Runtime

3/4/2019 • 10 minutes to read • [Edit Online](#)

This document describes best practices that apply to multiple areas of the Concurrency Runtime.

## Sections

This document contains the following sections:

- [Use Cooperative Synchronization Constructs When Possible](#)
- [Avoid Lengthy Tasks That Do Not Yield](#)
- [Use Oversubscription to Offset Operations That Block or Have High Latency](#)
- [Use Concurrent Memory Management Functions When Possible](#)
- [Use RAII to Manage the Lifetime of Concurrency Objects](#)
- [Do Not Create Concurrency Objects at Global Scope](#)
- [Do Not Use Concurrency Objects in Shared Data Segments](#)

## Use Cooperative Synchronization Constructs When Possible

The Concurrency Runtime provides many concurrency-safe constructs that do not require an external synchronization object. For example, the [concurrency::concurrent\\_vector](#) class provides concurrency-safe append and element access operations. However, for cases where you require exclusive access to a resource, the runtime provides the [concurrency::critical\\_section](#), [concurrency::reader\\_writer\\_lock](#), and [concurrency::event](#) classes. These types behave cooperatively; therefore, the task scheduler can reallocate processing resources to another context as the first task waits for data. When possible, use these synchronization types instead of other synchronization mechanisms, such as those provided by the Windows API, which do not behave cooperatively. For more information about these synchronization types and a code example, see [Synchronization Data Structures](#) and [Comparing Synchronization Data Structures to the Windows API](#).

[\[Top\]](#)

## Avoid Lengthy Tasks That Do Not Yield

Because the task scheduler behaves cooperatively, it does not provide fairness among tasks. Therefore, a task can prevent other tasks from starting. Although this is acceptable in some cases, in other cases this can cause deadlock or starvation.

The following example performs more tasks than the number of allocated processing resources. The first task does not yield to the task scheduler and therefore the second task does not start until the first task finishes.

```

// cooperative-tasks.cpp
// compile with: /EHsc
#include <ppl.h>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

// Data that the application passes to lightweight tasks.
struct task_data_t
{
    int id; // a unique task identifier.
    event e; // signals that the task has finished.
};

// A lightweight task that performs a lengthy operation.
void task(void* data)
{
    task_data_t* task_data = reinterpret_cast<task_data_t*>(data);

    // Create a large loop that occasionally prints a value to the console.
    int i;
    for (i = 0; i < 1000000000; ++i)
    {
        if (i > 0 && (i % 250000000) == 0)
        {
            wstringstream ss;
            ss << task_data->id << L": " << i << endl;
            wcout << ss.str();
        }
    }
    wstringstream ss;
    ss << task_data->id << L": " << i << endl;
    wcout << ss.str();

    // Signal to the caller that the thread is finished.
    task_data->e.set();
}

int wmain()
{
    // For illustration, limit the number of concurrent
    // tasks to one.
    Scheduler::SetDefaultSchedulerPolicy(SchedulerPolicy(2,
        MinConcurrency, 1, MaxConcurrency, 1));

    // Schedule two tasks.

    task_data_t t1;
    t1.id = 0;
    CurrentScheduler::ScheduleTask(task, &t1);

    task_data_t t2;
    t2.id = 1;
    CurrentScheduler::ScheduleTask(task, &t2);

    // Wait for the tasks to finish.

    t1.e.wait();
    t2.e.wait();
}

```

This example produces the following output:

```
1: 250000000 1: 500000000 1: 750000000 1: 1000000000 2: 250000000 2: 500000000 2: 750000000 2:
```

1000000000

There are several ways to enable cooperation between the two tasks. One way is to occasionally yield to the task scheduler in a long-running task. The following example modifies the `task` function to call the [concurrency::Context::Yield](#) method to yield execution to the task scheduler so that another task can run.

```
// A lightweight task that performs a lengthy operation.
void task(void* data)
{
    task_data_t* task_data = reinterpret_cast<task_data_t*>(data);

    // Create a large loop that occasionally prints a value to the console.
    int i;
    for (i = 0; i < 1000000000; ++i)
    {
        if (i > 0 && (i % 250000000) == 0)
        {
            wstringstream ss;
            ss << task_data->id << L": " << i << endl;
            wcout << ss.str();

            // Yield control back to the task scheduler.
            Context::Yield();
        }
    }
    wstringstream ss;
    ss << task_data->id << L": " << i << endl;
    wcout << ss.str();

    // Signal to the caller that the thread is finished.
    task_data->e.set();
}
```

This example produces the following output:

```
1: 250000000
2: 250000000
1: 500000000
2: 500000000
1: 750000000
2: 750000000
1: 1000000000
2: 1000000000
```

The `Context::Yield` method yields only another active thread on the scheduler to which the current thread belongs, a lightweight task, or another operating system thread. This method does not yield to work that is scheduled to run in a [concurrency::task\\_group](#) or [concurrency::structured\\_task\\_group](#) object but has not yet started.

There are other ways to enable cooperation among long-running tasks. You can break a large task into smaller subtasks. You can also enable oversubscription during a lengthy task. Oversubscription lets you create more threads than the available number of hardware threads. Oversubscription is especially useful when a lengthy task contains a high amount of latency, for example, reading data from disk or from a network connection. For more information about lightweight tasks and oversubscription, see [Task Scheduler](#).

[\[Top\]](#)

## Use Oversubscription to Offset Operations That Block or Have High Latency

The Concurrency Runtime provides synchronization primitives, such as [concurrency::critical\\_section](#), that enable tasks to cooperatively block and yield to each other. When one task cooperatively blocks or yields, the task scheduler can reallocate processing resources to another context as the first task waits for data.

There are cases in which you cannot use the cooperative blocking mechanism that is provided by the Concurrency Runtime. For example, an external library that you use might use a different synchronization mechanism. Another example is when you perform an operation that could have a high amount of latency, for example, when you use the Windows API `ReadFile` function to read data from a network connection. In these cases, oversubscription can enable other tasks to run when another task is idle. Oversubscription lets you create more threads than the available number of hardware threads.

Consider the following function, `download`, which downloads the file at the given URL. This example uses the [concurrency::Context::Oversubscribe](#) method to temporarily increase the number of active threads.

```
// Downloads the file at the given URL.
string download(const string& url)
{
    // Enable oversubscription.
    Context::Oversubscribe(true);

    // Download the file.
    string content = GetHttpFile(_session, url.c_str());

    // Disable oversubscription.
    Context::Oversubscribe(false);

    return content;
}
```

Because the `GetHttpFile` function performs a potentially latent operation, oversubscription can enable other tasks to run as the current task waits for data. For the complete version of this example, see [How to: Use Oversubscription to Offset Latency](#).

[\[Top\]](#)

## Use Concurrent Memory Management Functions When Possible

Use the memory management functions, [concurrency::Alloc](#) and [concurrency::Free](#), when you have fine-grained tasks that frequently allocate small objects that have a relatively short lifetime. The Concurrency Runtime holds a separate memory cache for each running thread. The `Alloc` and `Free` functions allocate and free memory from these caches without the use of locks or memory barriers.

For more information about these memory management functions, see [Task Scheduler](#). For an example that uses these functions, see [How to: Use Alloc and Free to Improve Memory Performance](#).

[\[Top\]](#)

## Use RAII to Manage the Lifetime of Concurrency Objects

The Concurrency Runtime uses exception handling to implement features such as cancellation. Therefore, write exception-safe code when you call into the runtime or call another library that calls into the runtime.

The *Resource Acquisition Is Initialization* (RAII) pattern is one way to safely manage the lifetime of a concurrency object under a given scope. Under the RAII pattern, a data structure is allocated on the stack. That data structure initializes or acquires a resource when it is created and destroys or releases that resource when the data structure is destroyed. The RAII pattern guarantees that the destructor is called before the enclosing scope exits. This pattern is useful when a function contains multiple `return` statements. This pattern also helps you write

exception-safe code. When a `throw` statement causes the stack to unwind, the destructor for the RAII object is called; therefore, the resource is always correctly deleted or released.

The runtime defines several classes that use the RAII pattern, for example, `concurrency::critical_section::scoped_lock` and `concurrency::reader_writer_lock::scoped_lock`. These helper classes are known as *scoped locks*. These classes provide several benefits when you work with `concurrency::critical_section` or `concurrency::reader_writer_lock` objects. The constructor of these classes acquires access to the provided `critical_section` or `reader_writer_lock` object; the destructor releases access to that object. Because a scoped lock releases access to its mutual exclusion object automatically when it is destroyed, you do not manually unlock the underlying object.

Consider the following class, `account`, which is defined by an external library and therefore cannot be modified.

```
// account.h
#pragma once
#include <exception>
#include <sstream>

// Represents a bank account.
class account
{
public:
    explicit account(int initial_balance = 0)
        : _balance(initial_balance)
    {
    }

    // Retrieves the current balance.
    int balance() const
    {
        return _balance;
    }

    // Deposits the specified amount into the account.
    int deposit(int amount)
    {
        _balance += amount;
        return _balance;
    }

    // Withdraws the specified amount from the account.
    int withdraw(int amount)
    {
        if (_balance < 0)
        {
            std::stringstream ss;
            ss << "negative balance: " << _balance << std::endl;
            throw std::exception((ss.str().c_str()));
        }

        _balance -= amount;
        return _balance;
    }

private:
    // The current balance.
    int _balance;
};
```

The following example performs multiple transactions on an `account` object in parallel. The example uses a `critical_section` object to synchronize access to the `account` object because the `account` class is not concurrency-safe. Each parallel operation uses a `critical_section::scoped_lock` object to guarantee that the `critical_section` object is unlocked when the operation either succeeds or fails. When the account balance is

negative, the `withdraw` operation fails by throwing an exception.

```
// account-transactions.cpp
// compile with: /EHsc
#include "account.h"
#include <ppl.h>
#include <iostream>
#include <sstream>

using namespace concurrency;
using namespace std;

int wmain()
{
    // Create an account that has an initial balance of 1924.
    account acc(1924);

    // Synchronizes access to the account object because the account class is
    // not concurrency-safe.
    critical_section cs;

    // Perform multiple transactions on the account in parallel.
    try
    {
        parallel_invoke(
            [&acc, &cs] {
                critical_section::scoped_lock lock(cs);
                wcout << L"Balance before deposit: " << acc.balance() << endl;
                acc.deposit(1000);
                wcout << L"Balance after deposit: " << acc.balance() << endl;
            },
            [&acc, &cs] {
                critical_section::scoped_lock lock(cs);
                wcout << L"Balance before withdrawal: " << acc.balance() << endl;
                acc.withdraw(50);
                wcout << L"Balance after withdrawal: " << acc.balance() << endl;
            },
            [&acc, &cs] {
                critical_section::scoped_lock lock(cs);
                wcout << L"Balance before withdrawal: " << acc.balance() << endl;
                acc.withdraw(3000);
                wcout << L"Balance after withdrawal: " << acc.balance() << endl;
            }
        );
    }
    catch (const exception& e)
    {
        wcout << L"Error details:" << endl << L"\t" << e.what() << endl;
    }
}
```

This example produces the following sample output:

```
Balance before deposit: 1924
Balance after deposit: 2924
Balance before withdrawal: 2924
Balance after withdrawal: -76
Balance before withdrawal: -76
Error details:
    negative balance: -76
```

For additional examples that use the RAII pattern to manage the lifetime of concurrency objects, see [Walkthrough: Removing Work from a User-Interface Thread](#), [How to: Use the Context Class to Implement a Cooperative Semaphore](#), and [How to: Use Oversubscription to Offset Latency](#).

[\[Top\]](#)

## Do Not Create Concurrency Objects at Global Scope

When you create a concurrency object at global scope you can cause issues such as deadlock or memory access violations to occur in your application.

For example, when you create a Concurrency Runtime object, the runtime creates a default scheduler for you if one was not yet created. A runtime object that is created during global object construction will accordingly cause the runtime to create this default scheduler. However, this process takes an internal lock, which can interfere with the initialization of other objects that support the Concurrency Runtime infrastructure. This internal lock might be required by another infrastructure object that has not yet been initialized, and can thus cause deadlock to occur in your application.

The following example demonstrates the creation of a global `concurrency::Scheduler` object. This pattern applies not only to the `Scheduler` class but all other types that are provided by the Concurrency Runtime. We recommend that you do not follow this pattern because it can cause unexpected behavior in your application.

```
// global-scheduler.cpp
// compile with: /EHsc
#include <concr.h>

using namespace concurrency;

static_assert(false, "This example illustrates a non-recommended practice.");

// Create a Scheduler object at global scope.
// BUG: This practice is not recommended because it can cause deadlock.
Scheduler* globalScheduler = Scheduler::Create(SchedulerPolicy(2,
    MinConcurrency, 2, MaxConcurrency, 4));

int wmain()
{
}
```

For examples of the correct way to create `Scheduler` objects, see [Task Scheduler](#).

[\[Top\]](#)

## Do Not Use Concurrency Objects in Shared Data Segments

The Concurrency Runtime does not support the use of concurrency objects in a shared data section, for example, a data section that is created by the `data_seg #pragma` directive. A concurrency object that is shared across process boundaries could put the runtime in an inconsistent or invalid state.

[\[Top\]](#)

## See also

[Concurrency Runtime Best Practices](#)

[Parallel Patterns Library \(PPL\)](#)

[Asynchronous Agents Library](#)

[Task Scheduler](#)

[Synchronization Data Structures](#)

[Comparing Synchronization Data Structures to the Windows API](#)

[How to: Use Alloc and Free to Improve Memory Performance](#)

[How to: Use Oversubscription to Offset Latency](#)

[How to: Use the Context Class to Implement a Cooperative Semaphore](#)

[Walkthrough: Removing Work from a User-Interface Thread](#)

[Best Practices in the Parallel Patterns Library](#)

[Best Practices in the Asynchronous Agents Library](#)

# Reference (Concurrency Runtime)

10/31/2018 • 2 minutes to read • [Edit Online](#)

This section contains reference information for the Concurrency Runtime.

## NOTE

The C++ language standard reserves the use of identifiers that begin with an underscore ( `_` ) character for implementations such as libraries. Do not use these names in your code. The behavior of code elements whose names follow this convention are not guaranteed and are subject to change in future releases. For these reasons, such code elements are omitted from the Concurrency Runtime documentation.

## In This Section

### [concurrency Namespace](#)

The concurrency namespace provides classes and functions that give you access to the Concurrency Runtime, a concurrent programming framework for C++. For more information, see [Concurrency Runtime](#).

### [std namespace](#)

### [stdx namespace](#)

# concurrency Namespace

3/4/2019 • 25 minutes to read • [Edit Online](#)

The `Concurrency` namespace provides classes and functions that give you access to the Concurrency Runtime, a concurrent programming framework for C++. For more information, see [Concurrency Runtime](#).

## Syntax

```
namespace concurrency;
```

## Members

### Typedefs

NAME	DESCRIPTION
<code>runtime_object_identity</code>	Each message instance has an identity that follows it as it is cloned and passed between messaging components. This cannot be the address of the message object.
<code>task_status</code>	A type that represents the terminal state of a task. Valid values are <code>completed</code> and <code>canceled</code> .
<code>TaskProc</code>	An elementary abstraction for a task, defined as <code>void (__cdecl * TaskProc)(void *)</code> . A <code>TaskProc</code> is called to invoke the body of a task.
<code>TaskProc_t</code>	An elementary abstraction for a task, defined as <code>void (__cdecl * TaskProc_t)(void *)</code> . A <code>TaskProc</code> is called to invoke the body of a task.

### Classes

NAME	DESCRIPTION
<a href="#">affinity_partitioner Class</a>	The <code>affinity_partitioner</code> class is similar to the <code>static_partitioner</code> class, but it improves cache affinity by its choice of mapping subranges to worker threads. It can improve performance significantly when a loop is re-executed over the same data set, and the data fits in cache. Note that the same <code>affinity_partitioner</code> object must be used with subsequent iterations of a parallel loop that is executed over a particular data set, to benefit from data locality.
<a href="#">agent Class</a>	A class intended to be used as a base class for all independent agents. It is used to hide state from other agents and interact using message-passing.

NAME	DESCRIPTION
<a href="#">auto_partitioner Class</a>	<p>The <code>auto_partitioner</code> class represents the default method <code>parallel_for</code>, <code>parallel_for_each</code> and <code>parallel_transform</code> use to partition the range they iterates over. This method of partitioning employs range stealing for load balancing as well as per-iterate cancellation.</p>
<a href="#">bad_target Class</a>	<p>This class describes an exception thrown when a messaging block is given a pointer to a target which is invalid for the operation being performed.</p>
<a href="#">call Class</a>	<p>A <code>call</code> messaging block is a multi-source, ordered <code>target_block</code> that invokes a specified function when receiving a message.</p>
<a href="#">cancellation_token Class</a>	<p>The <code>cancellation_token</code> class represents the ability to determine whether some operation has been requested to cancel. A given token can be associated with a <code>task_group</code>, <code>structured_task_group</code>, or <code>task</code> to provide implicit cancellation. It can also be polled for cancellation or have a callback registered for if and when the associated <code>cancellation_token_source</code> is canceled.</p>
<a href="#">cancellation_token_registration Class</a>	<p>The <code>cancellation_token_registration</code> class represents a callback notification from a <code>cancellation_token</code>. When the <code>register</code> method on a <code>cancellation_token</code> is used to receive notification of when cancellation occurs, a <code>cancellation_token_registration</code> object is returned as a handle to the callback so that the caller can request a specific callback no longer be made through use of the <code>deregister</code> method.</p>
<a href="#">cancellation_token_source Class</a>	<p>The <code>cancellation_token_source</code> class represents the ability to cancel some cancelable operation.</p>
<a href="#">choice Class</a>	<p>A <code>choice</code> messaging block is a multi-source, single-target block that represents a control-flow interaction with a set of sources. The choice block will wait for any one of multiple sources to produce a message and will propagate the index of the source that produced the message.</p>
<a href="#">combinable Class</a>	<p>The <code>combinable&lt;T&gt;</code> object is intended to provide thread-private copies of data, to perform lock-free thread-local sub-computations during parallel algorithms. At the end of the parallel operation, the thread-private sub-computations can then be merged into a final result. This class can be used instead of a shared variable, and can result in a performance improvement if there would otherwise be a lot of contention on that shared variable.</p>

NAME	DESCRIPTION
<a href="#">concurrent_priority_queue Class</a>	<p>The <code>concurrent_priority_queue</code> class is a container that allows multiple threads to concurrently push and pop items. Items are popped in priority order where priority is determined by a functor supplied as a template argument.</p>
<a href="#">concurrent_queue Class</a>	<p>The <code>concurrent_queue</code> class is a sequence container class that allows first-in, first-out access to its elements. It enables a limited set of concurrency-safe operations, such as <code>push</code> and <code>try_pop</code>.</p>
<a href="#">concurrent_unordered_map Class</a>	<p>The <code>concurrent_unordered_map</code> class is a concurrency-safe container that controls a varying-length sequence of elements of type <code>std::pair&lt;const K, _Element_type&gt;</code>. The sequence is represented in a way that enables concurrency-safe append, element access, iterator access, and iterator traversal operations.</p>
<a href="#">concurrent_unordered_multimap Class</a>	<p>The <code>concurrent_unordered_multimap</code> class is a concurrency-safe container that controls a varying-length sequence of elements of type <code>std::pair&lt;const K, _Element_type&gt;</code>. The sequence is represented in a way that enables concurrency-safe append, element access, iterator access and iterator traversal operations.</p>
<a href="#">concurrent_unordered_multiset Class</a>	<p>The <code>concurrent_unordered_multiset</code> class is a concurrency-safe container that controls a varying-length sequence of elements of type K. The sequence is represented in a way that enables concurrency-safe append, element access, iterator access and iterator traversal operations.</p>
<a href="#">concurrent_unordered_set Class</a>	<p>The <code>concurrent_unordered_set</code> class is a concurrency-safe container that controls a varying-length sequence of elements of type K. The sequence is represented in a way that enables concurrency-safe append, element access, iterator access and iterator traversal operations.</p>
<a href="#">concurrent_vector Class</a>	<p>The <code>concurrent_vector</code> class is a sequence container class that allows random access to any element. It enables concurrency-safe append, element access, iterator access, and iterator traversal operations.</p>
<a href="#">Context Class</a>	<p>Represents an abstraction for an execution context.</p>
<a href="#">context_self_unblock Class</a>	<p>This class describes an exception thrown when the <code>unblock</code> method of a <code>Context</code> object is called from the same context. This would indicate an attempt by a given context to unblock itself.</p>
<a href="#">context_unblock_unbalanced Class</a>	<p>This class describes an exception thrown when calls to the <code>Block</code> and <code>Unblock</code> methods of a <code>Context</code> object are not properly paired.</p>

NAME	DESCRIPTION
<a href="#">critical_section Class</a>	A non-reentrant mutex which is explicitly aware of the Concurrency Runtime.
<a href="#">CurrentScheduler Class</a>	Represents an abstraction for the current scheduler associated with the calling context.
<a href="#">default_scheduler_exists Class</a>	This class describes an exception thrown when the <code>Scheduler::SetDefaultSchedulerPolicy</code> method is called when a default scheduler already exists within the process.
<a href="#">event Class</a>	A manual reset event which is explicitly aware of the Concurrency Runtime.
<a href="#">improper_lock Class</a>	This class describes an exception thrown when a lock is acquired improperly.
<a href="#">improper_scheduler_attach Class</a>	This class describes an exception thrown when the <code>Attach</code> method is called on a <code>Scheduler</code> object which is already attached to the current context.
<a href="#">improper_scheduler_detach Class</a>	This class describes an exception thrown when the <code>CurrentScheduler::Detach</code> method is called on a context which has not been attached to any scheduler using the <code>Attach</code> method of a <code>Scheduler</code> object.
<a href="#">improper_scheduler_reference Class</a>	This class describes an exception thrown when the <code>Reference</code> method is called on a <code>Scheduler</code> object that is shutting down, from a context that is not part of that scheduler.
<a href="#">invalid_link_target Class</a>	This class describes an exception thrown when the <code>link_target</code> method of a messaging block is called and the messaging block is unable to link to the target. This can be the result of exceeding the number of links the messaging block is allowed or attempting to link a specific target twice to the same source.
<a href="#">invalid_multiple_scheduling Class</a>	This class describes an exception thrown when a <code>task_handle</code> object is scheduled multiple times using the <code>run</code> method of a <code>task_group</code> or <code>structured_task_group</code> object without an intervening call to either the <code>wait</code> or <code>run_and_wait</code> methods.
<a href="#">invalid_operation Class</a>	This class describes an exception thrown when an invalid operation is performed that is not more accurately described by another exception type thrown by the Concurrency Runtime.

NAME	DESCRIPTION
<a href="#">invalid_oversubscribe_operation Class</a>	<p>This class describes an exception thrown when the <code>Context::Oversubscribe</code> method is called with the <code>_BeginOversubscription</code> parameter set to <code>false</code> without a prior call to the <code>Context::Oversubscribe</code> method with the <code>_BeginOversubscription</code> parameter set to <code>true</code>.</p>
<a href="#">invalid_scheduler_policy_key Class</a>	<p>This class describes an exception thrown when an invalid or unknown key is passed to a <code>SchedulerPolicy</code> object constructor, or the <code>SetPolicyValue</code> method of a <code>SchedulerPolicy</code> object is passed a key that must be changed using other means such as the <code>SetConcurrencyLimits</code> method.</p>
<a href="#">invalid_scheduler_policy_thread_specification Class</a>	<p>This class describes an exception thrown when an attempt is made to set the concurrency limits of a <code>SchedulerPolicy</code> object such that the value of the <code>MinConcurrency</code> key is less than the value of the <code>MaxConcurrency</code> key.</p>
<a href="#">invalid_scheduler_policy_value Class</a>	<p>This class describes an exception thrown when a policy key of a <code>SchedulerPolicy</code> object is set to an invalid value for that key.</p>
<a href="#">ISource Class</a>	<p>The <code>ISource</code> class is the interface for all source blocks. Source blocks propagate messages to <code>ITarget</code> blocks.</p>
<a href="#">ITarget Class</a>	<p>The <code>ITarget</code> class is the interface for all target blocks. Target blocks consume messages offered to them by <code>ISource</code> blocks.</p>
<a href="#">join Class</a>	<p>A <code>join</code> messaging block is a single-target, multi-source, ordered <code>propagator_block</code> which combines together messages of type <code>T</code> from each of its sources.</p>
<a href="#">location Class</a>	<p>An abstraction of a physical location on hardware.</p>
<a href="#">message Class</a>	<p>The basic message envelope containing the data payload being passed between messaging blocks.</p>
<a href="#">message_not_found Class</a>	<p>This class describes an exception thrown when a messaging block is unable to find a requested message.</p>
<a href="#">message_processor Class</a>	<p>The <code>message_processor</code> class is the abstract base class for processing of <code>message</code> objects. There is no guarantee on the ordering of the messages.</p>

NAME	DESCRIPTION
missing_wait Class	This class describes an exception thrown when there are tasks still scheduled to a <code>task_group</code> or <code>structured_task_group</code> object at the time that object's destructor executes. This exception will never be thrown if the destructor is reached because of a stack unwinding as the result of an exception.
multi_link_registry Class	The <code>multi_link_registry</code> object is a <code>network_link_registry</code> that manages multiple source blocks or multiple target blocks.
multitype_join Class	A <code>multitype_join</code> messaging block is a multi-source, single-target messaging block that combines together messages of different types from each of its sources and offers a tuple of the combined messages to its targets.
nested_scheduler_missing_detach Class	This class describes an exception thrown when the Concurrency Runtime detects that you neglected to call the <code>CurrentScheduler::Detach</code> method on a context that attached to a second scheduler using the <code>Attach</code> method of the <code>Scheduler</code> object.
network_link_registry Class	The <code>network_link_registry</code> abstract base class manages the links between source and target blocks.
operation_timed_out Class	This class describes an exception thrown when an operation has timed out.
ordered_message_processor Class	An <code>ordered_message_processor</code> is a <code>message_processor</code> that allows message blocks to process messages in the order they were received.
overwrite_buffer Class	An <code>overwrite_buffer</code> messaging block is a multi-target, multi-source, ordered <code>propagator_block</code> capable of storing a single message at a time. New messages overwrite previously held ones.
progress_reporter Class	The progress reporter class allows reporting progress notifications of a specific type. Each <code>progress_reporter</code> object is bound to a particular asynchronous action or operation.
propagator_block Class	The <code>propagator_block</code> class is an abstract base class for message blocks that are both a source and target. It combines the functionality of both the <code>source_block</code> and <code>target_block</code> classes.
reader_writer_lock Class	A writer-preference queue-based reader-writer lock with local only spinning. The lock grants first in - first out (FIFO) access to writers and starves readers under a continuous load of writers.

NAME	DESCRIPTION
ScheduleGroup Class	Represents an abstraction for a schedule group. Schedule groups organize a set of related work that benefits from being scheduled close together either temporally, by executing another task in the same group before moving to another group, or spatially, by executing multiple items within the same group on the same NUMA node or physical socket.
Scheduler Class	Represents an abstraction for a Concurrency Runtime scheduler.
scheduler_not_attached Class	This class describes an exception thrown when an operation is performed which requires a scheduler to be attached to the current context and one is not.
scheduler_resource_allocation_error Class	This class describes an exception thrown because of a failure to acquire a critical resource in the Concurrency Runtime.
scheduler_worker_creation_error Class	This class describes an exception thrown because of a failure to create a worker execution context in the Concurrency Runtime.
SchedulerPolicy Class	The SchedulerPolicy class contains a set of key/value pairs, one for each policy element, that control the behavior of a scheduler instance.
simple_partitioner Class	The simple_partitioner class represents a static partitioning of the range iterated over by parallel_for. The partitioner divides the range into chunks such that each chunk has at least the number of iterations specified by the chunk size.
single_assignment Class	A single_assignment messaging block is a multi-target, multi-source, ordered propagator_block capable of storing a single, write-once message.
single_link_registry Class	The single_link_registry object is a network_link_registry that manages only a single source or target block.
source_block Class	The source_block class is an abstract base class for source-only blocks. The class provides basic link management functionality as well as common error checks.
source_link_manager Class	The source_link_manager object manages messaging block network links to ISource blocks.
static_partitioner Class	The static_partitioner class represents a static partitioning of the range iterated over by parallel_for. The partitioner divides the range into as many chunks as there are workers available to the underlying scheduler.

NAME	DESCRIPTION
<a href="#">structured_task_group Class</a>	<p>The <code>structured_task_group</code> class represents a highly structured collection of parallel work. You can queue individual parallel tasks to a <code>structured_task_group</code> using <code>task_handle</code> objects, and wait for them to complete, or cancel the task group before they have finished executing, which will abort any tasks that have not begun execution.</p>
<a href="#">target_block Class</a>	<p>The <code>target_block</code> class is an abstract base class that provides basic link management functionality and error checking for target only blocks.</p>
<a href="#">task Class (Concurrency Runtime)</a>	<p>The Parallel Patterns Library (PPL) <code>task</code> class. A <code>task</code> object represents work that can be executed asynchronously, and concurrently with other tasks and parallel work produced by parallel algorithms in the Concurrency Runtime. It produces a result of type <code>_ResultType</code> on successful completion. Tasks of type <code>task&lt;void&gt;</code> produce no result. A task can be waited upon and canceled independently of other tasks. It can also be composed with other tasks using <code>continuations( then )</code>, and <code>join( when_all )</code> and <code>choice( when_any )</code> patterns.</p>
<a href="#">task_canceled Class</a>	<p>This class describes an exception thrown by the PPL tasks layer in order to force the current task to cancel. It is also thrown by the <code>get()</code> method on <code>task</code>, for a canceled task.</p>
<a href="#">task_completion_event Class</a>	<p>The <code>task_completion_event</code> class allows you to delay the execution of a task until a condition is satisfied, or start a task in response to an external event.</p>
<a href="#">task_continuation_context Class</a>	<p>The <code>task_continuation_context</code> class allows you to specify where you would like a continuation to be executed. It is only useful to use this class from a UWP app. For non-Windows Runtime apps, the task continuation's execution context is determined by the runtime, and not configurable.</p>
<a href="#">task_group Class</a>	<p>The <code>task_group</code> class represents a collection of parallel work which can be waited on or canceled.</p>
<a href="#">task_handle Class</a>	<p>The <code>task_handle</code> class represents an individual parallel work item. It encapsulates the instructions and the data required to execute a piece of work.</p>
<a href="#">task_options Class (Concurrency Runtime)</a>	<p>Represents the allowed options for creating a task</p>
<a href="#">timer Class</a>	<p>A <code>timer</code> messaging block is a single-target <code>source_block</code> capable of sending a message to its target after a specified time period has elapsed or at specific intervals.</p>

NAME	DESCRIPTION
<a href="#">transformer Class</a>	A <code>transformer</code> messaging block is a single-target, multi-source, ordered <code>propagator_block</code> which can accept messages of one type and is capable of storing an unbounded number of messages of a different type.
<a href="#">unbounded_buffer Class</a>	An <code>unbounded_buffer</code> messaging block is a multi-target, multi-source, ordered <code>propagator_block</code> capable of storing an unbounded number of messages.
<a href="#">unsupported_os Class</a>	This class describes an exception thrown when an unsupported operating system is used.

## Structures

NAME	DESCRIPTION
<a href="#">DispatchState Structure</a>	The <code>DispatchState</code> structure is used to transfer state to the <code>ExecutionContext::Dispatch</code> method. It describes the circumstances under which the <code>Dispatch</code> method is invoked on an <code>ExecutionContext</code> interface.
<a href="#">ExecutionContext Structure</a>	An interface to an execution context which can run on a given virtual processor and be cooperatively context switched.
<a href="#">ExecutionContextResource Structure</a>	An abstraction for a hardware thread.
<a href="#">IResourceManager Structure</a>	An interface to the Concurrency Runtime's Resource Manager. This is the interface by which schedulers communicate with the Resource Manager.
<a href="#">IScheduler Structure</a>	An interface to an abstraction of a work scheduler. The Concurrency Runtime's Resource Manager uses this interface to communicate with work schedulers.
<a href="#">ISchedulerProxy Structure</a>	The interface by which schedulers communicate with the Concurrency Runtime's Resource Manager to negotiate resource allocation.
<a href="#">IThreadProxy Structure</a>	An abstraction for a thread of execution. Depending on the <code>SchedulerType</code> policy key of the scheduler you create, the Resource Manager will grant you a thread proxy that is backed by either a regular Win32 thread or a user-mode schedulable (UMS) thread. UMS threads are supported on 64-bit operating systems with version Windows 7 and higher.
<a href="#">ITopologyExecutionResource Structure</a>	An interface to an execution resource as defined by the Resource Manager.

NAME	DESCRIPTION
<a href="#">ITopologyNode Structure</a>	An interface to a topology node as defined by the Resource Manager. A node contains one or more execution resources.
<a href="#">IUMSCompletionList Structure</a>	Represents a UMS completion list. When a UMS thread blocks, the scheduler's designated scheduling context is dispatched in order to make a decision of what to schedule on the underlying virtual processor root while the original thread is blocked. When the original thread unblocks, the operating system queues it to the completion list which is accessible through this interface. The scheduler can query the completion list on the designated scheduling context or any other place it searches for work.
<a href="#">IUMSScheduler Structure</a>	An interface to an abstraction of a work scheduler that wants the Concurrency Runtime's Resource Manager to hand it user-mode schedulable (UMS) threads. The Resource Manager uses this interface to communicate with UMS thread schedulers. The <code>IUMSScheduler</code> interface inherits from the <code>IScheduler</code> interface.
<a href="#">IUMSThreadProxy Structure</a>	An abstraction for a thread of execution. If you want your scheduler to be granted user-mode schedulable (UMS) threads, set the value for the scheduler policy element <code>SchedulerKind</code> to <code>UmsThreadDefault</code> , and implement the <code>IUMSScheduler</code> interface. UMS threads are only supported on 64-bit operating systems with version Windows 7 and higher.
<a href="#">IUMSUnblockNotification Structure</a>	Represents a notification from the Resource Manager that a thread proxy which blocked and triggered a return to the scheduler's designated scheduling context has unblocked and is ready to be scheduled. This interface is invalid once the thread proxy's associated execution context, returned from the <code>GetContext</code> method, is rescheduled.
<a href="#">IVirtualProcessorRoot Structure</a>	An abstraction for a hardware thread on which a thread proxy can execute.
<a href="#">scheduler_interface Structure</a>	Scheduler Interface
<a href="#">scheduler_ptr Structure (Concurrency Runtime)</a>	Represents a pointer to a scheduler. This class exists to allow the specification of a shared lifetime by using <code>shared_ptr</code> or just a plain reference by using raw pointer.

## Enumerations

NAME	DESCRIPTION
<a href="#">agent_status</a>	The valid states for an <code>agent</code> .
<a href="#">Agents_EventType</a>	The types of events that can be traced using the tracing functionality offered by the Agents Library

NAME	DESCRIPTION
<a href="#">ConcRT_EventType</a>	The types of events that can be traced using the tracing functionality offered by the Concurrency Runtime.
<a href="#">Concrt_TraceFlags</a>	Trace flags for the event types
<a href="#">CriticalRegionType</a>	The type of critical region a context is inside.
<a href="#">DynamicProgressFeedbackType</a>	Used by the <a href="#">DynamicProgressFeedback</a> policy to describe whether resources for the scheduler will be rebalanced according to statistical information gathered from the scheduler or only based on virtual processors going in and out of the idle state through calls to the <a href="#">Activate</a> and <a href="#">Deactivate</a> methods on the <a href="#">IVirtualProcessorRoot</a> interface. For more information on available scheduler policies, see <a href="#">PolicyElementKey</a> .
<a href="#">join_type</a>	The type of a <a href="#">join</a> messaging block.
<a href="#">message_status</a>	The valid responses for an offer of a <a href="#">message</a> object to a block.
<a href="#">PolicyElementKey</a>	Policy keys describing aspects of scheduler behavior. Each policy element is described by a key-value pair. For more information about scheduler policies and their impact on schedulers, see <a href="#">Task Scheduler</a> .
<a href="#">SchedulerType</a>	Used by the <a href="#">SchedulerKind</a> policy to describe the type of threads that the scheduler should utilize for underlying execution contexts. For more information on available scheduler policies, see <a href="#">PolicyElementKey</a> .
<a href="#">SchedulingProtocolType</a>	Used by the <a href="#">SchedulingProtocol</a> policy to describe which scheduling algorithm will be utilized for the scheduler. For more information on available scheduler policies, see <a href="#">PolicyElementKey</a> .
<a href="#">SwitchingProxyState</a>	Used to denote the state a thread proxy is in, when it is executing a cooperative context switch to a different thread proxy.
<a href="#">task_group_status</a>	Describes the execution status of a <a href="#">task_group</a> or <a href="#">structured_task_group</a> object. A value of this type is returned by numerous methods that wait on tasks scheduled to a task group to complete.
<a href="#">WinRTInitializationType</a>	Used by the <a href="#">WinRTInitialization</a> policy to describe whether and how the Windows Runtime will be initialized on scheduler threads for an application which runs on operating systems with version Windows 8 or higher. For more information on available scheduler policies, see <a href="#">PolicyElementKey</a> .

## Functions

NAME	DESCRIPTION
Alloc Function	Allocates a block of memory of the size specified from the Concurrency Runtime Caching Suballocator.
asend Function	Overloaded. An asynchronous send operation, which schedules a task to propagate the data to the target block.
cancel_current_task Function	<p>Cancels the currently executing task. This function can be called from within the body of a task to abort the task's execution and cause it to enter the <code> canceled </code> state.</p> <p>It is not a supported scenario to call this function if you are not within the body of a <code> task </code>. Doing so will result in undefined behavior such as a crash or a hang in your application.</p>
create_async Function	<p>Creates a Windows Runtime asynchronous construct based on a user supplied lambda or function object. The return type of <code> create_async </code> is one of either</p> <pre data-bbox="906 869 1442 1025"> IAsyncAction^, IAsyncActionWithProgress&lt;TProgress&gt;^, IAsyncOperation&lt;TResult&gt;^, OR IAsyncOperationWithProgress&lt;TResult, TProgress&gt;^ </pre> <p>based on the signature of the lambda passed to the method.</p>
create_task Function	Overloaded. Creates a PPL <code> task </code> object. <code> create_task </code> can be used anywhere you would have used a task constructor. It is provided mainly for convenience, because it allows use of the <code> auto </code> keyword while creating tasks.
CreateResourceManager Function	Returns an interface that represents the singleton instance of the Concurrency Runtime's Resource Manager. The Resource Manager is responsible for assigning resources to schedulers that want to cooperate with each other.
DisableTracing Function	Disables tracing in the Concurrency Runtime. This function is deprecated because ETW tracing is unregistered by default.
EnableTracing Function	Enables tracing in the Concurrency Runtime. This function is deprecated because ETW tracing is now on by default.
Free Function	Releases a block of memory previously allocated by the <code> Alloc </code> method to the Concurrency Runtime Caching Suballocator.
get_ambient_scheduler Function (Concurrency Runtime)	

NAME	DESCRIPTION
GetExecutionContextId Function	Returns a unique identifier that can be assigned to an execution context that implements the <code>IExecutionContext</code> interface.
GetOSVersion Function	Returns the operating system version.
GetProcessorCount Function	Returns the number of hardware threads on the underlying system.
GetProcessorNodeCount Function	Returns the number of NUMA nodes or processor packages on the underlying system.
GetSchedulerId Function	Returns a unique identifier that can be assigned to a scheduler that implements the <code>IScheduler</code> interface.
interruption_point Function	Creates an interruption point for cancellation. If a cancellation is in progress in the context where this function is called, this will throw an internal exception that aborts the execution of the currently executing parallel work. If cancellation is not in progress, the function does nothing.
is_current_task_group_canceling Function	Returns an indication of whether the task group which is currently executing inline on the current context is in the midst of an active cancellation (or will be shortly). Note that if there is no task group currently executing inline on the current context, <code>false</code> will be returned.
make_choice Function	Overloaded. Constructs a <code>choice</code> messaging block from an optional <code>Scheduler</code> or <code>ScheduleGroup</code> and two or more input sources.
make_greedy_join Function	Overloaded. Constructs a <code>greedy_multitype_join</code> messaging block from an optional <code>Scheduler</code> or <code>ScheduleGroup</code> and two or more input sources.
make_join Function	Overloaded. Constructs a <code>non_greedy_multitype_join</code> messaging block from an optional <code>Scheduler</code> or <code>ScheduleGroup</code> and two or more input sources.
make_task Function	A factory method for creating a <code>task_handle</code> object.
parallel_buffered_sort Function	Overloaded. Arranges the elements in a specified range into a nondescending order, or according to an ordering criterion specified by a binary predicate, in parallel. This function is semantically similar to <code>std::sort</code> in that it is a compare-based, unstable, in-place sort except that it needs $O(n)$ additional space, and requires default initialization for the elements being sorted.

NAME	DESCRIPTION
<a href="#">parallel_for Function</a>	Overloaded. <code>parallel_for</code> iterates over a range of indices and executes a user-supplied function at each iteration, in parallel.
<a href="#">parallel_for_each Function</a>	Overloaded. <code>parallel_for_each</code> applies a specified function to each element within a range, in parallel. It is semantically equivalent to the <code>for_each</code> function in the <code>std</code> namespace, except that iteration over the elements is performed in parallel, and the order of iteration is unspecified. The argument <code>_Func</code> must support a function call operator of the form <code>operator()(T)</code> where the parameter <code>T</code> is the item type of the container being iterated over.
<a href="#">parallel_invoke Function</a>	Overloaded. Executes the function objects supplied as parameters in parallel, and blocks until they have finished executing. Each function object could be a lambda expression, a pointer to function, or any object that supports the function call operator with the signature <code>void operator()()</code> .
<a href="#">parallel_radixsort Function</a>	Overloaded. Arranges elements in a specified range into a non descending order using a radix sorting algorithm. This is a stable sort function which requires a projection function that can project elements to be sorted into unsigned integer-like keys. Default initialization is required for the elements being sorted.
<a href="#">parallel_reduce Function</a>	Overloaded. Computes the sum of all elements in a specified range by computing successive partial sums, or computes the result of successive partial results similarly obtained from using a specified binary operation other than sum, in parallel. <code>parallel_reduce</code> is semantically similar to <code>std::accumulate</code> , except that it requires the binary operation to be associative, and requires an identity value instead of an initial value.
<a href="#">parallel_sort Function</a>	Overloaded. Arranges the elements in a specified range into a nondescending order, or according to an ordering criterion specified by a binary predicate, in parallel. This function is semantically similar to <code>std::sort</code> in that it is a compare-based, unstable, in-place sort.
<a href="#">parallel_transform Function</a>	Overloaded. Applies a specified function object to each element in a source range, or to a pair of elements from two source ranges, and copies the return values of the function object into a destination range, in parallel. This functional is semantically equivalent to <code>std::transform</code> .
<a href="#">receive Function</a>	Overloaded. A general receive implementation, allowing a context to wait for data from exactly one source and filter the values that are accepted.

NAME	DESCRIPTION
<a href="#">run_with_cancellation_token</a> Function	Executes a function object immediately and synchronously in the context of a given cancellation token.
<a href="#">send</a> Function	Overloaded. A synchronous send operation, which waits until the target either accepts or declines the message.
<a href="#">set_ambient_scheduler</a> Function (Concurrency Runtime)	
<a href="#">set_task_execution_resources</a> Function	<p>Overloaded. Restricts the execution resources used by the Concurrency Runtime internal worker threads to the affinity set specified.</p> <p>It is valid to call this method only before the Resource Manager has been created, or between two Resource Manager lifetimes. It can be invoked multiple times as long as the Resource Manager does not exist at the time of invocation. After an affinity limit has been set, it remains in effect until the next valid call to the <code>set_task_execution_resources</code> method.</p> <p>The affinity mask provided need not be a subset of the process affinity mask. The process affinity will be updated if necessary.</p>
<a href="#">swap</a> Function	Exchanges the elements of two <code>concurrent_vector</code> objects.
<a href="#">task_from_exception</a> Function (Concurrency Runtime)	
<a href="#">task_from_result</a> Function (Concurrency Runtime)	
<a href="#">Trace_agents_register_name</a> Function	Associates the given name to the message block or agent in the ETW trace.
<a href="#">try_receive</a> Function	Overloaded. A general try-receive implementation, allowing a context to look for data from exactly one source and filter the values that are accepted. If the data is not ready, the method will return false.
<a href="#">wait</a> Function	Pauses the current context for a specified amount of time.
<a href="#">when_all</a> Function	Creates a task that will complete successfully when all of the tasks supplied as arguments complete successfully.
<a href="#">when_any</a> Function	Overloaded. Creates a task that will complete successfully when any of the tasks supplied as arguments completes successfully.

## Operators

NAME	DESCRIPTION
<code>operator!=</code>	Tests if the <code>concurrent_vector</code> object on the left side of the operator is not equal to the <code>concurrent_vector</code> object on the right side.
<code>operator&amp;&amp;</code>	Overloaded. Creates a task that will complete successfully when both of the tasks supplied as arguments complete successfully.
<code>operator  </code>	Overloaded. Creates a task that will complete successfully when either of the tasks supplied as arguments completes successfully.
<code>operator&lt;</code>	Tests if the <code>concurrent_vector</code> object on the left side of the operator is less than the <code>concurrent_vector</code> object on the right side.
<code>operator&lt;=</code>	Tests if the <code>concurrent_vector</code> object on the left side of the operator is less than or equal to the <code>concurrent_vector</code> object on the right side.
<code>operator==</code>	Tests if the <code>concurrent_vector</code> object on the left side of the operator is equal to the <code>concurrent_vector</code> object on the right side.
<code>operator&gt;</code>	Tests if the <code>concurrent_vector</code> object on the left side of the operator is greater than the <code>concurrent_vector</code> object on the right side.
<code>operator&gt;=</code>	Tests if the <code>concurrent_vector</code> object on the left side of the operator is greater than or equal to the <code>concurrent_vector</code> object on the right side.

## Constants

NAME	DESCRIPTION
<code>AgentEventGuid</code>	A category GUID ({B9B5B78C-0713-4898-A21A-C67949DCE07}) describing ETW events fired by the Agents library in the Concurrency Runtime.
<code>ChoreEventGuid</code>	A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to chores or tasks.
<code>ConcRT_ProviderGuid</code>	The ETW provider GUID for the Concurrency Runtime.
<code>CONCRT_RM_VERSION_1</code>	Indicates support of the Resource Manager interface defined in Visual Studio 2010.
<code>ConcRTEventGuid</code>	A category GUID describing ETW events fired by the Concurrency Runtime that are not more specifically described by another category.

NAME	DESCRIPTION
<a href="#">ContextEventGuid</a>	A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to contexts.
<a href="#">COOPERATIVE_TIMEOUT_INFINITE</a>	Value indicating that a wait should never time out.
<a href="#">COOPERATIVE_WAIT_TIMEOUT</a>	Value indicating that a wait timed out.
<a href="#">INHERIT_THREAD_PRIORITY</a>	Special value for the policy key <code>ContextPriority</code> indicating that the thread priority of all contexts in the scheduler should be the same as that of the thread which created the scheduler.
<a href="#">LockEventGuid</a>	A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to locks.
<a href="#">MaxExecutionResources</a>	Special value for the policy keys <code>MinConcurrency</code> and <code>MaxConcurrency</code> . Defaults to the number of hardware threads on the machine in the absence of other constraints.
<a href="#">PPLParallelForeachEventGuid</a>	A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to usage of the <code>parallel_for_each</code> function.
<a href="#">PPLParallelForEventGuid</a>	A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to usage of the <code>parallel_for</code> function.
<a href="#">PPLParallelInvokeEventGuid</a>	A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to usage of the <code>parallel_invoke</code> function.
<a href="#">ResourceManagerEventGuid</a>	A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to the resource manager.
<a href="#">ScheduleGroupEventGuid</a>	A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to schedule groups.
<a href="#">SchedulerEventGuid</a>	A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to scheduler activity.
<a href="#">VirtualProcessorEventGuid</a>	A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to virtual processors.

## Requirements

**Header:** `agents.h`, `concr.h`, `concrtrm.h`, `concurrent_priority_queue.h`, `concurrent_queue.h`, `concurrent_unordered_map.h`, `concurrent_unordered_set.h`, `concurrent_vector.h`, `internal_concurrent_hash.h`, `internal_split_ordered_list.h`, `ppl.h`, `pplcancellation_token.h`, `pplconcr.h`,

pplinterface.h, pptasks.h

## See also

[Reference](#)

# concurrency namespace functions

3/4/2019 • 38 minutes to read • [Edit Online](#)

Alloc	CreateResourceManager	DisableTracing
EnableTracing	Free	GetExecutionContextId
GetOSVersion	GetProcessorCount	GetProcessorNodeCount
GetSchedulerId	Trace_agents_register_name	asend
cancel_current_task	clear	create_async
create_task	get_ambient_scheduler	internal_assign_iterators
interruption_point	is_current_task_group_canceling	make_choice
make_greedy_join	make_join	make_task
parallel_buffered_sort	parallel_for	parallel_for_each
parallel_invoke	parallel_radixsort	parallel_reduce
parallel_sort	parallel_transform	receive
run_with_cancellation_token	send	set_ambient_scheduler
set_task_execution_resources	swap	task_from_exception
task_from_result	try_receive	wait
when_all	when_any	

## Alloc

Allocates a block of memory of the size specified from the Concurrency Runtime Caching Suballocator.

```
void* __cdecl Alloc(size_t _NumBytes);
```

### Parameters

*\_NumBytes*

The number of bytes of memory to allocate.

### Return Value

A pointer to newly allocated memory.

### Remarks

For more information about which scenarios in your application could benefit from using the Caching Suballocator, see [Task Scheduler](#).

## asend

An asynchronous send operation, which schedules a task to propagate the data to the target block.

```
template <class T>
bool asend(
    _Inout_ ITarget<T>* _Trg,
    const T& _Data);

template <class T>
bool asend(
    ITarget<T>& _Trg,
    const T& _Data);
```

### Parameters

*T*

The type of the data to be sent.

*\_Trg*

A pointer or reference to the target to which data is sent.

*\_Data*

A reference to the data to be sent.

### Return Value

**true** if the message was accepted before the method returned, **false** otherwise.

### Remarks

For more information, see [Message Passing Functions](#).

## cancel\_current\_task

Cancels the currently executing task. This function can be called from within the body of a task to abort the task's execution and cause it to enter the `canceled` state.

It is not a supported scenario to call this function if you are not within the body of a `task`. Doing so will result in undefined behavior such as a crash or a hang in your application.

```
inline __declspec(noreturn) void __cdecl cancel_current_task();
```

## clear

Clears the concurrent queue, destroying any currently enqueued elements. This method is not concurrency-safe.

```
template<typename T, class _Ax>
void concurrent_queue<T, _Ax>::clear();
```

### Parameters

*T*

*\_Ax*

# create\_async

Creates a Windows Runtime asynchronous construct based on a user supplied lambda or function object. The return type of `create_async` is one of either `IAsyncAction^`, `IAsyncActionWithProgress<TProgress>^`, `IAsyncOperation<TResult>^`, or `IAsyncOperationWithProgress<TResult, TProgress>^` based on the signature of the lambda passed to the method.

```
template<typename _Function>
_declspec(noinline) auto create_async(const _Function& _Func)
-> decltype(ref new details::_AsyncTaskGeneratorThunk<_Function>(_Func));
```

## Parameters

`_Function`

Type.

`_Func`

The lambda or function object from which to create a Windows Runtime asynchronous construct.

## Return Value

An asynchronous construct represented by an `IAsyncAction^`, `IAsyncActionWithProgress<TProgress>^`, `IAsyncOperation<TResult>^`, or an `IAsyncOperationWithProgress<TResult, TProgress>^`. The interface returned depends on the signature of the lambda passed into the function.

## Remarks

The return type of the lambda determines whether the construct is an action or an operation.

Lambdas that return void cause the creation of actions. Lambdas that return a result of type `TResult` cause the creation of operations of `TResult`.

The lambda may also return a `task<TResult>` which encapsulates the asynchronous work within itself or is the continuation of a chain of tasks that represent the asynchronous work. In this case, the lambda itself is executed inline, since the tasks are the ones that execute asynchronously, and the return type of the lambda is unwrapped to produce the asynchronous construct returned by `create_async`. This implies that a lambda that returns a `task<void>` will cause the creation of actions, and a lambda that returns a `task<TResult>` will cause the creation of operations of `TResult`.

The lambda may take either zero, one or two arguments. The valid arguments are `progress_reporter<TProgress>` and `cancellation_token`, in that order if both are used. A lambda without arguments causes the creation of an asynchronous construct without the capability for progress reporting. A lambda that takes a `progress_reporter<TProgress>` will cause `create_async` to return an asynchronous construct which reports progress of type `TProgress` each time the `report` method of the `progress_reporter` object is called. A lambda that takes a `cancellation_token` may use that token to check for cancellation, or pass it to tasks that it creates so that cancellation of the asynchronous construct causes cancellation of those tasks.

If the body of the lambda or function object returns a result (and not a `task<TResult>`), the lambda will be executed asynchronously within the process MTA in the context of a task the Runtime implicitly creates for it. The `IAsyncInfo::Cancel` method will cause cancellation of the implicit task.

If the body of the lambda returns a task, the lambda executes inline, and by declaring the lambda to take an argument of type `cancellation_token` you can trigger cancellation of any tasks you create within the lambda by passing that token in when you create them. You may also use the `register_callback` method on the token to cause the Runtime to invoke a callback when you call `IAsyncInfo::Cancel` on the async operation or action produced.

This function is only available to Windows Runtime apps.

# CreateResourceManager

Returns an interface that represents the singleton instance of the Concurrency Runtime's Resource Manager. The Resource Manager is responsible for assigning resources to schedulers that want to cooperate with each other.

```
IResourceManager* __cdecl CreateResourceManager();
```

## Return Value

An `IResourceManager` interface.

## Remarks

Multiple subsequent calls to this method will return the same instance of the Resource Manager. Each call to the method increments a reference count on the Resource Manager, and must be matched with a call to the [IResourceManager::Release](#) method when your scheduler is done communicating with the Resource Manager.

`unsupported_os` is thrown if the operating system is not supported by the Concurrency Runtime.

# create\_task

Creates a PPL `task` object. `create_task` can be used anywhere you would have used a task constructor. It is provided mainly for convenience, because it allows use of the `auto` keyword while creating tasks.

```
template<typename T>
__declspec(noinline) auto create_task(T _Param, const task_options& _TaskOptions = task_options())
    -> task<typename details::_TaskTypeFromParam<T>::T>;

template<typename _ReturnType>
__declspec( noinline) task<_ReturnType> create_task(const task<_ReturnType>& _Task);
```

## Parameters

*T*

The type of the parameter from which the task is to be constructed.

*\_ReturnType*

Type.

*\_Param*

The parameter from which the task is to be constructed. This could be a lambda or function object, a

`task_completion_event` object, a different `task` object, or a `Windows::Foundation::IAsyncInfo` interface if you are using tasks in your UWP app.

*\_TaskOptions*

The task options.

*\_Task*

The task to create.

## Return Value

A new task of type `T`, that is inferred from `_Param`.

## Remarks

The first overload behaves like a task constructor that takes a single parameter.

The second overload associates the cancellation token provided with the newly created task. If you use this overload you are not allowed to pass in a different `task` object as the first parameter.

The type of the returned task is inferred from the first parameter to the function. If `_Param` is a `task_completion_event<T>`, a `task<T>`, or a functor that returns either type `T` or `task<T>`, the type of the created task is `task<T>`.

In a UWP app, if `_Param` is of type `Windows::Foundation::IAsyncOperation<T>^` or `Windows::Foundation::IAsyncOperationWithProgress<T,P>^`, or a functor that returns either of those types, the created task will be of type `task<T>`. If `_Param` is of type `Windows::Foundation::IAsyncAction^` or `Windows::Foundation::IAsyncActionWithProgress<P>^`, or a functor that returns either of those types, the created task will have type `task<void>`.

## DisableTracing

Disables tracing in the Concurrency Runtime. This function is deprecated because ETW tracing is unregistered by default.

```
__declspec(deprecated("Concurrency::DisableTracing is a deprecated function. ")) _CRTIMP HRESULT __cdecl  
DisableTracing();
```

### Return Value

If tracing was correctly disabled, `S_OK` is returned. If tracing was not previously initiated, `E_NOT_STARTED` is returned.

## EnableTracing

Enables tracing in the Concurrency Runtime. This function is deprecated because ETW tracing is now on by default.

```
__declspec(deprecated("Concurrency::EnableTracing is a deprecated function. ")) _CRTIMP HRESULT __cdecl  
EnableTracing();
```

### Return Value

If tracing was correctly initiated, `S_OK` is returned; otherwise, `E_NOT_STARTED` is returned.

## Free

Releases a block of memory previously allocated by the `Alloc` method to the Concurrency Runtime Caching Suballocator.

```
void __cdecl Free(_Pre_maybenull_ _Post_invalid_ void* _PAllocation);
```

### Parameters

*\_PAllocation*

A pointer to memory previously allocated by the `Alloc` method which is to be freed. If the parameter `_PAllocation` is set to the value `NULL`, this method will ignore it and return immediately.

### Remarks

For more information about which scenarios in your application could benefit from using the Caching Suballocator, see [Task Scheduler](#).

## get\_ambient\_scheduler

```
inline std::shared_ptr<Concurrency::scheduler_interface> get_ambient_scheduler();
```

### Return Value

## GetExecutionContextId

Returns a unique identifier that can be assigned to an execution context that implements the `IEExecutionContext` interface.

```
unsigned int __cdecl GetExecutionContextId();
```

### Return Value

A unique identifier for an execution context.

### Remarks

Use this method to obtain an identifier for your execution context before you pass an `IEExecutionContext` interface as a parameter to any of the methods offered by the Resource Manager.

## GetOSVersion

Returns the operating system version.

```
IResourceManager::OSVersion __cdecl GetOSVersion();
```

### Return Value

An enumerated value representing the operating system.

### Remarks

[unsupported\\_os](#) is thrown if the operating system is not supported by the Concurrency Runtime.

## GetProcessorCount

Returns the number of hardware threads on the underlying system.

```
unsigned int __cdecl GetProcessorCount();
```

### Return Value

The number of hardware threads.

### Remarks

[unsupported\\_os](#) is thrown if the operating system is not supported by the Concurrency Runtime.

## GetProcessorNodeCount

Returns the number of NUMA nodes or processor packages on the underlying system.

```
unsigned int __cdecl GetProcessorNodeCount();
```

### Return Value

The number of NUMA nodes or processor packages.

## Remarks

If the system contains more NUMA nodes than processor packages, the number of NUMA nodes is returned, otherwise, the number of processor packages is returned.

`unsupported_os` is thrown if the operating system is not supported by the Concurrency Runtime.

## GetSchedulerId

Returns a unique identifier that can be assigned to a scheduler that implements the `IScheduler` interface.

```
unsigned int __cdecl GetSchedulerId();
```

## Return Value

A unique identifier for a scheduler.

## Remarks

Use this method to obtain an identifier for your scheduler before you pass an `IScheduler` interface as a parameter to any of the methods offered by the Resource Manager.

## internal\_assign\_iterators

```
template<typename T, class _Ax>
template<class _I>
void concurrent_vector<T, _Ax>::internal_assign_iterators(
    _I first,
    _I last);
```

## Parameters

*T*

*\_Ax*

*\_I*

*first*

*last*

## interruption\_point

Creates an interruption point for cancellation. If a cancellation is in progress in the context where this function is called, this will throw an internal exception that aborts the execution of the currently executing parallel work. If cancellation is not in progress, the function does nothing.

```
inline void interruption_point();
```

## Remarks

You should not catch the internal cancellation exception thrown by the `interruption_point()` function. The exception will be caught and handled by the runtime, and catching it may cause your program to behave abnormally.

## is\_current\_task\_group\_canceling

Returns an indication of whether the task group which is currently executing inline on the current context is in the midst of an active cancellation (or will be shortly). Note that if there is no task group currently executing inline on the current context, `false` will be returned.

```
bool __cdecl is_current_task_group_canceling();
```

### Return Value

**true** if the task group which is currently executing is canceling, **false** otherwise.

### Remarks

For more information, see [Cancellation](#).

## make\_choice

Constructs a `choice` messaging block from an optional `Scheduler` or `ScheduleGroup` and two or more input sources.

```
template<typename T1, typename T2, typename... Ts>
choice<std::tuple<T1, T2, Ts...>> make_choice(
    Scheduler& _PScheduler,
    T1 _Item1,
    T2 _Item2,
    Ts... _Items);

template<typename T1, typename T2, typename... Ts>
choice<std::tuple<T1, T2, Ts...>> make_choice(
    ScheduleGroup& _PScheduleGroup,
    T1 _Item1,
    T2 _Item2,
    Ts... _Items);

template<typename T1, typename T2, typename... Ts>
choice<std::tuple<T1, T2, Ts...>> make_choice(
    T1 _Item1,
    T2 _Item2,
    Ts... _Items);
```

### Parameters

*T1*

The message block type of the first source.

*T2*

The message block type of the second source.

*\_PScheduler*

The `Scheduler` object within which the propagation task for the `choice` messaging block is scheduled.

*\_Item1*

The first source.

*\_Item2*

The second source.

*\_Items*

Additional sources.

*\_PScheduleGroup*

The `ScheduleGroup` object within which the propagation task for the `choice` messaging block is scheduled. The

`Scheduler` object used is implied by the schedule group.

### Return Value

A `choice` message block with two or more input sources.

## make\_greedy\_join

Constructs a `greedy multitype_join` messaging block from an optional `Scheduler` or `ScheduleGroup` and two or more input sources.

```
template<typename T1, typename T2, typename... Ts>
multitype_join<std::tuple<T1, T2, Ts...>,greedy> make_greedy_join(
    Scheduler& _PScheduler,
    T1 _Item1,
    T2 _Item2,
    Ts... _Items);

template<typename T1, typename T2, typename... Ts>
multitype_join<std::tuple<T1, T2, Ts...>, greedy> make_greedy_join(
    ScheduleGroup& _PScheduleGroup,
    T1 _Item1,
    T2 _Item2,
    Ts... _Items);

template<typename T1, typename T2, typename... Ts>
multitype_join<std::tuple<T1, T2, Ts...>, greedy> make_greedy_join(
    T1 _Item1,
    T2 _Items,
    Ts... _Items);
```

### Parameters

*T1*

The message block type of the first source.

*T2*

The message block type of the second source.

*\_PScheduler*

The `Scheduler` object within which the propagation task for the `multitype_join` messaging block is scheduled.

*\_Item1*

The first source.

*\_Item2*

The second source.

*\_Items*

Additional sources.

*\_PScheduleGroup*

The `ScheduleGroup` object within which the propagation task for the `multitype_join` messaging block is scheduled. The `Scheduler` object used is implied by the schedule group.

### Return Value

A `greedy multitype_join` message block with two or more input sources.

## make\_join

Constructs a `non_greedy multitype_join` messaging block from an optional `Scheduler` or `ScheduleGroup` and two

or more input sources.

```
template<typename T1, typename T2, typename... Ts>
multitype_join<std::tuple<T1, T2, Ts...>>
    make_join(
Scheduler& _PScheduler,
    T1 _Item1,
    T2 _Item2,
    Ts... _Items);

template<typename T1, typename T2, typename... Ts>
multitype_join<std::tuple<T1, T2, Ts...>> make_join(
ScheduleGroup& _PScheduleGroup,
    T1 _Item1,
    T2 _Item2,
    Ts... _Items);

template<typename T1, typename T2, typename... Ts>
multitype_join<std::tuple<T1, T2, Ts...>> make_join(
    T1 _Item1,
    T2 _Item2,
    Ts... _Items);
```

### Parameters

*T1*

The message block type of the first source.

*T2*

The message block type of the second source.

*\_PScheduler*

The `Scheduler` object within which the propagation task for the `multitype_join` messaging block is scheduled.

*\_Item1*

The first source.

*\_Item2*

The second source.

*\_Items*

Additional sources.

*\_PScheduleGroup*

The `ScheduleGroup` object within which the propagation task for the `multitype_join` messaging block is scheduled. The `Scheduler` object used is implied by the schedule group.

### Return Value

A `non_greedy multitype_join` message block with two or more input sources.

## make\_task

A factory method for creating a `task_handle` object.

```
template <class _Function>
task_handle<_Function> make_task(const _Function& _Func);
```

### Parameters

*\_Function*

The type of the function object that will be invoked to execute the work represented by the `task_handle` object.

*\_Func*

The function that will be invoked to execute the work represented by the `task_handle` object. This may be a lambda functor, a pointer to a function, or any object that supports a version of the function call operator with the signature `void operator()()`.

#### **Return Value**

A `task_handle` object.

#### **Remarks**

This function is useful when you need to create a `task_handle` object with a lambda expression, because it allows you to create the object without knowing the true type of the lambda functor.

## **parallel\_buffered\_sort**

Arranges the elements in a specified range into a nondescending order, or according to an ordering criterion specified by a binary predicate, in parallel. This function is semantically similar to `std::sort` in that it is a compare-based, unstable, in-place sort except that it needs  $O(n)$  additional space, and requires default initialization for the elements being sorted.

```

template<typename _Random_iterator>
inline void parallel_buffered_sort(
    const _Random_iterator& _Begin,
    const _Random_iterator& _End);

template<typename _Allocator,
        typename _Random_iterator>
inline void parallel_buffered_sort(
    const _Random_iterator& _Begin,
    const _Random_iterator& _End);

template<typename _Allocator,
        typename _Random_iterator>
inline void parallel_buffered_sort(
    const _Allocator& _Alloc,
    const _Random_iterator& _Begin,
    const _Random_iterator& _End);

template<typename _Random_iterator,
        typename _Function>
inline void parallel_buffered_sort(
    const _Random_iterator& _Begin,
    const _Random_iterator& _End,
    const _Function& _Func,
    const size_t _Chunk_size = 2048);

template<typename _Allocator,
        typename _Random_iterator,
        typename _Function>
inline void parallel_buffered_sort(
    const _Random_iterator& _Begin,
    const _Random_iterator& _End,
    const _Function& _Func,
    const size_t _Chunk_size = 2048);

template<typename _Allocator,
        typename _Random_iterator,
        typename _Function>
inline void parallel_buffered_sort(
    const _Allocator& _Alloc,
    const _Random_iterator& _Begin,
    const _Random_iterator& _End,
    const _Function& _Func,
    const size_t _Chunk_size = 2048);

```

## Parameters

### *\_Random\_iterator*

The iterator type of the input range.

### *\_Allocator*

The type of a C++ Standard Library compatible memory allocator.

### *\_Function*

The type of the binary comparator.

### *\_Begin*

A random-access iterator addressing the position of the first element in the range to be sorted.

### *\_End*

A random-access iterator addressing the position one past the final element in the range to be sorted.

### *\_Alloc*

An instance of a C++ Standard Library compatible memory allocator.

`_Func`

A user-defined predicate function object that defines the comparison criterion to be satisfied by successive elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied. This comparator function must impose a strict weak ordering on pairs of elements from the sequence.

`_Chunk_size`

The minimum size of a chunk that will be split into two for parallel execution.

### Remarks

All overloads require `n * sizeof(T)` additional space, where `n` is the number of elements to be sorted, and `T` is the element type. In most cases `parallel_buffered_sort` will show an improvement in performance over `parallel_sort`, and you should use it over `parallel_sort` if you have the memory available.

If you do not supply a binary comparator `std::less` is used as the default, which requires the element type to provide the operator `operator<()`.

If you do not supply an allocator type or instance, the C++ Standard Library memory allocator `std::allocator<T>` is used to allocate the buffer.

The algorithm divides the input range into two chunks and successively divides each chunk into two sub-chunks for execution in parallel. The optional argument `_Chunk_size` can be used to indicate to the algorithm that it should handle chunks of size `< _Chunk_size` serially.

## parallel\_for

`parallel_for` iterates over a range of indices and executes a user-supplied function at each iteration, in parallel.

```

template <typename _Index_type, typename _Function, typename _Partitioner>
void parallel_for(
    _Index_type first,
    _Index_type last,
    _Index_type _Step,
    const _Function& _Func,
    _Partitioner&& _Part);

template <typename _Index_type, typename _Function>
void parallel_for(
    _Index_type first,
    _Index_type last,
    _Index_type _Step,
    const _Function& _Func);

template <typename _Index_type, typename _Function>
void parallel_for(
    _Index_type first,
    _Index_type last,
    const _Function& _Func,
    const auto_partitioner& _Part = auto_partitioner());

template <typename _Index_type, typename _Function>
void parallel_for(
    _Index_type first,
    _Index_type last,
    const _Function& _Func,
    const static_partitioner& _Part);

template <typename _Index_type, typename _Function>
void parallel_for(
    _Index_type first,
    _Index_type last,
    const _Function& _Func,
    const simple_partitioner& _Part);

template <typename _Index_type, typename _Function>
void parallel_for(
    _Index_type first,
    _Index_type last,
    const _Function& _Func,
    affinity_partitioner& _Part);

```

## Parameters

### *\_Index\_type*

The type of the index being used for the iteration.

### *\_Function*

The type of the function that will be executed at each iteration.

### *\_Partitioner*

The type of the partitioner that is used to partition the supplied range.

### *first*

The first index to be included in the iteration.

### *last*

The index one past the last index to be included in the iteration.

### *\_Step*

The value by which to step when iterating from `first` to `last`. The step must be positive. `invalid_argument` is thrown if the step is less than 1.

*\_Func*

The function to be executed at each iteration. This may be a lambda expression, a function pointer, or any object that supports a version of the function call operator with the signature `void operator()(_Index_type)`.

*\_Part*

A reference to the partitioner object. The argument can be one of `const auto_partitioner&`, `const static_partitioner&`, `const simple_partitioner&` or `affinity_partitioner&`. If an `affinity_partitioner` object is used, the reference must be a non-const l-value reference, so that the algorithm can store state for future loops to re-use.

### Remarks

For more information, see [Parallel Algorithms](#).

## parallel\_for\_each

`parallel_for_each` applies a specified function to each element within a range, in parallel. It is semantically equivalent to the `for_each` function in the `std` namespace, except that iteration over the elements is performed in parallel, and the order of iteration is unspecified. The argument `_Func` must support a function call operator of the form `operator()(T)` where the parameter `T` is the item type of the container being iterated over.

```
template <typename _Iterator, typename _Function>
void parallel_for_each(
    _Iterator first,
    _Iterator last,
    const _Function& _Func);

template <typename _Iterator, typename _Function, typename _Partitioner>
void parallel_for_each(
    _Iterator first,
    _Iterator last,
    const _Function& _Func,
    _Partitioner&& _Part);
```

### Parameters

*\_Iterator*

The type of the iterator being used to iterate over the container.

*\_Function*

The type of the function that will be applied to each element within the range.

*\_Partitioner*

*first*

An iterator addressing the position of the first element to be included in parallel iteration.

*last*

An iterator addressing the position one past the final element to be included in parallel iteration.

*\_Func*

A user-defined function object that is applied to each element in the range.

*\_Part*

A reference to the partitioner object. The argument can be one of `const auto_partitioner&`, `const static_partitioner&`, `const simple_partitioner&` or `affinity_partitioner&`. If an `affinity_partitioner` object is used, the reference must be a non-const l-value reference, so that the algorithm can store state for future loops to re-use.

### Remarks

[auto\\_partitioner](#) will be used for the overload without an explicit partitioner.

For iterators that do not support random access, only [auto\\_partitioner](#) is supported.

For more information, see [Parallel Algorithms](#).

## parallel\_invoke

Executes the function objects supplied as parameters in parallel, and blocks until they have finished executing. Each function object could be a lambda expression, a pointer to function, or any object that supports the function call operator with the signature `void operator()()`.

```
template <typename _Function1, typename _Function2>
void parallel_invoke(
    const _Function1& _Func1,
    const _Function2& _Func2);

template <typename _Function1, typename _Function2, typename _Function3>
void parallel_invoke(
    const _Function1& _Func1,
    const _Function2& _Func2,
    const _Function3& _Func3);

template <typename _Function1,
          typename _Function2,
          typename _Function3,
          typename _Function4>
void parallel_invoke(
    const _Function1& _Func1,
    const _Function2& _Func2,
    const _Function3& _Func3,
    const _Function4& _Func4);

template <typename _Function1,
          typename _Function2,
          typename _Function3,
          typename _Function4,
          typename _Function5>
void parallel_invoke(
    const _Function1& _Func1,
    const _Function2& _Func2,
    const _Function3& _Func3,
    const _Function4& _Func4,
    const _Function5& _Func5);

template <typename _Function1,
          typename _Function2,
          typename _Function3,
          typename _Function4,
          typename _Function5,
          typename _Function6>
void parallel_invoke(
    const _Function1& _Func1,
    const _Function2& _Func2,
    const _Function3& _Func3,
    const _Function4& _Func4,
    const _Function5& _Func5,
    const _Function6& _Func6);

template <typename _Function1,
          typename _Function2,
          typename _Function3,
          typename _Function4,
          typename _Function5,
          typename _Function6,
          typename _Function7>
```

```
void parallel_invoke(
    const _Function1& _Func1,
    const _Function2& _Func2,
    const _Function3& _Func3,
    const _Function4& _Func4,
    const _Function5& _Func5,
    const _Function6& _Func6,
    const _Function7& _Func7);

template <typename _Function1,
    typename _Function2,
    typename _Function3,
    typename _Function4,
    typename _Function5,
    typename _Function6,
    typename _Function7,
    typename _Function8>
void parallel_invoke(
    const _Function1& _Func1,
    const _Function2& _Func2,
    const _Function3& _Func3,
    const _Function4& _Func4,
    const _Function5& _Func5,
    const _Function6& _Func6,
    const _Function7& _Func7,
    const _Function8& _Func8);

template <typename _Function1,
    typename _Function2,
    typename _Function3,
    typename _Function4,
    typename _Function5,
    typename _Function6,
    typename _Function7,
    typename _Function8,
    typename _Function9>
void parallel_invoke(
    const _Function1& _Func1,
    const _Function2& _Func2,
    const _Function3& _Func3,
    const _Function4& _Func4,
    const _Function5& _Func5,
    const _Function6& _Func6,
    const _Function7& _Func7,
    const _Function8& _Func8,
    const _Function9& _Func9);

template <typename _Function1,
    typename _Function2,
    typename _Function3,
    typename _Function4,
    typename _Function5,
    typename _Function6,
    typename _Function7,
    typename _Function8,
    typename _Function9,
    typename _Function10>
void parallel_invoke(
    const _Function1& _Func1,
    const _Function2& _Func2,
    const _Function3& _Func3,
    const _Function4& _Func4,
    const _Function5& _Func5,
    const _Function6& _Func6,
    const _Function7& _Func7,
    const _Function8& _Func8,
    const _Function9& _Func9,
    const _Function10& _Func10);
```

---

## Parameters

*\_Function1*

The type of the first function object to be executed in parallel.

*\_Function2*

The type of the second function object to be executed in parallel.

*\_Function3*

The type of the third function object to be executed in parallel.

*\_Function4*

The type of the fourth function object to be executed in parallel.

*\_Function5*

The type of the fifth function object to be executed in parallel.

*\_Function6*

The type of the sixth function object to be executed in parallel.

*\_Function7*

The type of the seventh function object to be executed in parallel.

*\_Function8*

The type of the eighth function object to be executed in parallel.

*\_Function9*

The type of the ninth function object to be executed in parallel.

*\_Function10*

The type of the tenth function object to be executed in parallel.

*\_Func1*

The first function object to be executed in parallel.

*\_Func2*

The second function object to be executed in parallel.

*\_Func3*

The third function object to be executed in parallel.

*\_Func4*

The fourth function object to be executed in parallel.

*\_Func5*

The fifth function object to be executed in parallel.

*\_Func6*

The sixth function object to be executed in parallel.

*\_Func7*

The seventh function object to be executed in parallel.

*\_Func8*

The eighth function object to be executed in parallel.

*\_Func9*

The ninth function object to be executed in parallel.

*\_Func10*

The tenth function object to be executed in parallel.

## Remarks

Note that one or more of the function objects supplied as parameters may execute inline on the calling context.

If one or more of the function objects passed as parameters to this function throws an exception, the runtime will select one such exception of its choosing and propagate it out of the call to `parallel_invoke`.

For more information, see [Parallel Algorithms](#).

## parallel\_radixsort

Arranges elements in a specified range into a non descending order using a radix sorting algorithm. This is a stable sort function which requires a projection function that can project elements to be sorted into unsigned integer-like keys. Default initialization is required for the elements being sorted.

```
template<typename _Random_iterator>
inline void parallel_radixsort(
    const _Random_iterator& _Begin,
    const _Random_iterator& _End);

template<typename _Allocator, typename _Random_iterator>
inline void parallel_radixsort(
    const _Random_iterator& _Begin,
    const _Random_iterator& _End);

template<typename _Allocator, typename _Random_iterator>
inline void parallel_radixsort(
    const _Allocator& _Alloc,
    const _Random_iterator& _Begin,
    const _Random_iterator& _End);

template<typename _Random_iterator, typename _Function>
inline void parallel_radixsort(
    const _Random_iterator& _Begin,
    const _Random_iterator& _End,
    const _Function& _Proj_func,
    const size_t _Chunk_size = 256* 256);

template<typename _Allocator, typename _Random_iterator,
        typename _Function>
inline void parallel_radixsort(
    const _Random_iterator& _Begin,
    const _Random_iterator& _End,
    const _Function& _Proj_func,
    const size_t _Chunk_size = 256* 256);

template<typename _Allocator,
        typename _Random_iterator,
        typename _Function>
inline void parallel_radixsort(
    const _Allocator& _Alloc,
    const _Random_iterator& _Begin,
    const _Random_iterator& _End,
    const _Function& _Proj_func,
    const size_t _Chunk_size = 256* 256);
```

## Parameters

*\_Random\_iterator*

The iterator type of the input range.

*\_Allocator*

The type of a C++ Standard Library compatible memory allocator.

*\_Function*

The type of the projection function.

*\_Begin*

A random-access iterator addressing the position of the first element in the range to be sorted.

*\_End*

A random-access iterator addressing the position one past the final element in the range to be sorted.

*\_Alloc*

An instance of a C++ Standard Library compatible memory allocator.

*\_Proj\_func*

A user-defined projection function object that converts an element into an integral value.

*\_Chunk\_size*

The minimum size of a chunk that will be split into two for parallel execution.

### Remarks

All overloads require `n * sizeof(T)` additional space, where `n` is the number of elements to be sorted, and `T` is the element type. An unary projection functor with the signature `I _Proj_func(T)` is required to return a key when given an element, where `T` is the element type and `I` is an unsigned integer-like type.

If you do not supply a projection function, a default projection function which simply returns the element is used for integral types. The function will fail to compile if the element is not an integral type in the absence of a projection function.

If you do not supply an allocator type or instance, the C++ Standard Library memory allocator `std::allocator<T>` is used to allocate the buffer.

The algorithm divides the input range into two chunks and successively divides each chunk into two sub-chunks for execution in parallel. The optional argument `_Chunk_size` can be used to indicate to the algorithm that it should handle chunks of size `< _Chunk_size` serially.

## parallel\_reduce

Computes the sum of all elements in a specified range by computing successive partial sums, or computes the result of successive partial results similarly obtained from using a specified binary operation other than sum, in parallel. `parallel_reduce` is semantically similar to `std::accumulate`, except that it requires the binary operation to be associative, and requires an identity value instead of an initial value.

```

template<typename _Forward_iterator>
inline typename std::iterator_traits<_Forward_iterator>::value_type parallel_reduce(
    _Forward_iterator _Begin,
    _Forward_iterator _End,
    const typename std::iterator_traits<_Forward_iterator>::value_type& _Identity);

template<typename _Forward_iterator, typename _Sym_reduce_fun>
inline typename std::iterator_traits<_Forward_iterator>::value_type parallel_reduce(
    _Forward_iterator _Begin,
    _Forward_iterator _End,
    const typename std::iterator_traits<_Forward_iterator>::value_type& _Identity,
    _Sym_reduce_fun _Sym_fun);

template<typename _Reduce_type,
    typename _Forward_iterator,
    typename _Range_reduce_fun,
    typename _Sym_reduce_fun>
inline _Reduce_type parallel_reduce(
    _Forward_iterator _Begin,
    _Forward_iterator _End,
    const _Reduce_type& _Identity,
    const _Range_reduce_fun& _Range_fun,
    const _Sym_reduce_fun& _Sym_fun);

```

## Parameters

### *\_Forward\_iterator*

The iterator type of input range.

### *\_Sym\_reduce\_fun*

The type of the symmetric reduction function. This must be a function type with signature

`_Reduce_type _Sym_fun(_Reduce_type, _Reduce_type)`, where `_Reduce_type` is the same as the identity type and the result type of the reduction. For the third overload, this should be consistent with the output type of

`_Range_reduce_fun`.

### *\_Reduce\_type*

The type that the input will reduce to, which can be different from the input element type. The return value and identity value will have this type.

### *\_Range\_reduce\_fun*

The type of the range reduction function. This must be a function type with signature

`_Reduce_type _Range_fun(_Forward_iterator, _Forward_iterator, _Reduce_type)`, `_Reduce_type` is the same as the identity type and the result type of the reduction.

### *\_Begin*

An input iterator addressing the first element in the range to be reduced.

### *\_End*

An input iterator addressing the element that is one position beyond the final element in the range to be reduced.

### *\_Identity*

The identity value `_Identity` is of the same type as the result type of the reduction and also the `value_type` of the iterator for the first and second overloads. For the third overload, the identity value must have the same type as the result type of the reduction, but can be different from the `value_type` of the iterator. It must have an appropriate value such that the range reduction operator `_Range_fun`, when applied to a range of a single element of type `value_type` and the identity value, behaves like a type cast of the value from type `value_type` to the identity type.

### *\_Sym\_fun*

The symmetric function that will be used in the second of the reduction. Refer to Remarks for more information.

## *\_Range\_fun*

The function that will be used in the first phase of the reduction. Refer to Remarks for more information.

### Return Value

The result of the reduction.

### Remarks

To perform a parallel reduction, the function divides the range into chunks based on the number of workers available to the underlying scheduler. The reduction takes place in two phases, the first phase performs a reduction within each chunk, and the second phase performs a reduction between the partial results from each chunk.

The first overload requires that the iterator's `value_type`, `T`, be the same as the identity value type as well as the reduction result type. The element type `T` must provide the operator `T T::operator + (T)` to reduce elements in each chunk. The same operator is used in the second phase as well.

The second overload also requires that the iterator's `value_type` be the same as the identity value type as well as the reduction result type. The supplied binary operator `_Sym_fun` is used in both reduction phases, with the identity value as the initial value for the first phase.

For the third overload, the identity value type must be the same as the reduction result type, but the iterator's `value_type` may be different from both. The range reduction function `_Range_fun` is used in the first phase with the identity value as the initial value, and the binary function `_Sym_reduce_fun` is applied to sub results in the second phase.

## parallel\_sort

Arranges the elements in a specified range into a nondescending order, or according to an ordering criterion specified by a binary predicate, in parallel. This function is semantically similar to `std::sort` in that it is a compare-based, unstable, in-place sort.

```
template<typename _Random_iterator>
inline void parallel_sort(
    const _Random_iterator& _Begin,
    const _Random_iterator& _End);

template<typename _Random_iterator, typename _Function>
inline void parallel_sort(
    const _Random_iterator& _Begin,
    const _Random_iterator& _End,
    const _Function& _Func,
    const size_t _Chunk_size = 2048);
```

### Parameters

#### *\_Random\_iterator*

The iterator type of the input range.

#### *\_Function*

The type of the binary comparison functor.

#### *\_Begin*

A random-access iterator addressing the position of the first element in the range to be sorted.

#### *\_End*

A random-access iterator addressing the position one past the final element in the range to be sorted.

#### *\_Func*

A user-defined predicate function object that defines the comparison criterion to be satisfied by successive

elements in the ordering. A binary predicate takes two arguments and returns **true** when satisfied and **false** when not satisfied. This comparator function must impose a strict weak ordering on pairs of elements from the sequence.

*\_Chunk\_size*

The minimum size of a chunk that will be split into two for parallel execution.

### Remarks

The first overload uses the binary comparator `std::less`.

The second overload uses the supplied binary comparator that should have the signature `bool _Func(T, T)` where `T` is the type of the elements in the input range.

The algorithm divides the input range into two chunks and successively divides each chunk into two sub-chunks for execution in parallel. The optional argument `_Chunk_size` can be used to indicate to the algorithm that it should handle chunks of size `< _Chunk_size` serially.

## parallel\_transform

Applies a specified function object to each element in a source range, or to a pair of elements from two source ranges, and copies the return values of the function object into a destination range, in parallel. This functional is semantically equivalent to `std::transform`.

```

template <typename _Input_iterator1,
         typename _Output_iterator,
         typename _Unary_operator>
_Output_iterator parallel_transform(
    _Input_iterator1 first1,
    _Input_iterator1 last1,
    _Output_iterator _Result,
    const _Unary_operator& _Unary_op,
    const auto_partitioner& _Part = auto_partitioner());

```

```

template <typename _Input_iterator1,
         typename _Output_iterator,
         typename _Unary_operator>
_Output_iterator parallel_transform(
    _Input_iterator1 first1,
    _Input_iterator1 last1,
    _Output_iterator _Result,
    const _Unary_operator& _Unary_op,
    const static_partitioner& _Part);

```

```

template <typename _Input_iterator1,
         typename _Output_iterator,
         typename _Unary_operator>
_Output_iterator parallel_transform(
    _Input_iterator1 first1,
    _Input_iterator1 last1,
    _Output_iterator _Result,
    const _Unary_operator& _Unary_op,
    const simple_partitioner& _Part);

```

```

template <typename _Input_iterator1,
         typename _Output_iterator,
         typename _Unary_operator>
_Output_iterator parallel_transform(
    _Input_iterator1 first1,
    _Input_iterator1 last1,
    _Output_iterator _Result,
    const _Unary_operator& _Unary_op,
    affinity_partitioner& _Part);

```

```

template <typename _Input_iterator1,
         typename _Input_iterator2,
         typename _Output_iterator,
         typename _Binary_operator,
         typename _Partitioner>
_Output_iterator parallel_transform(
    _Input_iterator1 first1,
    _Input_iterator1 last1,
    _Input_iterator2
first2,
    _Output_iterator _Result,
    const _Binary_operator& _Binary_op,
    _Partitioner&& _Part);

```

```

template <typename _Input_iterator1,
         typename _Input_iterator2,
         typename _Output_iterator,
         typename _Binary_operator>
_Output_iterator parallel_transform(
    _Input_iterator1 first1,
    _Input_iterator1 last1,
    _Input_iterator2
first2,
    _Output_iterator _Result,
    const _Binary_operator& _Binary_op);

```

## Parameters

*\_Input\_iterator1*

The type of the first or only input iterator.

*\_Output\_iterator*

The type of the output iterator.

*\_Unary\_operator*

The type of the unary functor to be executed on each element in the input range.

*\_Input\_iterator2*

The type of second input iterator.

*\_Binary\_operator*

The type of the binary functor executed pairwise on elements from the two source ranges.

*\_Partitioner*

*first1*

An input iterator addressing the position of the first element in the first or only source range to be operated on.

*last1*

An input iterator addressing the position one past the final element in the first or only source range to be operated on.

*\_Result*

An output iterator addressing the position of the first element in the destination range.

*\_Unary\_op*

A user-defined unary function object that is applied to each element in the source range.

*\_Part*

A reference to the partitioner object. The argument can be one of `const auto_partitioner &`, `const static_partitioner &`, `const simple_partitioner &` or `affinity_partitioner &`. If an `affinity_partitioner` object is used, the reference must be a non-const l-value reference, so that the algorithm can store state for future loops to re-use.

*first2*

An input iterator addressing the position of the first element in the second source range to be operated on.

*\_Binary\_op*

A user-defined binary function object that is applied pairwise, in a forward order, to the two source ranges.

## Return Value

An output iterator addressing the position one past the final element in the destination range that is receiving the output elements transformed by the function object.

## Remarks

`auto_partitioner` will be used for the overloads without an explicit partitioner argument.

For iterators that do not support random access, only `auto_partitioner` is supported.

The overloads that take the argument `_Unary_op` transform the input range into the output range by applying the unary functor to each element in the input range. `_Unary_op` must support the function call operator with signature `operator()(T)` where `T` is the value type of the range being iterated over.

The overloads that take the argument `_Binary_op` transform two input ranges into the output range by applying the binary functor to one element from the first input range and one element from the second input range.

`_Binary_op` must support the function call operator with signature `operator()(T, U)` where `T`, `U` are value

types of the two input iterators.

For more information, see [Parallel Algorithms](#).

## receive

A general receive implementation, allowing a context to wait for data from exactly one source and filter the values that are accepted.

```
template <class T>
T receive(
    _Inout_ ISource<T>* _Src,
    unsigned int _Timeout = COOPERATIVE_TIMEOUT_INFINITE);

template <class T>
T receive(
    _Inout_ ISource<T>* _Src,
    typename ITarget<T>::filter_method const& _Filter_proc,
    unsigned int _Timeout = COOPERATIVE_TIMEOUT_INFINITE);

template <class T>
T receive(
    ISource<T>& _Src,
    unsigned int _Timeout = COOPERATIVE_TIMEOUT_INFINITE);

template <class T>
T receive(
    ISource<T>& _Src,
    typename ITarget<T>::filter_method const& _Filter_proc,
    unsigned int _Timeout = COOPERATIVE_TIMEOUT_INFINITE);
```

### Parameters

*T*

The payload type.

*\_Src*

A pointer or reference to the source from which data is expected.

*\_Timeout*

The maximum time for which the method should for the data, in milliseconds.

*\_Filter\_proc*

A filter function which determines whether messages should be accepted.

### Return Value

A value from the source, of the payload type.

### Remarks

If the parameter `_Timeout` has a value other than the constant `COOPERATIVE_TIMEOUT_INFINITE`, the exception [operation\\_timed\\_out](#) is thrown if the specified amount of time expires before a message is received. If you want a zero length timeout, you should use the [try\\_receive](#) function, as opposed to calling `receive` with a timeout of `0` (zero), as it is more efficient and does not throw exceptions on timeouts.

For more information, see [Message Passing Functions](#).

## run\_with\_cancellation\_token

Executes a function object immediately and synchronously in the context of a given cancellation token.

```
template<typename _Function>
void run_with_cancellation_token(
    const _Function& _Func,
    cancellation_token _Ct);
```

### Parameters

*\_Function*

The type of the function object that will be invoked.

*\_Func*

The function object which will be executed. This object must support the function call operator with a signature of void(void).

*\_Ct*

The cancellation token which will control implicit cancellation of the function object. Use

`cancellation_token::none()` if you want the function execute without any possibility of implicit cancellation from a parent task group being canceled.

### Remarks

Any interruption points in the function object will be triggered when the `cancellation_token` is canceled. The explicit token `_ct` will isolate this `_Func` from parent cancellation if the parent has a different token or no token.

## send

A synchronous send operation, which waits until the target either accepts or declines the message.

```
template <class T>
bool send(_Inout_ ITarget<T>* _Trg, const T& _Data);

template <class T>
bool send(ITarget<T>& _Trg, const T& _Data);
```

### Parameters

*T*

The payload type.

*\_Trg*

A pointer or reference to the target to which data is sent.

*\_Data*

A reference to the data to be sent.

### Return Value

**true** if the message was accepted, **false** otherwise.

### Remarks

For more information, see [Message Passing Functions](#).

## set\_ambient\_scheduler

```
inline void set_ambient_scheduler(std::shared_ptr<Concurrency::scheduler_interface> _Scheduler);
```

### Parameters

*\_Scheduler*

The ambient scheduler to set.

## set\_task\_execution\_resources

Restricts the execution resources used by the Concurrency Runtime internal worker threads to the affinity set specified.

It is valid to call this method only before the Resource Manager has been created, or between two Resource Manager lifetimes. It can be invoked multiple times as long as the Resource Manager does not exist at the time of invocation. After an affinity limit has been set, it remains in effect until the next valid call to the `set_task_execution_resources` method.

The affinity mask provided need not be a subset of the process affinity mask. The process affinity will be updated if necessary.

```
void __cdecl set_task_execution_resources(  
    DWORD_PTR _ProcessAffinityMask);  
  
void __cdecl set_task_execution_resources(  
    unsigned short count,  
    PGROUP_AFFINITY _PGroupAffinity);
```

### Parameters

*\_ProcessAffinityMask*

The affinity mask that the Concurrency Runtime worker threads are to be restricted to. Use this method on a system with greater than 64 hardware threads only if you want to limit the Concurrency Runtime to a subset of the current processor group. In general, you should use the version of the method that accepts an array of group affinities as a parameter, to restrict affinity on machines with greater than 64 hardware threads.

*count*

The number of `GROUP_AFFINITY` entries in the array specified by the parameter `_PGroupAffinity`.

*\_PGroupAffinity*

An array of `GROUP_AFFINITY` entries.

### Remarks

The method will throw an [invalid\\_operation](#) exception if a Resource Manager is present at the time it is invoked, and an [invalid\\_argument](#) exception if the affinity specified results in an empty set of resources.

The version of the method that takes an array of group affinities as a parameter should only be used on operating systems with version Windows 7 or higher. Otherwise, an [invalid\\_operation](#) exception is thrown.

Programmatically modifying the process affinity after this method has been invoked will not cause the Resource Manager to re-evaluate the affinity it is restricted to. Therefore, all changes to process affinity should be made before calling this method.

## swap

Exchanges the elements of two `concurrent_vector` objects.

```
template<typename T, class _Ax>  
inline void swap(  
    concurrent_vector<T, _Ax>& _A,  
    concurrent_vector<T, _Ax>& _B);
```

## Parameters

*T*

The data type of the elements stored in the concurrent vectors.

*\_Ax*

The allocator type of the concurrent vectors.

*\_A*

The concurrent vector whose elements are to be exchanged with those of the concurrent vector `_B`.

*\_B*

The concurrent vector providing the elements to be swapped, or the vector whose elements are to be exchanged with those of the concurrent vector `_A`.

## Remarks

The template function is an algorithm specialized on the container class `concurrent_vector` to execute the member function `_A.concurrent_vector::swap(_B)`. These are instances of the partial ordering of function templates by the compiler. When template functions are overloaded in such a way that the match of the template with the function call is not unique, then the compiler will select the most specialized version of the template function. The general version of the template function, `template <class T> void swap(T&, T&)`, in the algorithm class works by assignment and is a slow operation. The specialized version in each container is much faster as it can work with the internal representation of the container class.

This method is not concurrency-safe. You must ensure that no other threads are performing operations on either of the concurrent vectors when you call this method.

## task\_from\_exception

```
template<typename _TaskType, typename _ExType>
task<_TaskType> task_from_exception(
    _ExType _Exception,
    const task_options& _TaskOptions = task_options());
```

## Parameters

*\_TaskType*

*\_ExType*

*\_Exception*

*\_TaskOptions*

## Return Value

## task\_from\_result

```
template<typename T>
task<T> task_from_result(
    T _Param,
    const task_options& _TaskOptions = task_options());

inline task<bool> task_from_result(ool _Param);

inline task<void> task_from_result(
    const task_options& _TaskOptions = task_options());
```

## Parameters

*T*

*\_Param*

*\_TaskOptions*

## Return Value

# Trace\_agents\_register\_name

Associates the given name to the message block or agent in the ETW trace.

```
template <class T>
void Trace_agents_register_name(
    _Inout_ T* _PObject,
    _In_z_ const wchar_t* _Name);
```

## Parameters

*T*

The type of the object. This is typically a message block or an agent.

*\_PObject*

A pointer to the message block or agent that is being named in the trace.

*\_Name*

The name for the given object.

# try\_receive

A general try-receive implementation, allowing a context to look for data from exactly one source and filter the values that are accepted. If the data is not ready, the method will return **false**.

```
template <class T>
bool try_receive(_Inout_ ISource<T>* _Src, T& _value);

template <class T>
bool try_receive(
    _Inout_ ISource<T>* _Src,
    T& _value,
    typename ITarget<T>::filter_method const& _Filter_proc);

template <class T>
bool try_receive(ISource<T>& _Src, T& _value);

template <class T>
bool try_receive(
    ISource<T>& _Src,
    T& _value,
    typename ITarget<T>::filter_method const& _Filter_proc);
```

## Parameters

*T*

The payload type

*\_Src*

A pointer or reference to the source from which data is expected.

*\_value*

A reference to a location where the result will be placed.

*\_Filter\_proc*

A filter function which determines whether messages should be accepted.

### Return Value

A `bool` value indicating whether or not a payload was placed in `_value`.

### Remarks

For more information, see [Message Passing Functions](#).

## wait

Pauses the current context for a specified amount of time.

```
void __cdecl wait(unsigned int _Milliseconds);
```

### Parameters

*\_Milliseconds*

The number of milliseconds the current context should be paused for. If the `_Milliseconds` parameter is set to the value `0`, the current context should yield execution to other runnable contexts before continuing.

### Remarks

If this method is called on a Concurrency Runtime scheduler context, the scheduler will find a different context to run on the underlying resource. Because the scheduler is cooperative in nature, this context cannot resume exactly after the number of milliseconds specified. If the scheduler is busy executing other tasks that do not cooperatively yield to the scheduler, the wait period could be indefinite.

## when\_all

Creates a task that will complete successfully when all of the tasks supplied as arguments complete successfully.

```
template <typename _Iterator>
auto when_all(
    _Iterator _Begin,
    _Iterator _End,
    const task_options& _TaskOptions = task_options()) ->
decltype (details::_WhenAllImpl<typename std::iterator_traits<_Iterator>::value_type::result_type,
    _Iterator>::_Perform(_TaskOptions, _Begin, _End));
```

### Parameters

*\_Iterator*

The type of the input iterator.

*\_Begin*

The position of the first element in the range of elements to be combined into the resulting task.

*\_End*

The position of the first element beyond the range of elements to be combined into the resulting task.

*\_TaskOptions*

The `task_options` object.

### Return Value

A task that completes successfully when all of the input tasks have completed successfully. If the input tasks are of

type `T`, the output of this function will be a `task<std::vector<T>>`. If the input tasks are of type `void` the output task will also be a `task<void>`.

## Remarks

`when_all` is a non-blocking function that produces a `task` as its result. Unlike `task::wait`, it is safe to call this function in a UWP app on the ASTA (Application STA) thread.

If one of the tasks is canceled or throws an exception, the returned task will complete early, in the canceled state, and the exception, if one is encountered, will be thrown if you call `task::get` or `task::wait` on that task.

For more information, see [Task Parallelism](#).

## when\_any

Creates a task that will complete successfully when any of the tasks supplied as arguments completes successfully.

```
template<typename _Iterator>
auto when_any(
    _Iterator _Begin,
    _Iterator _End,
    const task_options& _TaskOptions = task_options())
-> decltype (
    details::_WhenAnyImpl<
        typename std::iterator_traits<_Iterator>::value_type::result_type,
        _Iterator>::_Perform(_TaskOptions, _Begin, _End));

template<typename _Iterator>
auto when_any(
    _Iterator _Begin,
    _Iterator _End,
    cancellation_token _CancellationToken)
-> decltype (
    details::_WhenAnyImpl<
        typename std::iterator_traits<_Iterator>::value_type::result_type,
        _Iterator>::_Perform(_CancellationToken.GetImplValue(), _Begin, _End));
```

## Parameters

*\_Iterator*

The type of the input iterator.

*\_Begin*

The position of the first element in the range of elements to be combined into the resulting task.

*\_End*

The position of the first element beyond the range of elements to be combined into the resulting task.

*\_TaskOptions*

*\_CancellationToken*

The cancellation token which controls cancellation of the returned task. If you do not provide a cancellation token, the resulting task will receive the cancellation token of the task that causes it to complete.

## Return Value

A task that completes successfully when any one of the input tasks has completed successfully. If the input tasks are of type `T`, the output of this function will be a `task<std::pair<T, size_t>>>`, where the first element of the pair is the result of the completing task, and the second element is the index of the task that finished. If the input tasks are of type `void` the output is a `task<size_t>`, where the result is the index of the completing task.

## Remarks

`when_any` is a non-blocking function that produces a `task` as its result. Unlike `task::wait`, it is safe to call this function in a UWP app on the ASTA (Application STA) thread.

For more information, see [Task Parallelism](#).

## See also

[concurrency Namespace](#)

# concurrency namespace Operators

3/4/2019 • 6 minutes to read • [Edit Online](#)

<code>operator!=</code>	<code>operator&amp;&amp;</code>	<code>operator&gt;</code>
<code>operator&gt;=</code>	<code>operator&lt;</code>	<code>operator&lt;=</code>
<code>operator==</code>	<code>operator  </code>	

## `operator||` Operator

Creates a task that will complete successfully when either of the tasks supplied as arguments completes successfully.

```
template<typename ReturnType>
task<ReturnType> operator||(
    const task<ReturnType>& lhs,
    const task<ReturnType>& rhs);

template<typename ReturnType>
task<std::vector<ReturnType>> operator||(
    const task<std::vector<ReturnType>>& lhs,
    const task<ReturnType>& rhs);

template<typename ReturnType>
task<std::vector<ReturnType>> operator||(
    const task<ReturnType>& lhs,
    const task<std::vector<ReturnType>>& rhs);

inline task<void> operator||(
    const task<void>& lhs,
    const task<void>& rhs);
```

### Parameters

*ReturnType*

The type of the returned task.

*lhs*

The first task to combine into the resulting task.

*rhs*

The second task to combine into the resulting task.

### Return Value

A task that completes successfully when either of the input tasks has completed successfully. If the input tasks are of type `T`, the output of this function will be a `task<std::vector<T>`. If the input tasks are of type `void` the output task will also be a `task<void>`.

### Remarks

If both of the tasks are canceled or throw exceptions, the returned task will complete in the canceled state, and one of the exceptions, if any are encountered, will be thrown when you call `get()` or `wait()` on that task.

# operator&& Operator

Creates a task that will complete successfully when both of the tasks supplied as arguments complete successfully.

```
template<typename ReturnType>
task<std::vector<ReturnType>> operator&&(
    const task<ReturnType>& lhs,
    const task<ReturnType>& rhs);

template<typename ReturnType>
task<std::vector<ReturnType>> operator&&(
    const task<std::vector<ReturnType>>& lhs,
    const task<ReturnType>& rhs);

template<typename ReturnType>
task<std::vector<ReturnType>> operator&&(
    const task<ReturnType>& lhs,
    const task<std::vector<ReturnType>>& rhs);

template<typename ReturnType>
task<std::vector<ReturnType>> operator&&(
    const task<std::vector<ReturnType>>& lhs,
    const task<std::vector<ReturnType>>& rhs);

inline task<void> operator&&(
    const task<void>& lhs,
    const task<void>& rhs);
```

## Parameters

*ReturnType*

The type of the returned task.

*lhs*

The first task to combine into the resulting task.

*rhs*

The second task to combine into the resulting task.

## Return Value

A task that completes successfully when both of the input tasks have completed successfully. If the input tasks are of type `T`, the output of this function will be a `task<std::vector<T>>`. If the input tasks are of type `void` the output task will also be a `task<void>`.

## Remarks

If one of the tasks is canceled or throws an exception, the returned task will complete early, in the canceled state, and the exception, if one is encountered, will be thrown if you call `get()` or `wait()` on that task.

# operator== Operator

Tests if the `concurrent_vector` object on the left side of the operator is equal to the `concurrent_vector` object on the right side.

```
template<typename T, class A1, class A2>
inline bool operator==(
    const concurrent_vector<T, A1>& _A,
    const concurrent_vector<T, A2>& _B);
```

## Parameters

*T*

The data type of the elements stored in the concurrent vectors.

*A1*

The allocator type of the first `concurrent_vector` object.

*A2*

The allocator type of the second `concurrent_vector` object.

*\_A*

An object of type `concurrent_vector`.

*\_B*

An object of type `concurrent_vector`.

### Return Value

**true** if the concurrent vector on the left side of the operator is equal to the concurrent vector on the right side of the operator; otherwise **false**.

### Remarks

Two concurrent vectors are equal if they have the same number of elements and their respective elements have the same values. Otherwise, they are unequal.

This method is not concurrency-safe with respect to other methods that could modify either of the concurrent vectors `_A` or `_B`.

## operator!= Operator

Tests if the `concurrent_vector` object on the left side of the operator is not equal to the `concurrent_vector` object on the right side.

```
template<typename T, class A1, class A2>
inline bool operator!= (
    const concurrent_vector<T, A1>& _A,
    const concurrent_vector<T, A2>& _B);
```

### Parameters

*T*

The data type of the elements stored in the concurrent vectors.

*A1*

The allocator type of the first `concurrent_vector` object.

*A2*

The allocator type of the second `concurrent_vector` object.

*\_A*

An object of type `concurrent_vector`.

*\_B*

An object of type `concurrent_vector`.

### Return Value

**true** if the concurrent vectors are not equal; **false** if the concurrent vectors are equal.

### Remarks

Two concurrent vectors are equal if they have the same number of elements and their respective elements have the

same values. Otherwise, they are unequal.

This method is not concurrency-safe with respect to other methods that could modify either of the concurrent vectors `_A` or `_B`.

## operator< Operator

Tests if the `concurrent_vector` object on the left side of the operator is less than the `concurrent_vector` object on the right side.

```
template<typename T, class A1, class A2>
inline bool operator<(
    const concurrent_vector<T, A1>& _A,
    const concurrent_vector<T, A2>& _B);
```

### Parameters

*T*

The data type of the elements stored in the concurrent vectors.

*A1*

The allocator type of the first `concurrent_vector` object.

*A2*

The allocator type of the second `concurrent_vector` object.

*\_A*

An object of type `concurrent_vector`.

*\_B*

An object of type `concurrent_vector`.

### Return Value

**true** if the concurrent vector on the left side of the operator is less than the concurrent vector on the right side of the operator; otherwise **false**.

### Remarks

The behavior of this operator is identical to the equivalent operator for the `vector` class in the `std` namespace.

This method is not concurrency-safe with respect to other methods that could modify either of the concurrent vectors `_A` or `_B`.

## operator<= Operator

Tests if the `concurrent_vector` object on the left side of the operator is less than or equal to the `concurrent_vector` object on the right side.

```
template<typename T, class A1, class A2>
inline bool operator<= (
    const concurrent_vector<T, A1>& _A,
    const concurrent_vector<T, A2>& _B);
```

### Parameters

*T*

The data type of the elements stored in the concurrent vectors.

*A1*

The allocator type of the first `concurrent_vector` object.

*A2*

The allocator type of the second `concurrent_vector` object.

*\_A*

An object of type `concurrent_vector`.

*\_B*

An object of type `concurrent_vector`.

### Return Value

**true** if the concurrent vector on the left side of the operator is less than or equal to the concurrent vector on the right side of the operator; otherwise **false**.

### Remarks

The behavior of this operator is identical to the equivalent operator for the `vector` class in the `std` namespace.

This method is not concurrency-safe with respect to other methods that could modify either of the concurrent vectors `_A` or `_B`.

## operator> Operator

Tests if the `concurrent_vector` object on the left side of the operator is greater than the `concurrent_vector` object on the right side.

```
template<typename T, class A1, class A2>
inline bool operator>(
    const concurrent_vector<T, A1>& _A,
    const concurrent_vector<T, A2>& _B);
```

### Parameters

*T*

The data type of the elements stored in the concurrent vectors.

*A1*

The allocator type of the first `concurrent_vector` object.

*A2*

The allocator type of the second `concurrent_vector` object.

*\_A*

An object of type `concurrent_vector`.

*\_B*

An object of type `concurrent_vector`.

### Return Value

**true** if the concurrent vector on the left side of the operator is greater than the concurrent vector on the right side of the operator; otherwise **false**.

### Remarks

The behavior of this operator is identical to the equivalent operator for the `vector` class in the `std` namespace.

This method is not concurrency-safe with respect to other methods that could modify either of the concurrent vectors `_A` or `_B`.

# operator>= Operator

Tests if the `concurrent_vector` object on the left side of the operator is greater than or equal to the `concurrent_vector` object on the right side.

```
template<typename T, class A1, class A2>
inline bool operator>= (
    const concurrent_vector<T, A1>& _A,
    const concurrent_vector<T, A2>& _B);
```

## Parameters

*T*

The data type of the elements stored in the concurrent vectors.

*A1*

The allocator type of the first `concurrent_vector` object.

*A2*

The allocator type of the second `concurrent_vector` object.

*\_A*

An object of type `concurrent_vector`.

*\_B*

An object of type `concurrent_vector`.

## Return Value

**true** if the concurrent vector on the left side of the operator is greater than or equal to the concurrent vector on the right side of the operator; otherwise **false**.

## Remarks

The behavior of this operator is identical to the equivalent operator for the `vector` class in the `std` namespace.

This method is not concurrency-safe with respect to other methods that could modify either of the concurrent vectors `_A` or `_B`.

## See also

[concurrency Namespace](#)

# concurrency namespace constants

3/4/2019 • 3 minutes to read • [Edit Online](#)

AgentEventGuid	CONCRT_RM_VERSION_1	COOPERATIVE_TIMEOUT_INFINITE
COOPERATIVE_WAIT_TIMEOUT	ChoreEventGuid	ConcRTEventGuid
ConcRT_ProviderGuid	ContextEventGuid	INHERIT_THREAD_PRIORITY
LockEventGuid	MaxExecutionResources	PPLParallelForEventGuid
PPLParallelForeachEventGuid	PPLParallelInvokeEventGuid	ResourceManagerEventGuid
ScheduleGroupEventGuid	SchedulerEventGuid	VirtualProcessorEventGuid

## AgentEventGuid

A category GUID ({B9B5B78C-0713-4898-A21A-C67949DCED07}) describing ETW events fired by the Agents library in the Concurrency Runtime.

```
const __declspec(selectany) GUID AgentEventGuid = {0xb9b5b78c, 0x713, 0x4898, { 0xa2, 0x1a, 0xc6, 0x79, 0x49, 0xdc, 0xed, 0x7 } };
```

## ChoreEventGuid

A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to chores or tasks.

```
const __declspec(selectany) GUID ChoreEventGuid = { 0x7E854EC7, 0xCDC4, 0x405a, { 0xB5, 0xB2, 0xAA, 0xF7, 0xC9, 0xE7, 0xD4, 0x0C } };
```

### Remarks

This category of events is not currently fired by the Concurrency Runtime.

## ConcRT\_ProviderGuid

The ETW provider GUID for the Concurrency Runtime.

```
const __declspec(selectany) GUID ConcRT_ProviderGuid = { 0xF7B697A3, 0x4DB5, 0x4d3b, { 0xBE, 0x71, 0xC4, 0xD2, 0x84, 0xE6, 0x59, 0x2F } };
```

## CONCRT\_RM\_VERSION\_1

Indicates support of the Resource Manager interface defined in Visual Studio 2010.

```
const unsigned int CONCRT_RM_VERSION_1 = 0x00010000;
```

## ConcRTEventGuid

A category GUID describing ETW events fired by the Concurrency Runtime that are not more specifically described by another category.

```
const __declspec(selectany) GUID ConcRTEventGuid =  
    { 0x72B14A7D, 0x704C, 0x423e, { 0x92, 0xF8, 0x7E, 0x6D, 0x64, 0xBC, 0xB9, 0x2A } };
```

### Remarks

This category of events is not currently fired by the Concurrency Runtime.

## COOPERATIVE\_TIMEOUT\_INFINITE

Value indicating that a wait should never time out.

```
const unsigned int COOPERATIVE_TIMEOUT_INFINITE = (unsigned int)-1;
```

## COOPERATIVE\_WAIT\_TIMEOUT

Value indicating that a wait timed out.

```
const size_t COOPERATIVE_WAIT_TIMEOUT = SIZE_MAX;
```

## ContextEventGuid

A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to contexts.

```
const __declspec(selectany) GUID ContextEventGuid =  
    { 0x5727A00F, 0x50BE, 0x4519, { 0x82, 0x56, 0xF7, 0x69, 0x98, 0x71, 0xFE, 0xCB } };
```

## INHERIT\_THREAD\_PRIORITY

Special value for the policy key `ContextPriority` indicating that the thread priority of all contexts in the scheduler should be the same as that of the thread which created the scheduler.

```
const unsigned int INHERIT_THREAD_PRIORITY = 0x0000F000;
```

## LockEventGuid

A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to locks.

```
const __declspec(selectany) GUID LockEventGuid =  
    { 0x79A60DC6, 0x5FC8, 0x4952, { 0xA4, 0x1C, 0x11, 0x63, 0xAE, 0xEC, 0x5E, 0xB8 } };
```

### Remarks

This category of events is not currently fired by the Concurrency Runtime.

## MaxExecutionResources

Special value for the policy keys `MinConcurrency` and `MaxConcurrency`. Defaults to the number of hardware threads on the machine in the absence of other constraints.

```
const unsigned int MaxExecutionResources = 0xFFFFFFFF;
```

## PPLParallelForEventGuid

A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to usage of the `parallel_for` function.

```
const __declspec(selectany) GUID PPLParallelForEventGuid =  
    { 0x31c8da6b, 0x6165, 0x4042, { 0x8b, 0x92, 0x94, 0x9e, 0x31, 0x5f, 0x4d, 0x84 } };
```

## PPLParallelForeachEventGuid

A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to usage of the `parallel_for_each` function.

```
const __declspec(selectany) GUID PPLParallelForeachEventGuid =  
    { 0x5cb7d785, 0x9d66, 0x465d, { 0xba, 0xe1, 0x46, 0x11, 0x6, 0x1b, 0x54, 0x34 } };
```

## PPLParallelInvokeEventGuid

A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to usage of the `parallel_invoke` function.

```
const __declspec(selectany) GUID PPLParallelInvokeEventGuid =  
    { 0xd1b5b133, 0xec3d, 0x49f4, { 0x98, 0xa3, 0x46, 0x4d, 0x1a, 0x9e, 0x46, 0x82 } };
```

## ResourceManagerEventGuid

A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to the resource manager.

```
const __declspec(selectany) GUID ResourceManagerEventGuid =  
    { 0x2718D25B, 0x5BF5, 0x4479, { 0x8E, 0x88, 0xBA, 0xBC, 0x64, 0xBD, 0xBF, 0xCA } };
```

### Remarks

This category of events is not currently fired by the Concurrency Runtime.

## ScheduleGroupEventGuid

A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to schedule groups.

```
const __declspec(selectany) GUID ScheduleGroupEventGuid =
    { 0xE8A3BF1F, 0xA86B, 0x4390, { 0x9C, 0x60, 0x53, 0x90, 0xB9, 0x69, 0xD2, 0x2C } };
```

### Remarks

This category of events is not currently fired by the Concurrency Runtime.

## SchedulerEventGuid

A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to scheduler activity.

```
const __declspec(selectany) GUID SchedulerEventGuid =
    { 0xE2091F8A, 0x1E0A, 0x4731, { 0x84, 0xA2, 0x0D, 0xD5, 0x7C, 0x8A, 0x52, 0x61 } };
```

## VirtualProcessorEventGuid

A category GUID describing ETW events fired by the Concurrency Runtime that are directly related to virtual processors.

```
const __declspec(selectany) GUID VirtualProcessorEventGuid =
    { 0x2f27805f, 0x1676, 0x4ecc, { 0x96, 0xfa, 0x7e, 0xb0, 0x9d, 0x44, 0x30, 0x2f } };
```

## See also

[concurrency Namespace](#)

# concurrency namespace enums

3/4/2019 • 9 minutes to read • [Edit Online](#)

Agents_EventType	ConcRT_EventType	Concrt_TraceFlags
CriticalRegionType	DynamicProgressFeedbackType	PolicyElementKey
SchedulerType	SchedulingProtocolType	SwitchingProxyState
WinRTInitializationType	agent_status	join_type
message_status	task_group_status	

## agent\_status Enumeration

The valid states for an `agent`.

```
enum agent_status;
```

### Values

NAME	DESCRIPTION
<code>agent_canceled</code>	The <code>agent</code> was canceled.
<code>agent_created</code>	The <code>agent</code> has been created but not started.
<code>agent_done</code>	The <code>agent</code> finished without being canceled.
<code>agent_runnable</code>	The <code>agent</code> has been started, but not entered its <code>run</code> method.
<code>agent_started</code>	The <code>agent</code> has started.

### Remarks

For more information, see [Asynchronous Agents](#).

### Requirements

**Header:** `concrth`

## Agents\_EventType Enumeration

The types of events that can be traced using the tracing functionality offered by the Agents Library

```
enum Agents_EventType;
```

## Values

NAME	DESCRIPTION
<code>AGENTS_EVENT_CREATE</code>	An event type that represents the creation of an object
<code>AGENTS_EVENT_DESTROY</code>	An event type that represents the deletion of an object
<code>AGENTS_EVENT_END</code>	An event type that represents the conclusion of some processing
<code>AGENTS_EVENT_LINK</code>	An event type that represents the linking of message blocks
<code>AGENTS_EVENT_NAME</code>	An event type that represents the name for an object
<code>AGENTS_EVENT_SCHEDULE</code>	An event type that represents the scheduling of a process
<code>AGENTS_EVENT_START</code>	An event type that represents the initiation of some processing
<code>AGENTS_EVENT_UNLINK</code>	An event type that represents the unlinking of message blocks

## Requirements

**Header:** `concr.h`

## ConcRT\_EventType Enumeration

The types of events that can be traced using the tracing functionality offered by the Concurrency Runtime.

```
enum ConcRT_EventType;
```

## Values

NAME	DESCRIPTION
<code>CONCRT_EVENT_ATTACH</code>	An event type that represents the act of a attaching to a scheduler.
<code>CONCRT_EVENT_BLOCK</code>	An event type that represents the act of a context blocking.
<code>CONCRT_EVENT_DETACH</code>	An event type that represents the act of a detaching from a scheduler.
<code>CONCRT_EVENT_END</code>	An event type that marks the beginning of a start/end event pair.
<code>CONCRT_EVENT_GENERIC</code>	An event type used for miscellaneous events.
<code>CONCRT_EVENT_IDLE</code>	An event type that represents the act of a context becoming idle.

NAME	DESCRIPTION
<code>CONCRT_EVENT_START</code>	An event type that marks the beginning of a start/end event pair.
<code>CONCRT_EVENT_UNBLOCK</code>	An event type that represents the act of unblocking a context.
<code>CONCRT_EVENT_YIELD</code>	An event type that represents the act of a context yielding.

### Requirements

**Header:** `concr.h` **Namespace:** `concurrency`

## Concrt\_TraceFlags Enumeration

Trace flags for the event types

```
enum Concrt_TraceFlags;
```

### Values

NAME	DESCRIPTION
<code>AgentEventFlag</code>	
<code>AllEventsFlag</code>	
<code>ContextEventFlag</code>	
<code>PPLEventFlag</code>	
<code>ResourceManagerEventFlag</code>	
<code>SchedulerEventFlag</code>	
<code>VirtualProcessorEventFlag</code>	

### Requirements

**Header:** `concr.h`

## CriticalRegionType Enumeration

The type of critical region a context is inside.

```
enum CriticalRegionType;
```

### Values

NAME	DESCRIPTION
------	-------------

NAME	DESCRIPTION
<code>InsideCriticalRegion</code>	Indicates that the context is inside a critical region. When inside a critical region, asynchronous suspensions are hidden from the scheduler. Should such a suspension happen, the Resource Manager will wait for the thread to become runnable and simply resume it instead of invoking the scheduler again. Any locks taken inside such a region must be taken with extreme care.
<code>InsideHyperCriticalRegion</code>	Indicates that the context is inside a hyper-critical region. When inside a hyper-critical region, both synchronous and asynchronous suspensions are hidden from the scheduler. Should such a suspension or blocking happen, the resource manager will wait for the thread to become runnable and simply resume it instead of invoking the scheduler again. Locks taken inside such a region must never be shared with code running outside such a region. Doing so will cause unpredictable deadlock.
<code>OutsideCriticalRegion</code>	Indicates that the context is outside any critical region.

## Requirements

**Header:** `concrtrm.h`

## DynamicProgressFeedbackType Enumeration

Used by the `DynamicProgressFeedback` policy to describe whether resources for the scheduler will be rebalanced according to statistical information gathered from the scheduler or only based on virtual processors going in and out of the idle state through calls to the `Activate` and `Deactivate` methods on the `IVirtualProcessorRoot` interface. For more information on available scheduler policies, see [PolicyElementKey](#).

```
enum DynamicProgressFeedbackType;
```

## Values

NAME	DESCRIPTION
<code>ProgressFeedbackDisabled</code>	The scheduler does not gather progress information. Rebalancing is done based solely on the subscription level of the underlying hardware thread. For more information on subscription levels, see <a href="#">IExecutionResource::CurrentSubscriptionLevel</a> .  This value is reserved for use by the runtime.
<code>ProgressFeedbackEnabled</code>	The scheduler gathers progress information and passes it to the resource manager. The resource manager will utilize this statistical information to rebalance resources on behalf of the scheduler in addition to the subscription level of the underlying hardware thread. For more information on subscription levels, see <a href="#">IExecutionResource::CurrentSubscriptionLevel</a> .

## join\_type Enumeration

The type of a `join` messaging block.

```
enum join_type;
```

### Values

NAME	DESCRIPTION
<code>greedy</code>	Greedy <code>join</code> messaging blocks immediately accept a message upon propagation. This is more efficient, but has the possibility for live-lock, depending on the network configuration.
<code>non_greedy</code>	Non-greedy <code>join</code> messaging blocks postpone messages and try and consume them after all have arrived. These are guaranteed to work, but slower.

### Requirements

**Header:** agents.h

## message\_status Enumeration

The valid responses for an offer of a `message` object to a block.

```
enum message_status;
```

### Values

NAME	DESCRIPTION
<code>accepted</code>	The target accepted the message.
<code>declined</code>	The target did not accept the message.
<code>missed</code>	The target tried to accept the message, but it was no longer available.
<code>postponed</code>	The target postponed the message.

### Requirements

**Header:** agents.h

## PolicyElementKey Enumeration

Policy keys describing aspects of scheduler behavior. Each policy element is described by a key-value pair. For more information about scheduler policies and their impact on schedulers, see [Task Scheduler](#).

```
enum PolicyElementKey;
```

### Values

NAME	DESCRIPTION
<p><code>ContextPriority</code></p>	<p>The operating system thread priority of each context in the scheduler. If this key is set to the value <code>INHERIT_THREAD_PRIORITY</code> the contexts in the scheduler will inherit the priority of the thread that created the scheduler.</p> <p>Valid values : Any of the valid values for the Windows <code>SetThreadPriority</code> function and the special value <code>INHERIT_THREAD_PRIORITY</code></p> <p>Default value : <code>THREAD_PRIORITY_NORMAL</code></p>
<p><code>ContextStackSize</code></p>	<p>The reserved stack size of each context in the scheduler in kilobytes.</p> <p>Valid values : Positive integers</p> <p>Default value : <code>0</code>, indicating that the process' default value for stack size be used.</p>
<p><code>DynamicProgressFeedback</code></p>	<p>Determines whether the resources for the scheduler will be rebalanced according to statistical information gathered from the scheduler or only based on the subscription level of underlying hardware threads. For more information, see <a href="#">DynamicProgressFeedbackType</a>.</p> <p>Valid values : A member of the <code>DynamicProgressFeedbackType</code> enumeration, either <code>ProgressFeedbackEnabled</code> OR <code>ProgressFeedbackDisabled</code></p> <p>Default value : <code>ProgressFeedbackEnabled</code></p>
<p><code>LocalContextCacheSize</code></p>	<p>When the <code>SchedulingProtocol</code> policy key is set to the value <code>EnhanceScheduleGroupLocality</code>, this specifies the maximum number of runnable contexts allowed to be cached in per virtual processor local queues. Such contexts will typically run in last-in-first-out (LIFO) order on the virtual processor that caused them to become runnable. Note that this policy key has no meaning when the <code>SchedulingProtocol</code> key is set to the value <code>EnhanceForwardProgress</code>.</p> <p>Valid values : Non-negative integers</p> <p>Default value : <code>8</code></p>

NAME	DESCRIPTION
<p><code>MaxConcurrency</code></p>	<p>The maximum concurrency level desired by the scheduler. The resource manager will try to initially allocate this many virtual processors. The special value <code>MaxExecutionResources</code> indicates that the desired concurrency level is same as the number of hardware threads on the machine. If the value specified for <code>MinConcurrency</code> is greater than the number of hardware threads on the machine and <code>MaxConcurrency</code> is specified as <code>MaxExecutionResources</code>, the value for <code>MaxConcurrency</code> is raised to match what is set for <code>MinConcurrency</code>.</p> <p>Valid values : Positive integers and the special value <code>MaxExecutionResources</code></p> <p>Default value : <code>MaxExecutionResources</code></p>
<p><code>MaxPolicyElementKey</code></p>	<p>The maximum policy element key. Not a valid element key.</p>
<p><code>MinConcurrency</code></p>	<p>The minimum concurrency level that must be provided to the scheduler by the resource manager. The number of virtual processors assigned to a scheduler will never go below the minimum. The special value <code>MaxExecutionResources</code> indicates that the minimum concurrency level is same as the number of hardware threads on the machine. If the value specified for <code>MaxConcurrency</code> is less than the number of hardware threads on the machine and <code>MinConcurrency</code> is specified as <code>MaxExecutionResources</code>, the value for <code>MinConcurrency</code> is lowered to match what is set for <code>MaxConcurrency</code>.</p> <p>Valid values : Non-negative integers and the special value <code>MaxExecutionResources</code>. Note that for scheduler policies used for the construction of Concurrency Runtime schedulers, the value <code>0</code> is invalid.</p> <p>Default value : <code>1</code></p>
<p><code>SchedulerKind</code></p>	<p>The type of threads that the scheduler will utilize for underlying execution contexts. For more information, see <a href="#">SchedulerType</a>.</p> <p>Valid values : A member of the <code>SchedulerType</code> enumeration, for example, <code>ThreadScheduler</code></p> <p>Default value : <code>ThreadScheduler</code>. This translates to Win32 threads on all operating systems.</p>
<p><code>SchedulingProtocol</code></p>	<p>Describes which scheduling algorithm will be used by the scheduler. For more information, see <a href="#">SchedulingProtocolType</a>.</p> <p>Valid values : A member of the <code>SchedulingProtocolType</code> enumeration, either <code>EnhanceScheduleGroupLocality</code> or <code>EnhanceForwardProgress</code></p> <p>Default value : <code>EnhanceScheduleGroupLocality</code></p>

NAME	DESCRIPTION
<code>TargetOversubscriptionFactor</code>	<p>Tentative number of virtual processors per hardware thread. The target oversubscription factor can be increased by the Resource Manager, if necessary, to satisfy <code>MaxConcurrency</code> with the hardware threads on the machine.</p> <p>Valid values : Positive integers</p> <p>Default value : <code>1</code></p>
<code>WinRTInitialization</code>	

## Requirements

**Header:** `concrth`

## SchedulerType Enumeration

Used by the `SchedulerKind` policy to describe the type of threads that the scheduler should utilize for underlying execution contexts. For more information on available scheduler policies, see [PolicyElementKey](#).

```
enum SchedulerType;
```

## Values

NAME	DESCRIPTION
<code>ThreadScheduler</code>	Indicates an explicit request of regular Win32 threads.
<code>UmsThreadDefault</code>	<p>User-mode schedulable (UMS) threads are not supported in the Concurrency Runtime in Visual Studio 2013. Using <code>UmsThreadDefault</code> as a value for the <code>SchedulerType</code> policy will not result in an error. However, a scheduler created with that policy will default to using Win32 threads.</p>

## Requirements

**Header:** `concrth`

## SchedulingProtocolType Enumeration

Used by the `SchedulingProtocol` policy to describe which scheduling algorithm will be utilized for the scheduler. For more information on available scheduler policies, see [PolicyElementKey](#).

```
enum SchedulingProtocolType;
```

## Values

NAME	DESCRIPTION
<code>EnhanceForwardProgress</code>	<p>The scheduler prefers to round-robin through schedule groups after executing each task. Unblocked contexts are typically scheduled in a first-in-first-out (FIFO) fashion. Virtual processors do not cache unblocked contexts.</p>

NAME	DESCRIPTION
<code>EnhanceScheduleGroupLocality</code>	The scheduler prefers to continue to work on tasks within the current schedule group before moving to another schedule group. Unblocked contexts are cached per virtual-processor and are typically scheduled in a last-in-first-out (LIFO) fashion by the virtual processor which unblocked them.

## Requirements

**Header:** `concr.h`

## SwitchingProxyState Enumeration

Used to denote the state a thread proxy is in, when it is executing a cooperative context switch to a different thread proxy.

```
enum SwitchingProxyState;
```

### Values

NAME	DESCRIPTION
<code>Blocking</code>	Indicates that the calling thread is cooperatively blocking and should be exclusively owned by the caller until subsequently running again and performing other action.
<code>Idle</code>	Indicates that the calling thread is no longer needed by the scheduler and is being returned to the Resource Manager. The context which was being dispatched is no longer able to be utilized by the Resource Manager.
<code>Nesting</code>	Indicates that the calling thread is nesting a child scheduler and is needed by the caller, in order to attach to a different scheduler.

### Remarks

A parameter of type `SwitchingProxyState` is passed in to the method `IThreadProxy::SwitchTo` to instruct the Resource Manager how to treat the thread proxy that is making the call.

For more information on how this type is used, see [IThreadProxy::SwitchTo](#).

## task\_group\_status Enumeration

Describes the execution status of a `task_group` or `structured_task_group` object. A value of this type is returned by numerous methods that wait on tasks scheduled to a task group to complete.

```
enum task_group_status;
```

### Values

NAME	DESCRIPTION
------	-------------

NAME	DESCRIPTION
<code>canceled</code>	The <code>task_group</code> or <code>structured_task_group</code> object was canceled. One or more tasks may not have executed.
<code>completed</code>	The tasks queued to the <code>task_group</code> or <code>structured_task_group</code> object completed successfully.
<code>not_complete</code>	The tasks queued to the <code>task_group</code> object have not completed. Note that this value is not presently returned by the Concurrency Runtime.

## Requirements

**Header:** `pplinterface.h`

## WinRTInitializationType Enumeration

Used by the `WinRTInitialization` policy to describe whether and how the Windows Runtime will be initialized on scheduler threads for an application which runs on operating systems with version Windows 8 or higher. For more information on available scheduler policies, see [PolicyElementKey](#).

```
enum WinRTInitializationType;
```

## Values

NAME	DESCRIPTION
<code>DoNotInitializeWinRT</code>	When the application is run on operating systems with version Windows 8 or higher, threads within the scheduler will not initialize the Windows Runtime .
<code>InitializeWinRTAsMTA</code>	When the application is run on operating systems with version Windows 8 or higher, each thread within the scheduler will initialize the Windows Runtime and declare that it is part of the multithreaded apartment.

## Requirements

**Header:** `concrct.h`

## See also

[concurrency Namespace](#)

# affinity\_partitioner Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `affinity_partitioner` class is similar to the `static_partitioner` class, but it improves cache affinity by its choice of mapping subranges to worker threads. It can improve performance significantly when a loop is re-executed over the same data set, and the data fits in cache. Note that the same `affinity_partitioner` object must be used with subsequent iterations of a parallel loop that is executed over a particular data set, to benefit from data locality.

## Syntax

```
class affinity_partitioner;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">affinity_partitioner</a>	Constructs an <code>affinity_partitioner</code> object.
<a href="#">~affinity_partitioner Destructor</a>	Destroys an <code>affinity_partitioner</code> object.

## Inheritance Hierarchy

```
affinity_partitioner
```

## Requirements

**Header:** ppl.h

**Namespace:** concurrency

### ~affinity\_partitioner

Destroys an `affinity_partitioner` object.

```
~affinity_partitioner();
```

### affinity\_partitioner

Constructs an `affinity_partitioner` object.

```
affinity_partitioner();
```

## See also



# agent Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

A class intended to be used as a base class for all independent agents. It is used to hide state from other agents and interact using message-passing.

## Syntax

```
class agent;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">agent</a>	Overloaded. Constructs an agent.
<a href="#">~agent Destructor</a>	Destroys the agent.

### Public Methods

NAME	DESCRIPTION
<a href="#">cancel</a>	Moves an agent from either the <code>agent_created</code> or <code>agent_runnable</code> states to the <code>agent_canceled</code> state.
<a href="#">start</a>	Moves an agent from the <code>agent_created</code> state to the <code>agent_runnable</code> state, and schedules it for execution.
<a href="#">status</a>	A synchronous source of status information from the agent.
<a href="#">status_port</a>	An asynchronous source of status information from the agent.
<a href="#">wait</a>	Waits for an agent to complete its task.
<a href="#">wait_for_all</a>	Waits for all of the specified agents to complete their tasks.
<a href="#">wait_for_one</a>	Waits for any one of the specified agents to complete its task.

### Protected Methods

NAME	DESCRIPTION
<a href="#">done</a>	Moves an agent into the <code>agent_done</code> state, indicating that the agent has completed.

NAME	DESCRIPTION
<code>run</code>	Represents the main task of an agent. <code>run</code> should be overridden in a derived class, and specifies what the agent should do after it has been started.

## Remarks

For more information, see [Asynchronous Agents](#).

## Inheritance Hierarchy

`agent`

## Requirements

**Header:** `agents.h`

**Namespace:** `concurrency`

## agent

Constructs an agent.

```
agent();

agent(Scheduler& _PScheduler);

agent(ScheduleGroup& _PGroup);
```

### Parameters

*\_PScheduler*

The `Scheduler` object within which the execution task of the agent is scheduled.

*\_PGroup*

The `ScheduleGroup` object within which the execution task of the agent is scheduled. The `Scheduler` object used is implied by the schedule group.

### Remarks

The runtime uses the default scheduler if you do not specify the `_PScheduler` or `_PGroup` parameters.

## ~agent

Destroys the agent.

```
virtual ~agent();
```

### Remarks

It is an error to destroy an agent that is not in a terminal state (either `agent_done` or `agent_cancelled`). This can be avoided by waiting for the agent to reach a terminal state in the destructor of a class that inherits from the `agent` class.

## cancel

Moves an agent from either the `agent_created` or `agent_runnable` states to the `agent_canceled` state.

```
bool cancel();
```

### Return Value

**true** if the agent was canceled, **false** otherwise. An agent cannot be canceled if it has already started running or has already completed.

## done

Moves an agent into the `agent_done` state, indicating that the agent has completed.

```
bool done();
```

### Return Value

**true** if the agent is moved to the `agent_done` state, **false** otherwise. An agent that has been canceled cannot be moved to the `agent_done` state.

### Remarks

This method should be called at the end of the `run` method, when you know the execution of your agent has completed.

## run

Represents the main task of an agent. `run` should be overridden in a derived class, and specifies what the agent should do after it has been started.

```
virtual void run() = 0;
```

### Remarks

The agent status is changed to `agent_started` right before this method is invoked. The method should invoke `done` on the agent with an appropriate status before returning, and may not throw any exceptions.

## start

Moves an agent from the `agent_created` state to the `agent_runnable` state, and schedules it for execution.

```
bool start();
```

### Return Value

**true** if the agent started correctly, **false** otherwise. An agent that has been canceled cannot be started.

## status

A synchronous source of status information from the agent.

```
agent_status status();
```

### Return Value

Returns the current state of the agent. Note that this returned state could change immediately after being returned.

## status\_port

An asynchronous source of status information from the agent.

```
ISource<agent_status>* status_port();
```

### Return Value

Returns a message source that can send messages about the current state of the agent.

## wait

Waits for an agent to complete its task.

```
static agent_status __cdecl wait(  
    _Inout_ agent* _PAgent,  
    unsigned int _Timeout = COOPERATIVE_TIMEOUT_INFINITE);
```

### Parameters

*\_PAgent*

A pointer to the agent to wait for.

*\_Timeout*

The maximum time for which to wait, in milliseconds.

### Return Value

The `agent_status` of the agent when the wait completes. This can either be `agent_canceled` or `agent_done`.

### Remarks

An agent task is completed when the agent enters the `agent_canceled` or `agent_done` states.

If the parameter `_Timeout` has a value other than the constant `COOPERATIVE_TIMEOUT_INFINITE`, the exception [operation\\_timed\\_out](#) is thrown if the specified amount of time expires before the agent has completed its task.

## wait\_for\_all

Waits for all of the specified agents to complete their tasks.

```
static void __cdecl wait_for_all(  
    size_t count,  
    _In_reads_(count) agent** _PAgents,  
    _Out_writes_opt_(count) agent_status* _PStatus = NULL,  
    unsigned int _Timeout = COOPERATIVE_TIMEOUT_INFINITE);
```

### Parameters

*count*

The number of agent pointers present in the array `_PAgents`.

*\_PAgents*

An array of pointers to the agents to wait for.

*\_PStatus*

A pointer to an array of agent statuses. Each status value will represent the status of the corresponding agent when the method returns.

*\_Timeout*

The maximum time for which to wait, in milliseconds.

### Remarks

An agent task is completed when the agent enters the `agent_canceled` or `agent_done` states.

If the parameter `_Timeout` has a value other than the constant `COOPERATIVE_TIMEOUT_INFINITE`, the exception `operation_timed_out` is thrown if the specified amount of time expires before the agent has completed its task.

## wait\_for\_one

Waits for any one of the specified agents to complete its task.

```
static void __cdecl wait_for_one(  
    size_t count,  
    _In_reads_(count) agent** _PAgents,  
    agent_status& _Status,  
    size_t& _Index,  
    unsigned int _Timeout = COOPERATIVE_TIMEOUT_INFINITE);
```

### Parameters

*count*

The number of agent pointers present in the array `_PAgents`.

*\_PAgents*

An array of pointers to the agents to wait for.

*\_Status*

A reference to a variable where the agent status will be placed.

*\_Index*

A reference to a variable where the agent index will be placed.

*\_Timeout*

The maximum time for which to wait, in milliseconds.

### Remarks

An agent task is completed when the agent enters the `agent_canceled` or `agent_done` states.

If the parameter `_Timeout` has a value other than the constant `COOPERATIVE_TIMEOUT_INFINITE`, the exception `operation_timed_out` is thrown if the specified amount of time expires before the agent has completed its task.

## See also

[concurrency Namespace](#)

# auto\_partitioner Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `auto_partitioner` class represents the default method `parallel_for`, `parallel_for_each` and `parallel_transform` use to partition the range they iterates over. This method of partitioning employs range stealing for load balancing as well as per-iterate cancellation.

## Syntax

```
class auto_partitioner;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">auto_partitioner</a>	Constructs a <code>auto_partitioner</code> object.
<a href="#">~auto_partitioner</a> Destructor	Destroys a <code>auto_partitioner</code> object.

## Inheritance Hierarchy

```
auto_partitioner
```

## Requirements

**Header:** `ppl.h`

**Namespace:** `concurrency`

### `~auto_partitioner`

Destroys a `auto_partitioner` object.

```
~auto_partitioner();
```

### `auto_partitioner`

Constructs a `auto_partitioner` object.

```
auto_partitioner();
```

## See also

[concurrency Namespace](#)

# bad\_target Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when a messaging block is given a pointer to a target which is invalid for the operation being performed.

## Syntax

```
class bad_target : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">bad_target</a>	Overloaded. Constructs a <code>bad_target</code> object.

## Remarks

This exception is typically thrown for reasons such as a target attempting to consume a message which is reserved for a different target or releasing a reservation that it does not hold.

## Inheritance Hierarchy

`exception`

`bad_target`

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## bad\_target

Constructs a `bad_target` object.

```
explicit _CRTIMP bad_target(_In_z_ const char* _Message) throw();  
  
bad_target() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

concurrency Namespace  
Asynchronous Message Blocks

# call Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

A `call` messaging block is a multi-source, ordered `target_block` that invokes a specified function when receiving a message.

## Syntax

```
template<class T, class _FunctorType = std::function<void(T const&)>>
class call : public target_block<multi_link_registry<ISource<T>>>;
```

### Parameters

*T*

The payload type of the messages propagated to this block.

*\_FunctorType*

The signature of functions that this block can accept.

## Members

### Public Constructors

NAME	DESCRIPTION
<code>call</code>	Overloaded. Constructs a <code>call</code> messaging block.
<code>~call</code> Destructor	Destroys the <code>call</code> messaging block.

### Protected Methods

NAME	DESCRIPTION
<code>process_input_messages</code>	Executes the <code>call</code> function on the input messages.
<code>process_message</code>	Processes a message that was accepted by this <code>call</code> messaging block.
<code>propagate_message</code>	Asynchronously passes a message from an <code>ISource</code> block to this <code>call</code> messaging block. It is invoked by the <code>propagate</code> method, when called by a source block.
<code>send_message</code>	Synchronously passes a message from an <code>ISource</code> block to this <code>call</code> messaging block. It is invoked by the <code>send</code> method, when called by a source block.
<code>supports_anonymous_source</code>	Overrides the <code>supports_anonymous_source</code> method to indicate that this block can accept messages offered to it by a source that is not linked. (Overrides <code>ITarget::supports_anonymous_source</code> .)

# Remarks

For more information, see [Asynchronous Message Blocks](#).

# Inheritance Hierarchy

[ITarget](#)

[target\\_block](#)

`call`

# Requirements

**Header:** agents.h

**Namespace:** concurrency

# call

Constructs a `call` messaging block.

```
call(
    _Call_method const& _Func);

call(
    _Call_method const& _Func,
    filter_method const& _Filter);

call(
    Scheduler& _PScheduler,
    _Call_method const& _Func);

call(
    Scheduler& _PScheduler,
    _Call_method const& _Func,
    filter_method const& _Filter);

call(
    ScheduleGroup& _PScheduleGroup,
    _Call_method const& _Func);

call(
    ScheduleGroup& _PScheduleGroup,
    _Call_method const& _Func,
    filter_method const& _Filter);
```

## Parameters

*\_Func*

A function that will be invoked for each accepted message.

*\_Filter*

A filter function which determines whether offered messages should be accepted.

*\_PScheduler*

The `Scheduler` object within which the propagation task for the `call` messaging block is scheduled.

*\_PScheduleGroup*

The `ScheduleGroup` object within which the propagation task for the `call` messaging block is scheduled. The `Scheduler` object used is implied by the schedule group.

## Remarks

The runtime uses the default scheduler if you do not specify the `_PScheduler` or `_PScheduleGroup` parameters.

The type `_Call_method` is a functor with signature `void (T const &)` which is invoked by this `call` messaging block to process a message.

The type `filter_method` is a functor with signature `bool (T const &)` which is invoked by this `call` messaging block to determine whether or not it should accept an offered message.

## ~call

Destroys the `call` messaging block.

```
~call();
```

## process\_input\_messages

Executes the call function on the input messages.

```
virtual void process_input_messages(_Inout_ message<T>* _PMessage);
```

### Parameters

*\_PMessage*

A pointer to the message that is to be handled.

## process\_message

Processes a message that was accepted by this `call` messaging block.

```
virtual void process_message(_Inout_ message<T>* _PMessage);
```

### Parameters

*\_PMessage*

A pointer to the message that is to be handled.

## propagate\_message

Asynchronously passes a message from an `ISource` block to this `call` messaging block. It is invoked by the `propagate` method, when called by a source block.

```
virtual message_status propagate_message(  
    _Inout_ message<T>* _PMessage,  
    _Inout_ ISource<T>* _PSource);
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A [message\\_status](#) indication of what the target decided to do with the message.

## send\_message

Synchronously passes a message from an [ISource](#) block to this [call](#) messaging block. It is invoked by the [send](#) method, when called by a source block.

```
virtual message_status send_message(  
    _Inout_ message<T>* _PMessage,  
    _Inout_ ISource<T>* _PSource);
```

### Parameters

*\_PMessage*

A pointer to the [message](#) object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A [message\\_status](#) indication of what the target decided to do with the message.

## supports\_anonymous\_source

Overrides the [supports\\_anonymous\\_source](#) method to indicate that this block can accept messages offered to it by a source that is not linked.

```
virtual bool supports_anonymous_source();
```

### Return Value

**true** because the block does not postpone offered messages.

## See also

[concurrency Namespace](#)

[transformer Class](#)

# cancellation\_token Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `cancellation_token` class represents the ability to determine whether some operation has been requested to cancel. A given token can be associated with a `task_group`, `structured_task_group`, or `task` to provide implicit cancellation. It can also be polled for cancellation or have a callback registered for if and when the associated `cancellation_token_source` is canceled.

## Syntax

```
class cancellation_token;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<code>cancellation_token</code>	
<code>~cancellation_token</code> Destructor	

### Public Methods

NAME	DESCRIPTION
<code>deregister_callback</code>	Removes a callback previously registered via the <code>register</code> method based on the <code>cancellation_token_registration</code> object returned at the time of registration.
<code>is_cancelable</code>	Returns an indication of whether this token can be canceled or not.
<code>is_canceled</code>	Returns <b>true</b> if the token has been canceled.
<code>none</code>	Returns a cancellation token which can never be subject to cancellation.
<code>register_callback</code>	Registers a callback function with the token. If and when the token is canceled, the callback will be made. Note that if the token is already canceled at the point where this method is called, the callback will be made immediately and synchronously.

### Public Operators

NAME	DESCRIPTION
<code>operator!=</code>	

NAME	DESCRIPTION
<code>operator=</code>	
<code>operator==</code>	

## Inheritance Hierarchy

`cancellation_token`

## Requirements

**Header:** `pplcancellation_token.h`

**Namespace:** `concurrency`

## `~cancellation_token`

```
~cancellation_token();
```

## `cancellation_token`

```
cancellation_token(const cancellation_token& _Src);
cancellation_token(cancellation_token&& _Src);
```

### Parameters

`_Src`

The `cancellation_token` to be copied or moved.

## `deregister_callback`

Removes a callback previously registered via the `register` method based on the `cancellation_token_registration` object returned at the time of registration.

```
void deregister_callback(const cancellation_token_registration& _Registration) const;
```

### Parameters

`_Registration`

The `cancellation_token_registration` object corresponding to the callback to be deregistered. This token must have been previously returned from a call to the `register` method.

## `is_cancelable`

Returns an indication of whether this token can be canceled or not.

```
bool is_cancelable() const;
```

### Return Value

An indication of whether this token can be canceled or not.

## is\_canceled

Returns **true** if the token has been canceled.

```
bool is_canceled() const;
```

### Return Value

The value **true** if the token has been canceled; otherwise, the value **false**.

## none

Returns a cancellation token which can never be subject to cancellation.

```
static cancellation_token none();
```

### Return Value

A cancellation token that cannot be canceled.

## operator!=

```
bool operator!= (const cancellation_token& _Src) const;
```

### Parameters

*\_Src*

The `cancellation_token` to compare.

### Return Value

## operator=

```
cancellation_token& operator= (const cancellation_token& _Src);  
cancellation_token& operator= (cancellation_token&& _Src);
```

### Parameters

*\_Src*

The `cancellation_token` to assign.

### Return Value

## operator==

```
bool operator== (const cancellation_token& _Src) const;
```

### Parameters

*\_Src*

The `cancellation_token` to compare.

## Return Value

# register\_callback

Registers a callback function with the token. If and when the token is canceled, the callback will be made. Note that if the token is already canceled at the point where this method is called, the callback will be made immediately and synchronously.

```
template<typename _Function>
::Concurrency::cancellation_token_registration register_callback(const _Function& _Func) const;
```

## Parameters

*\_Function*

The type of the function object that will be called back when this `cancellation_token` is canceled.

*\_Func*

The function object that will be called back when this `cancellation_token` is canceled.

## Return Value

A `cancellation_token_registration` object which can be utilized in the `deregister` method to deregister a previously registered callback and prevent it from being made. The method will throw an [invalid\\_operation](#) exception if it is called on a `cancellation_token` object that was created using the `cancellation_token::none` method.

## See also

[concurrency Namespace](#)

# cancellation\_token\_registration Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `cancellation_token_registration` class represents a callback notification from a `cancellation_token`. When the `register` method on a `cancellation_token` is used to receive notification of when cancellation occurs, a `cancellation_token_registration` object is returned as a handle to the callback so that the caller can request a specific callback no longer be made through use of the `deregister` method.

## Syntax

```
class cancellation_token_registration;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<code>cancellation_token_registration</code>	
<code>~cancellation_token_registration</code> Destructor	

### Public Operators

NAME	DESCRIPTION
<code>operator!=</code>	
<code>operator=</code>	
<code>operator==</code>	

## Inheritance Hierarchy

```
cancellation_token_registration
```

## Requirements

**Header:** `pplcancellation_token.h`

**Namespace:** `concurrency`

### `~cancellation_token_registration`

```
~cancellation_token_registration();
```

## `cancellation_token_registration`

```
cancellation_token_registration();  
  
cancellation_token_registration(const cancellation_token_registration& _Src);  
  
cancellation_token_registration(cancellation_token_registration&& _Src);
```

### Parameters

*\_Src*

The `cancellation_token_registration` to copy or move.

## operator!=

```
bool operator!= (const cancellation_token_registration& _Rhs) const;
```

### Parameters

*\_Rhs*

The `cancellation_token_registration` to compare.

### Return Value

## operator=

```
cancellation_token_registration& operator= (const cancellation_token_registration& _Src);  
  
cancellation_token_registration& operator= (cancellation_token_registration&& _Src);
```

### Parameters

*\_Src*

The `cancellation_token_registration` to assign.

### Return Value

## operator==

```
bool operator== (const cancellation_token_registration& _Rhs) const;
```

### Parameters

*\_Rhs*

The `cancellation_token_registration` to compare.

### Return Value

## See also

[concurrency Namespace](#)

# cancellation\_token\_source Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `cancellation_token_source` class represents the ability to cancel some cancelable operation.

## Syntax

```
class cancellation_token_source;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<code>cancellation_token_source</code>	Overloaded. Constructs a new <code>cancellation_token_source</code> . The source can be used to flag cancellation of some cancelable operation.
<code>~cancellation_token_source</code> Destructor	

### Public Methods

NAME	DESCRIPTION
<code>cancel</code>	Cancels the token. Any <code>task_group</code> , <code>structured_task_group</code> , or <code>task</code> which utilizes the token will be canceled upon this call and throw an exception at the next interruption point.
<code>create_linked_source</code>	Overloaded. Creates a <code>cancellation_token_source</code> which is canceled when the provided token is canceled.
<code>get_token</code>	Returns a cancellation token associated with this source. The returned token can be polled for cancellation or provide a callback if and when cancellation occurs.

### Public Operators

NAME	DESCRIPTION
<code>operator!=</code>	
<code>operator=</code>	
<code>operator==</code>	

## Inheritance Hierarchy

`cancellation_token_source`

# Requirements

**Header:** `pplcancellation_token.h`

**Namespace:** `concurrency`

## `~cancellation_token_source`

```
~cancellation_token_source();
```

## `cancel`

Cancel the token. Any `task_group`, `structured_task_group`, or `task` which utilizes the token will be canceled upon this call and throw an exception at the next interruption point.

```
void cancel() const;
```

## `cancellation_token_source`

Constructs a new `cancellation_token_source`. The source can be used to flag cancellation of some cancelable operation.

```
cancellation_token_source();  
cancellation_token_source(const cancellation_token_source& _Src);  
cancellation_token_source(cancellation_token_source&& _Src);
```

### Parameters

*\_Src*

Object to copy or move.

## `create_linked_source`

Creates a `cancellation_token_source` which is canceled when the provided token is canceled.

```
static cancellation_token_source create_linked_source(  
    cancellation_token& _Src);  
  
template<typename _Iter>  
static cancellation_token_source create_linked_source(_Iter _Begin, _Iter _End);
```

### Parameters

*\_Iter*

Iterator type.

*\_Src*

A token whose cancellation will cause cancellation of the returned token source. Note that the returned token source can also be canceled independently of the source contained in this parameter.

*\_Begin*

The C++ Standard Library iterator corresponding to the beginning of the range of tokens to listen for

cancellation of.

*\_End*

The C++ Standard Library iterator corresponding to the ending of the range of tokens to listen for cancellation of.

### Return Value

A `cancellation_token_source` which is canceled when the token provided by the `_Src` parameter is canceled.

## get\_token

Returns a cancellation token associated with this source. The returned token can be polled for cancellation or provide a callback if and when cancellation occurs.

```
cancellation_token get_token() const;
```

### Return Value

A cancellation token associated with this source.

## operator!=

```
bool operator!= (const cancellation_token_source& _Src) const;
```

### Parameters

*\_Src*

Operand.

### Return Value

## operator=

```
cancellation_token_source& operator= (const cancellation_token_source& _Src);  
cancellation_token_source& operator= (cancellation_token_source&& _Src);
```

### Parameters

*\_Src*

Operand.

### Return Value

## operator==

```
bool operator== (const cancellation_token_source& _Src) const;
```

### Parameters

*\_Src*

Operand.

### Return Value

See also



# choice Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

A `choice` messaging block is a multi-source, single-target block that represents a control-flow interaction with a set of sources. The choice block will wait for any one of multiple sources to produce a message and will propagate the index of the source that produced the message.

## Syntax

```
template<
    class T
>
class choice: public ISource<size_t>;
```

### Parameters

`T`

A `tuple`-based type representing the payloads of the input sources.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>type</code>	A type alias for <code>T</code> .

### Public Constructors

NAME	DESCRIPTION
<code>choice</code>	Overloaded. Constructs a <code>choice</code> messaging block.
<code>~choice</code> Destructor	Destroys the <code>choice</code> messaging block.

### Public Methods

NAME	DESCRIPTION
<code>accept</code>	Accepts a message that was offered by this <code>choice</code> block, transferring ownership to the caller.
<code>acquire_ref</code>	Acquires a reference count on this <code>choice</code> messaging block, to prevent deletion.
<code>consume</code>	Consumes a message previously offered by this <code>choice</code> messaging block and successfully reserved by the target, transferring ownership to the caller.
<code>has_value</code>	Checks whether this <code>choice</code> messaging block has been initialized with a value yet.

NAME	DESCRIPTION
<a href="#">index</a>	Returns an index into the <code>tuple</code> representing the element selected by the <code>choice</code> messaging block.
<a href="#">link_target</a>	Links a target block to this <code>choice</code> messaging block.
<a href="#">release</a>	Releases a previous successful message reservation.
<a href="#">release_ref</a>	Releases a reference count on this <code>choice</code> messaging block.
<a href="#">reserve</a>	Reserves a message previously offered by this <code>choice</code> messaging block.
<a href="#">unlink_target</a>	Unlinks a target block from this <code>choice</code> messaging block.
<a href="#">unlink_targets</a>	Unlinks all targets from this <code>choice</code> messaging block. (Overrides <code>ISource::unlink_targets</code> .)
<a href="#">value</a>	Gets the message whose index was selected by the <code>choice</code> messaging block.

## Remarks

The choice block ensures that only one of the incoming messages is consumed.

For more information, see [Asynchronous Message Blocks](#).

## Inheritance Hierarchy

[ISource](#)

`choice`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## accept

Accepts a message that was offered by this `choice` block, transferring ownership to the caller.

```
virtual message<size_t>* accept(
    runtime_object_identity _MsgId,
    _Inout_ ITarget<size_t>* _PTarget);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the offered `message` object.

*\_PTarget*

A pointer to the target block that is calling the `accept` method.

## Return Value

A pointer to the message that the caller now has ownership of.

# acquire\_ref

Acquires a reference count on this `choice` messaging block, to prevent deletion.

```
virtual void acquire_ref(_Inout_ ITarget<size_t>* _PTarget);
```

## Parameters

*\_PTarget*

A pointer to the target block that is calling this method.

## Remarks

This method is called by an `ITarget` object that is being linked to this source during the `link_target` method.

# choice

Constructs a `choice` messaging block.

```
explicit choice(
    T _Tuple);

choice(
    Scheduler& _PScheduler,
    T _Tuple);

choice(
    ScheduleGroup& _PScheduleGroup,
    T _Tuple);

choice(
    choice&& _Choice);
```

## Parameters

*\_Tuple*

A `tuple` of sources for the choice.

*\_PScheduler*

The `Scheduler` object within which the propagation task for the `choice` messaging block is scheduled.

*\_PScheduleGroup*

The `ScheduleGroup` object within which the propagation task for the `choice` messaging block is scheduled. The `Scheduler` object used is implied by the schedule group.

*\_Choice*

A `choice` messaging block to copy from. Note that the original object is orphaned, making this a move constructor.

## Remarks

The runtime uses the default scheduler if you do not specify the `_PScheduler` or `_PScheduleGroup` parameters.

Move construction is not performed under a lock, which means that it is up to the user to make sure that there are no light-weight tasks in flight at the time of moving. Otherwise, numerous races can occur, leading to exceptions or inconsistent state.

## ~choice

Destroys the `choice` messaging block.

```
~choice();
```

## consume

Consumes a message previously offered by this `choice` messaging block and successfully reserved by the target, transferring ownership to the caller.

```
virtual message<size_t>* consume(  
    runtime_object_identity _MsgId,  
    _Inout_ ITarget<size_t>* _PTarget);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the reserved `message` object.

*\_PTarget*

A pointer to the target block that is calling the `consume` method.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

The `consume` method is similar to `accept`, but must always be preceded by a call to `reserve` that returned **true**.

## has\_value

Checks whether this `choice` messaging block has been initialized with a value yet.

```
bool has_value() const;
```

### Return Value

**true** if the block has received a value, **false** otherwise.

## index

Returns an index into the `tuple` representing the element selected by the `choice` messaging block.

```
size_t index();
```

### Return Value

The message index.

### Remarks

The message payload can be extracted using the `get` method.

## link\_target

Links a target block to this `choice` messaging block.

```
virtual void link_target(_Inout_ ITarget<size_t>* _PTarget);
```

#### Parameters

*\_PTarget*

A pointer to an `ITarget` block to link to this `choice` messaging block.

## release

Releases a previous successful message reservation.

```
virtual void release(  
    runtime_object_identity _MsgId,  
    _Inout_ ITarget<size_t>* _PTarget);
```

#### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being released.

*\_PTarget*

A pointer to the target block that is calling the `release` method.

## release\_ref

Releases a reference count on this `choice` messaging block.

```
virtual void release_ref(_Inout_ ITarget<size_t>* _PTarget);
```

#### Parameters

*\_PTarget*

A pointer to the target block that is calling this method.

#### Remarks

This method is called by an `ITarget` object that is being unlinked from this source. The source block is allowed to release any resources reserved for the target block.

## reserve

Reserves a message previously offered by this `choice` messaging block.

```
virtual bool reserve(  
    runtime_object_identity _MsgId,  
    _Inout_ ITarget<size_t>* _PTarget);
```

#### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being reserved.

*\_PTarget*

A pointer to the target block that is calling the `reserve` method.

## Return Value

**true** if the message was successfully reserved, **false** otherwise. Reservations can fail for many reasons, including: the message was already reserved or accepted by another target, the source could deny reservations, and so forth.

## Remarks

After you call `reserve`, if it succeeds, you must call either `consume` or `release` in order to take or give up possession of the message, respectively.

## unlink\_target

Unlinks a target block from this `choice` messaging block.

```
virtual void unlink_target(_Inout_ ITarget<size_t>* _PTarget);
```

## Parameters

*\_PTarget*

A pointer to an `ITarget` block to unlink from this `choice` messaging block.

## unlink\_targets

Unlinks all targets from this `choice` messaging block.

```
virtual void unlink_targets();
```

## Remarks

This method does not need to be called from the destructor because destructor for the internal `single_assignment` block will unlink properly.

## value

Gets the message whose index was selected by the `choice` messaging block.

```
template <
    typename _Payload_type
>
_Payload_type const& value();
```

## Parameters

*\_Payload\_type*

The type of the message payload.

## Return Value

The payload of the message.

## Remarks

Because a `choice` messaging block can take inputs with different payload types, you must specify the type of the payload at the point of retrieval. You can determine the type based on the result of the `index` method.

## See also

[concurrency Namespace](#)

join Class

single\_assignment Class

# combinable Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `combinable<T>` object is intended to provide thread-private copies of data, to perform lock-free thread-local sub-computations during parallel algorithms. At the end of the parallel operation, the thread-private sub-computations can then be merged into a final result. This class can be used instead of a shared variable, and can result in a performance improvement if there would otherwise be a lot of contention on that shared variable.

## Syntax

```
template<typename T>
class combinable;
```

### Parameters

*T*

The data type of the final merged result. The type must have a copy constructor and a default constructor.

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">combinable</a>	Overloaded. Constructs a new <code>combinable</code> object.
<a href="#">~combinable Destructor</a>	Destroys a <code>combinable</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">clear</a>	Clears any intermediate computational results from a previous usage.
<a href="#">combine</a>	Computes a final value from the set of thread-local sub-computations by calling the supplied combine functor.
<a href="#">combine_each</a>	Computes a final value from the set of thread-local sub-computations by calling the supplied combine functor once per thread-local sub-computation. The final result is accumulated by the function object.
<a href="#">local</a>	Overloaded. Returns a reference to the thread-private sub-computation.

### Public Operators

NAME	DESCRIPTION
<code>operator=</code>	Assigns to a <code>combinable</code> object from another <code>combinable</code> object.

## Remarks

For more information, see [Parallel Containers and Objects](#).

## Inheritance Hierarchy

`combinable`

## Requirements

**Header:** `ppl.h`

**Namespace:** `concurrency`

## clear

Clears any intermediate computational results from a previous usage.

```
void clear();
```

## combinable

Constructs a new `combinable` object.

```
combinable();

template <typename _Function>
explicit combinable(_Function _FnInitialize);

combinable(const combinable& _Copy);
```

### Parameters

*\_Function*

The type of the initialization functor object.

*\_FnInitialize*

A function which will be called to initialize each new thread-private value of the type `T`. It must support a function call operator with the signature `T ()`.

*\_Copy*

An existing `combinable` object to be copied into this one.

### Remarks

The first constructor initializes new elements with the default constructor for the type `T`.

The second constructor initializes new elements using the initialization functor supplied as the `_FnInitialize` parameter.

The third constructor is the copy constructor.

## ~combinable

Destroys a `combinable` object.

```
~combinable();
```

## combine

Computes a final value from the set of thread-local sub-computations by calling the supplied combine functor.

```
template<typename _Function>  
T combine(_Function _FnCombine) const;
```

### Parameters

*\_Function*

The type of the function object that will be invoked to combine two thread-local sub-computations.

*\_FnCombine*

The functor that is used to combine the sub-computations. Its signature is `T (T, T)` or

`T (const T&, const T&)`, and it must be associative and commutative.

### Return Value

The final result of combining all the thread-private sub-computations.

## combine\_each

Computes a final value from the set of thread-local sub-computations by calling the supplied combine functor once per thread-local sub-computation. The final result is accumulated by the function object.

```
template<typename _Function>  
void combine_each(_Function _FnCombine) const;
```

### Parameters

*\_Function*

The type of the function object that will be invoked to combine a single thread-local sub-computation.

*\_FnCombine*

The functor that is used to combine one sub-computation. Its signature is `void (T)` or `void (const T&)`, and must be associative and commutative.

## local

Returns a reference to the thread-private sub-computation.

```
T& local();  
  
T& local(bool& _Exists);
```

### Parameters

*\_Exists*

A reference to a boolean. The boolean value referenced by this argument will be set to **true** if the sub-computation already existed on this thread, and set to **false** if this was the first sub-computation on this thread.

### Return Value

A reference to the thread-private sub-computation.

## operator=

Assigns to a `combinable` object from another `combinable` object.

```
combinable& operator= (const combinable& _Copy);
```

### Parameters

*\_Copy*

An existing `combinable` object to be copied into this one.

### Return Value

A reference to this `combinable` object.

## See also

[concurrency Namespace](#)

# concurrent\_priority\_queue Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

The `concurrent_priority_queue` class is a container that allows multiple threads to concurrently push and pop items. Items are popped in priority order where priority is determined by a functor supplied as a template argument.

## Syntax

```
template <typename T,  
         typename _Compare= std::less<T>,  
         typename _Ax = std::allocator<T>  
>,  
         typename _Ax = std::allocator<T>> class concurrent_priority_queue;
```

### Parameters

*T*

The data type of the elements to be stored in the priority queue.

*\_Compare*

The type of the function object that can compare two element values as sort keys to determine their relative order in the priority queue. This argument is optional and the binary predicate `less<T>` is the default value.

*\_Ax*

The type that represents the stored allocator object that encapsulates details about the allocation and deallocation of memory for the concurrent priority queue. This argument is optional and the default value is `allocator<T>`.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>allocator_type</code>	A type that represents the allocator class for the concurrent priority queue.
<code>const_reference</code>	A type that represents a const reference to an element of the type stored in a concurrent priority queue.
<code>reference</code>	A type that represents a reference to an element of the type stored in a concurrent priority queue.
<code>size_type</code>	A type that counts the number of elements in a concurrent priority queue.
<code>value_type</code>	A type that represents the data type stored in a concurrent priority queue.

### Public Constructors

NAME	DESCRIPTION
<code>concurrent_priority_queue</code>	Overloaded. Constructs a concurrent priority queue.

### Public Methods

NAME	DESCRIPTION
<code>clear</code>	Erases all elements in the concurrent priority. This method is not concurrency-safe.
<code>empty</code>	Tests if the concurrent priority queue is empty at the time this method is called. This method is concurrency-safe.
<code>get_allocator</code>	Returns a copy of the allocator used to construct the concurrent priority queue. This method is concurrency-safe.
<code>push</code>	Overloaded. Adds an element to the concurrent priority queue. This method is concurrency-safe.
<code>size</code>	Returns the number of elements in the concurrent priority queue. This method is concurrency-safe.
<code>swap</code>	Swaps the contents of two concurrent priority queues. This method is not concurrency-safe.
<code>try_pop</code>	Removes and returns the highest priority element from the queue if the queue is non-empty. This method is concurrency-safe.

### Public Operators

NAME	DESCRIPTION
<code>operator=</code>	Overloaded. Assigns the contents of another <code>concurrent_priority_queue</code> object to this one. This method is not concurrency-safe.

## Remarks

For detailed information on the `concurrent_priority_queue` class, see [Parallel Containers and Objects](#).

## Inheritance Hierarchy

`concurrent_priority_queue`

## Requirements

**Header:** `concurrent_priority_queue.h`

**Namespace:** `concurrency`

## `clear`

Erases all elements in the concurrent priority. This method is not concurrency-safe.

```
void clear();
```

### Remarks

`clear` is not concurrency-safe. You must ensure that no other threads are invoking methods on the concurrent priority queue when you call this method. `clear` does not free memory.

## concurrent\_priority\_queue

Constructs a concurrent priority queue.

```
explicit concurrent_priority_queue(
    const allocator_type& _Al = allocator_type());

explicit concurrent_priority_queue(
    size_type _Init_capacity,
    const allocator_type& _Al = allocator_type());

template<typename _InputIterator>
concurrent_priority_queue(_InputIterator _Begin,
    _InputIterator _End,
    const allocator_type& _Al = allocator_type());

concurrent_priority_queue(
    const concurrent_priority_queue& _Src);

concurrent_priority_queue(
    const concurrent_priority_queue& _Src,
    const allocator_type& _Al);

concurrent_priority_queue(
    concurrent_priority_queue&& _Src);

concurrent_priority_queue(
    concurrent_priority_queue&& _Src,
    const allocator_type& _Al);
```

### Parameters

#### *\_InputIterator*

The type of the input iterator.

#### *\_Al*

The allocator class to use with this object.

#### *\_Init\_capacity*

The initial capacity of the `concurrent_priority_queue` object.

#### *\_Begin*

The position of the first element in the range of elements to be copied.

#### *\_End*

The position of the first element beyond the range of elements to be copied.

#### *\_Src*

The source `concurrent_priority_queue` object to copy or move elements from.

### Remarks

All constructors store an allocator object `_Al` and initialize the priority queue.

The first constructor specifies an empty initial priority queue and optionally specifies an allocator.

The second constructor specifies a priority queue with an initial capacity `_Init_capacity` and optionally specifies an allocator.

The third constructor specifies values supplied by the iterator range [ `_Begin` , `_End` ) and optionally specifies an allocator.

The fourth and fifth constructors specify a copy of the priority queue `_Src` .

The sixth and seventh constructors specify a move of the priority queue `_Src` .

## empty

Tests if the concurrent priority queue is empty at the time this method is called. This method is concurrency-safe.

```
bool empty() const;
```

### Return Value

**true** if the priority queue was empty at the moment the function was called, **false** otherwise.

## get\_allocator

Returns a copy of the allocator used to construct the concurrent priority queue. This method is concurrency-safe.

```
allocator_type get_allocator() const;
```

### Return Value

A copy of the allocator used to construct the `concurrent_priority_queue` object.

## operator=

Assigns the contents of another `concurrent_priority_queue` object to this one. This method is not concurrency-safe.

```
concurrent_priority_queue& operator= (const concurrent_priority_queue& _Src);  
concurrent_priority_queue& operator= (concurrent_priority_queue&& _Src);
```

### Parameters

`_Src`

The source `concurrent_priority_queue` object.

### Return Value

A reference to this `concurrent_priority_queue` object.

## push

Adds an element to the concurrent priority queue. This method is concurrency-safe.

```
void push(const value_type& _Elem);  
void push(value_type&& _Elem);
```

### Parameters

*\_Elem*

The element to be added to the concurrent priority queue.

## size

Returns the number of elements in the concurrent priority queue. This method is concurrency-safe.

```
size_type size() const;
```

### Return Value

The number of elements in this `concurrent_priority_queue` object.

### Remarks

The returned size is guaranteed to include all elements added by calls to the function `push`. However, it may not reflect results of pending concurrent operations.

## swap

Swaps the contents of two concurrent priority queues. This method is not concurrency-safe.

```
void swap(concurrent_priority_queue& _Queue);
```

### Parameters

*\_Queue*

The `concurrent_priority_queue` object to swap contents with.

## try\_pop

Removes and returns the highest priority element from the queue if the queue is non-empty. This method is concurrency-safe.

```
bool try_pop(reference _Elem);
```

### Parameters

*\_Elem*

A reference to a variable that will be populated with the highest priority element, if the queue is non-empty.

### Return Value

**true** if a value was popped, **false** otherwise.

## See also

[concurrency Namespace](#)

[Parallel Containers and Objects](#)

# concurrent\_queue Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

The `concurrent_queue` class is a sequence container class that allows first-in, first-out access to its elements. It enables a limited set of concurrency-safe operations, such as `push` and `try_pop`.

## Syntax

```
template<typename T, class _Ax>  
class concurrent_queue: public ::Concurrency::details::_Concurrent_queue_base_v4;
```

### Parameters

`T`

The data type of the elements to be stored in the queue.

`_Ax`

The type that represents the stored allocator object that encapsulates details about the allocation and deallocation of memory for this concurrent queue. This argument is optional and the default value is `allocator<T>`.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>allocator_type</code>	A type that represents the allocator class for the concurrent queue.
<code>const_iterator</code>	A type that represents a non-thread-safe <code>const</code> iterator over elements in a concurrent queue.
<code>const_reference</code>	A type that provides a reference to a <code>const</code> element stored in a concurrent queue for reading and performing <code>const</code> operations.
<code>difference_type</code>	A type that provides the signed distance between two elements in a concurrent queue.
<code>iterator</code>	A type that represents a non-thread-safe iterator over the elements in a concurrent queue.
<code>reference</code>	A type that provides a reference to an element stored in a concurrent queue.
<code>size_type</code>	A type that counts the number of elements in a concurrent queue.
<code>value_type</code>	A type that represents the data type stored in a concurrent queue.

### Public Constructors

NAME	DESCRIPTION
<code>concurrent_queue</code>	Overloaded. Constructs a concurrent queue.
<code>~concurrent_queue</code> Destructor	Destroys the concurrent queue.

## Public Methods

NAME	DESCRIPTION
<code>clear</code>	Clears the concurrent queue, destroying any currently enqueued elements. This method is not concurrency-safe.
<code>empty</code>	Tests if the concurrent queue is empty at the moment this method is called. This method is concurrency-safe.
<code>get_allocator</code>	Returns a copy of the allocator used to construct the concurrent queue. This method is concurrency-safe.
<code>push</code>	Overloaded. Enqueues an item at tail end of the concurrent queue. This method is concurrency-safe.
<code>try_pop</code>	Dequeues an item from the queue if one is available. This method is concurrency-safe.
<code>unsafe_begin</code>	Overloaded. Returns an iterator of type <code>iterator</code> or <code>const_iterator</code> to the beginning of the concurrent queue. This method is not concurrency-safe.
<code>unsafe_end</code>	Overloaded. Returns an iterator of type <code>iterator</code> or <code>const_iterator</code> to the end of the concurrent queue. This method is not concurrency-safe.
<code>unsafe_size</code>	Returns the number of items in the queue. This method is not concurrency-safe.

## Remarks

For more information, see [Parallel Containers and Objects](#).

## Inheritance Hierarchy

`concurrent_queue`

## Requirements

**Header:** `concurrent_queue.h`

**Namespace:** `concurrency`

## `clear`

Clears the concurrent queue, destroying any currently enqueued elements. This method is not concurrency-safe.

```
void clear();
```

## concurrent\_queue

Constructs a concurrent queue.

```
explicit concurrent_queue(
    const allocator_type& _Al = allocator_type());

concurrent_queue(
    const concurrent_queue& _OtherQ,
    const allocator_type& _Al = allocator_type());

concurrent_queue(
    concurrent_queue&& _OtherQ,
    const allocator_type& _Al = allocator_type());

template<typename _InputIterator>
concurrent_queue(_InputIterator _Begin,
    _InputIterator _End);
```

### Parameters

*\_InputIterator*

The type of the input iterator that specifies a range of values.

*\_Al*

The allocator class to use with this object.

*\_OtherQ*

The source `concurrent_queue` object to copy or move elements from.

*\_Begin*

Position of the first element in the range of elements to be copied.

*\_End*

Position of the first element beyond the range of elements to be copied.

### Remarks

All constructors store an allocator object `_Al` and initialize the queue.

The first constructor specifies an empty initial queue and explicitly specifies the allocator type to be used.

The second constructor specifies a copy of the concurrent queue `_OtherQ`.

The third constructor specifies a move of the concurrent queue `_OtherQ`.

The fourth constructor specifies values supplied by the iterator range [ `_Begin` , `_End` ).

## ~concurrent\_queue

Destroys the concurrent queue.

```
~concurrent_queue();
```

## empty

Tests if the concurrent queue is empty at the moment this method is called. This method is concurrency-safe.

```
bool empty() const;
```

### Return Value

**true** if the concurrent queue was empty at the moment we looked, **false** otherwise.

### Remarks

While this method is concurrency-safe with respect to calls to the methods `push`, `try_pop`, and `empty`, the value returned might be incorrect by the time it is inspected by the calling thread.

## get\_allocator

Returns a copy of the allocator used to construct the concurrent queue. This method is concurrency-safe.

```
allocator_type get_allocator() const;
```

### Return Value

A copy of the allocator used to construct the concurrent queue.

## push

Enqueues an item at tail end of the concurrent queue. This method is concurrency-safe.

```
void push(const T& _Src);  
  
void push(T&& _Src);
```

### Parameters

`_Src`

The item to be added to the queue.

### Remarks

`push` is concurrency-safe with respect to calls to the methods `push`, `try_pop`, and `empty`.

## try\_pop

Dequeues an item from the queue if one is available. This method is concurrency-safe.

```
bool try_pop(T& _Dest);
```

### Parameters

`_Dest`

A reference to a location to store the dequeued item.

### Return Value

**true** if an item was successfully dequeued, **false** otherwise.

### Remarks

If an item was successfully dequeued, the parameter `_Dest` receives the dequeued value, the original value held in the queue is destroyed, and this function returns **true**. If there was no item to dequeue, this function returns

`false` without blocking, and the contents of the `_Dest` parameter are undefined.

`try_pop` is concurrency-safe with respect to calls to the methods `push`, `try_pop`, and `empty`.

## unsafe\_begin

Returns an iterator of type `iterator` or `const_iterator` to the beginning of the concurrent queue. This method is not concurrency-safe.

```
iterator unsafe_begin();  
const_iterator unsafe_begin() const;
```

### Return Value

An iterator of type `iterator` or `const_iterator` to the beginning of the concurrent queue object.

### Remarks

The iterators for the `concurrent_queue` class are primarily intended for debugging, as they are slow, and iteration is not concurrency-safe with respect to other queue operations.

## unsafe\_end

Returns an iterator of type `iterator` or `const_iterator` to the end of the concurrent queue. This method is not concurrency-safe.

```
iterator unsafe_end();  
const_iterator unsafe_end() const;
```

### Return Value

An iterator of type `iterator` or `const_iterator` to the end of the concurrent queue.

### Remarks

The iterators for the `concurrent_queue` class are primarily intended for debugging, as they are slow, and iteration is not concurrency-safe with respect to other queue operations.

## unsafe\_size

Returns the number of items in the queue. This method is not concurrency-safe.

```
size_type unsafe_size() const;
```

### Return Value

The size of the concurrent queue.

### Remarks

`unsafe_size` is not concurrency-safe and can produce incorrect results if called concurrently with calls to the methods `push`, `try_pop`, and `empty`.

## See also

[concurrency Namespace](#)

# concurrent\_unordered\_map Class

3/4/2019 • 12 minutes to read • [Edit Online](#)

The `concurrent_unordered_map` class is a concurrency-safe container that controls a varying-length sequence of elements of type `std::pair<const K, _Element_type>`. The sequence is represented in a way that enables concurrency-safe append, element access, iterator access, and iterator traversal operations.

## Syntax

```
template <typename K,
    typename _Element_type,
    typename _Hasher = std::hash<K>,
    typename key_equality = std::equal_to<K>,
    typename _Allocator_type = std::allocator<std::pair<const K,
        _Element_type>>
>,
    typename key_equality = std::equal_to<K>,
    typename _Allocator_type = std::allocator<std::pair<const K,
        _Element_type>>> class concurrent_unordered_map : public
    details::_Concurrent_hash<details::_Concurrent_unordered_map_traits<K,
        _Element_type,
    details::_Hash_compare<K,
        _Hasher,
    key_equality>,
        _Allocator_type,
    false>>;
```

### Parameters

*K*

The key type.

*\_Element\_type*

The mapped type.

*\_Hasher*

The hash function object type. This argument is optional and the default value is `std::hash<K>`.

*key\_equality*

The equality comparison function object type. This argument is optional and the default value is `std::equal_to<K>`.

*\_Allocator\_type*

The type that represents the stored allocator object that encapsulates details about the allocation and deallocation of memory for the concurrent unordered map. This argument is optional and the default value is

`std::allocator<std::pair<K, _Element_type>>`.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>allocator_type</code>	The type of an allocator for managing storage.

NAME	DESCRIPTION
<code>const_iterator</code>	The type of a constant iterator for the controlled sequence.
<code>const_local_iterator</code>	The type of a constant bucket iterator for the controlled sequence.
<code>const_pointer</code>	The type of a constant pointer to an element.
<code>const_reference</code>	The type of a constant reference to an element.
<code>difference_type</code>	The type of a signed distance between two elements.
<code>hasher</code>	The type of the hash function.
<code>iterator</code>	The type of an iterator for the controlled sequence.
<code>key_equal</code>	The type of the comparison function.
<code>key_type</code>	The type of an ordering key.
<code>local_iterator</code>	The type of a bucket iterator for the controlled sequence.
<code>mapped_type</code>	The type of a mapped value associated with each key.
<code>pointer</code>	The type of a pointer to an element.
<code>reference</code>	The type of a reference to an element.
<code>size_type</code>	The type of an unsigned distance between two elements.
<code>value_type</code>	The type of an element.

## Public Constructors

NAME	DESCRIPTION
<a href="#">concurrent_unordered_map</a>	Overloaded. Constructs a concurrent unordered map.

## Public Methods

NAME	DESCRIPTION
<a href="#">at</a>	Overloaded. Finds an element in a <code>concurrent_unordered_map</code> with a specified key value.. This method is concurrency-safe.
<a href="#">hash_function</a>	Gets the stored hash function object.
<a href="#">insert</a>	Overloaded. Adds elements to the <code>concurrent_unordered_map</code> object.



## Remarks

If the argument key value is not found, the function throws an object of class `out_of_range`.

## begin

Returns an iterator pointing to the first element in the concurrent container. This method is concurrency safe.

```
iterator begin();  
const_iterator begin() const;
```

## Return Value

An iterator to the first element in the concurrent container.

## cbegin

Returns a const iterator pointing to the first element in the concurrent container. This method is concurrency safe.

```
const_iterator cbegin() const;
```

## Return Value

A const iterator to the first element in the concurrent container.

## cend

Returns a const iterator pointing to the location succeeding the last element in the concurrent container. This method is concurrency safe.

```
const_iterator cend() const;
```

## Return Value

A const iterator to the location succeeding the last element in the concurrent container.

## clear

Erases all the elements in the concurrent container. This function is not concurrency safe.

```
void clear();
```

## concurrent\_unordered\_map

Constructs a concurrent unordered map.

```

explicit concurrent_unordered_map(
    size_type _Number_of_buckets = 8,
    const hasher& _Hasher = hasher(),
    const key_equal& key_equality = key_equal(),
    const allocator_type& _Allocator = allocator_type());

concurrent_unordered_map(
    const allocator_type& _Allocator);

template <typename _Iterator>
concurrent_unordered_map(_Iterator _Begin,
    _Iterator _End,
    size_type _Number_of_buckets = 8,
    const hasher& _Hasher = hasher(),
    const key_equal& key_equality = key_equal(),
    const allocator_type& _Allocator = allocator_type());

concurrent_unordered_map(
    const concurrent_unordered_map& _Umap);

concurrent_unordered_map(
    const concurrent_unordered_map& _Umap,
    const allocator_type& _Allocator);

concurrent_unordered_map(
    concurrent_unordered_map&& _Umap);

```

## Parameters

### *\_Iterator*

The type of the input iterator.

### *\_Number\_of\_buckets*

The initial number of buckets for this unordered map.

### *\_Hasher*

The hash function for this unordered map.

### *key\_equality*

The equality comparison function for this unordered map.

### *\_Allocator*

The allocator for this unordered map.

### *\_Begin*

The position of the first element in the range of elements to be copied.

### *\_End*

The position of the first element beyond the range of elements to be copied.

### *\_Umap*

The source `concurrent_unordered_map` object to copy or move elements from.

## Remarks

All constructors store an allocator object `_Allocator` and initialize the unordered map.

The first constructor specifies an empty initial map and explicitly specifies the number of buckets, hash function, equality function and allocator type to be used.

The second constructor specifies an allocator for the unordered map.

The third constructor specifies values supplied by the iterator range [ `_Begin`, `_End` ).

The fourth and fifth constructors specify a copy of the concurrent unordered map `_Umap`.

The last constructor specifies a move of the concurrent unordered map `_Umap`.

## count

Counts the number of elements matching a specified key. This function is concurrency safe.

```
size_type count(const key_type& KVal) const;
```

### Parameters

*KVal*

The key to search for.

### Return Value

The number of times number of times the key appears in the container.

## empty

Tests whether no elements are present. This method is concurrency safe.

```
bool empty() const;
```

### Return Value

**true** if the concurrent container is empty, **false** otherwise.

### Remarks

In the presence of concurrent inserts, whether or not the concurrent container is empty may change immediately after calling this function, before the return value is even read.

## end

Returns an iterator pointing to the location succeeding the last element in the concurrent container. This method is concurrency safe.

```
iterator end();  
const_iterator end() const;
```

### Return Value

An iterator to the location succeeding the last element in the concurrent container.

## equal\_range

Finds a range that matches a specified key. This function is concurrency safe.

```
std::pair<iterator,
  iterator> equal_range(
  const key_type& KVal);

std::pair<const_iterator,
  const_iterator> equal_range(
  const key_type& KVal) const;
```

### Parameters

*KVal*

The key value to search for.

### Return Value

A [pair](#) where the first element is an iterator to the beginning and the second element is an iterator to the end of the range.

### Remarks

It is possible for concurrent inserts to cause additional keys to be inserted after the begin iterator and before the end iterator.

## find

Finds an element that matches a specified key. This function is concurrency safe.

```
iterator find(const key_type& KVal);

const_iterator find(const key_type& KVal) const;
```

### Parameters

*KVal*

The key value to search for.

### Return Value

An iterator pointing to the location of the first element that matched the key provided, or the iterator `end()` if no such element exists.

## get\_allocator

Returns the stored allocator object for this concurrent container. This method is concurrency safe.

```
allocator_type get_allocator() const;
```

### Return Value

The stored allocator object for this concurrent container.

## hash\_function

Gets the stored hash function object.

```
hasher hash_function() const;
```

### Return Value

The stored hash function object.

## insert

Adds elements to the `concurrent_unordered_map` object.

```
std::pair<iterator,
    bool> insert(
    const value_type& value);

iterator insert(
    const_iterator _Where,
    const value_type& value);

template<class _Iterator>
void insert(_Iterator first,
    _Iterator last);

template<class V>
std::pair<iterator,
    bool> insert(
    V&& value);

template<class V>
typename std::enable_if<!std::is_same<const_iterator,
    typename std::remove_reference<V>::type>::value,
    iterator>::type insert(
    const_iterator _Where,
    V&& value);
```

### Parameters

*\_Iterator*

The iterator type used for insertion.

*V*

The type of the value inserted into the map.

*value*

The value to be inserted.

*\_Where*

The starting location to search for an insertion point.

*first*

The beginning of the range to insert.

*last*

The end of the range to insert.

### Return Value

A pair that contains an iterator and a boolean value. See the Remarks section for more details.

### Remarks

The first member function determines whether an element *X* exists in the sequence whose key has equivalent ordering to that of `value`. If not, it creates such an element *X* and initializes it with `value`. The function then determines the iterator `where` that designates *X*. If an insertion occurred, the function returns `std::pair(where, true)`. Otherwise, it returns `std::pair(where, false)`.

The second member function returns `insert(value)`, using `_Where` as a starting place within the controlled sequence to search for the insertion point.

The third member function inserts the sequence of element values from the range [ `first`, `last` ).

The last two member functions behave the same as the first two, except that `value` is used to construct the inserted value.

## key\_eq

Gets the stored equality comparison function object.

```
key_equal key_eq() const;
```

### Return Value

The stored equality comparison function object.

## load\_factor

Computes and returns the current load factor of the container. The load factor is the number of elements in the container divided by the number of buckets.

```
float load_factor() const;
```

### Return Value

The load factor for the container.

## max\_load\_factor

Gets or sets the maximum load factor of the container. The maximum load factor is the largest number of elements than can be in any bucket before the container grows its internal table.

```
float max_load_factor() const;  
void max_load_factor(float _Newmax);
```

### Parameters

`_Newmax`

### Return Value

The first member function returns the stored maximum load factor. The second member function does not return a value but throws an [out\\_of\\_range](#) exception if the supplied load factor is invalid..

## max\_size

Returns the maximum size of the concurrent container, determined by the allocator. This method is concurrency safe.

```
size_type max_size() const;
```

### Return Value

The maximum number of elements that can be inserted into this concurrent container.

### Remarks

This upper bound value may actually be higher than what the container can actually hold.

## operator[]

Finds or inserts an element with the specified key. This method is concurrency-safe.

```
mapped_type& operator[](const key_type& kval);  
  
mapped_type& operator[](key_type&& kval);
```

### Parameters

*KVal*

The key value to

find or insert.

### Return Value

A reference to the data value of the found or inserted element.

### Remarks

If the argument key value is not found, then it is inserted along with the default value of the data type.

`operator[]` may be used to insert elements into a map `m` using `m[key] = DataValue;`, where `DataValue` is the value of the `mapped_type` of the element with a key value of `key`.

When using `operator[]` to insert elements, the returned reference does not indicate whether an insertion is changing a pre-existing element or creating a new one. The member functions `find` and `insert` can be used to determine whether an element with a specified key is already present before an insertion.

## operator=

Assigns the contents of another `concurrent_unordered_map` object to this one. This method is not concurrency-safe.

```
concurrent_unordered_map& operator= (const concurrent_unordered_map& _Umap);  
  
concurrent_unordered_map& operator= (concurrent_unordered_map&& _Umap);
```

### Parameters

*\_Umap*

The source `concurrent_unordered_map` object.

### Return Value

A reference to this `concurrent_unordered_map` object.

### Remarks

After erasing any existing elements a concurrent vector, `operator=` either copies or moves the contents of `_Umap` into the concurrent vector.

## rehash

Rebuilds the hash table.

```
void rehash(size_type _Buckets);
```

## Parameters

*\_Buckets*

The desired number of buckets.

## Remarks

The member function alters the number of buckets to be at least `_Buckets` and rebuilds the hash table as needed. The number of buckets must be a power of 2. If not a power of 2, it will be rounded up to the next largest power of 2.

It throws an [out\\_of\\_range](#) exception if the number of buckets is invalid (either 0 or greater than the maximum number of buckets).

## size

Returns the number of elements in this concurrent container. This method is concurrency safe.

```
size_type size() const;
```

## Return Value

The number of items in the container.

## Remarks

In the presence of concurrent inserts, the number of elements in the concurrent container may change immediately after calling this function, before the return value is even read.

## swap

Swaps the contents of two `concurrent_unordered_map` objects. This method is not concurrency-safe.

```
void swap(concurrent_unordered_map& _Umap);
```

## Parameters

*\_Umap*

The `concurrent_unordered_map` object to swap with.

## unsafe\_begin

Returns an iterator to the first element in this container for a specific bucket.

```
local_iterator unsafe_begin(size_type _Bucket);  
  
const_local_iterator unsafe_begin(size_type _Bucket) const;
```

## Parameters

*\_Bucket*

The bucket index.

## Return Value

An iterator pointing to the beginning of the bucket.

## unsafe\_bucket

Returns the bucket index that a specific key maps to in this container.

```
size_type unsafe_bucket(const key_type& KVal) const;
```

#### Parameters

*KVal*

The element key being searched for.

#### Return Value

The bucket index for the key in this container.

## unsafe\_bucket\_count

Returns the current number of buckets in this container.

```
size_type unsafe_bucket_count() const;
```

#### Return Value

The current number of buckets in this container.

## unsafe\_bucket\_size

Returns the number of items in a specific bucket of this container.

```
size_type unsafe_bucket_size(size_type _Bucket);
```

#### Parameters

*\_Bucket*

The bucket to search for.

#### Return Value

The current number of buckets in this container.

## unsafe\_cbegin

Returns an iterator to the first element in this container for a specific bucket.

```
const_local_iterator unsafe_cbegin(size_type _Bucket) const;
```

#### Parameters

*\_Bucket*

The bucket index.

#### Return Value

An iterator pointing to the beginning of the bucket.

## unsafe\_cend

Returns an iterator to the location succeeding the last element in a specific bucket.

```
const_local_iterator unsafe_cend(size_type _Bucket) const;
```

### Parameters

*\_Bucket*

The bucket index.

### Return Value

An iterator pointing to the beginning of the bucket.

## unsafe\_end

Returns an iterator to the last element in this container for a specific bucket.

```
local_iterator unsafe_end(size_type _Bucket);  
  
const_local_iterator unsafe_end(size_type _Bucket) const;
```

### Parameters

*\_Bucket*

The bucket index.

### Return Value

An iterator pointing to the end of the bucket.

## unsafe\_erase

Removes elements from the `concurrent_unordered_map` at specified positions. This method is not concurrency-safe.

```
iterator unsafe_erase(  
    const_iterator _Where);  
  
iterator unsafe_erase(  
    const_iterator _Begin,  
    const_iterator _End);  
  
size_type unsafe_erase(  
    const key_type& KVal);
```

### Parameters

*\_Where*

The iterator position to erase from.

*\_Begin*

The position of the first element in the range of elements to be erased.

*\_End*

The position of the first element beyond the range of elements to be erased.

*KVal*

The key value to erase.

### Return Value

The first two member functions return an iterator that designates the first element remaining beyond any

elements removed, or `concurrent_unordered_map::end()` if no such element exists. The third member function returns the number of elements it removes.

### Remarks

The first member function removes the element of the controlled sequence pointed to by `_Where`. The second member function removes the elements in the range `[_Begin, _End)`.

The third member function removes the elements in the range delimited by `concurrent_unordered_map::equal_range(KVal)`.

## unsafe\_max\_bucket\_count

Returns the maximum number of buckets in this container.

```
size_type unsafe_max_bucket_count() const;
```

### Return Value

The maximum number of buckets in this container.

## See also

[concurrency Namespace](#)

[Parallel Containers and Objects](#)

# concurrent\_unordered\_multimap Class

3/4/2019 • 10 minutes to read • [Edit Online](#)

The `concurrent_unordered_multimap` class is an concurrency-safe container that controls a varying-length sequence of elements of type `std::pair<const K, _Element_type>`. The sequence is represented in a way that enables concurrency-safe append, element access, iterator access and iterator traversal operations.

## Syntax

```
template <typename K,
          typename _Element_type,
          typename _Hasher = std::hash<K>,
          typename key_equality = std::equal_to<K>,
          typename _Allocator_type = std::allocator<std::pair<const K,
          _Element_type>>
>,
typename key_equality = std::equal_to<K>,
          typename _Allocator_type = std::allocator<std::pair<const K,
          _Element_type>>> class concurrent_unordered_multimap : public
details::_Concurrent_hash<details::_Concurrent_unordered_map_traits<K,
          _Element_type,
details::_Hash_compare<K,
          _Hasher,
key_equality>,
          _Allocator_type,
true>>;
```

### Parameters

*K*

The key type.

*\_Element\_type*

The mapped type.

*\_Hasher*

The hash function object type. This argument is optional and the default value is `std::hash<K>`.

*key\_equality*

The equality comparison function object type. This argument is optional and the default value is `std::equal_to<K>`.

*\_Allocator\_type*

The type that represents the stored allocator object that encapsulates details about the allocation and deallocation of memory for the concurrent vector. This argument is optional and the default value is

`std::allocator<std::pair<K, _Element_type>>`.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>allocator_type</code>	The type of an allocator for managing storage.

NAME	DESCRIPTION
<code>const_iterator</code>	The type of a constant iterator for the controlled sequence.
<code>const_local_iterator</code>	The type of a constant bucket iterator for the controlled sequence.
<code>const_pointer</code>	The type of a constant pointer to an element.
<code>const_reference</code>	The type of a constant reference to an element.
<code>difference_type</code>	The type of a signed distance between two elements.
<code>hasher</code>	The type of the hash function.
<code>iterator</code>	The type of an iterator for the controlled sequence.
<code>key_equal</code>	The type of the comparison function.
<code>key_type</code>	The type of an ordering key.
<code>local_iterator</code>	The type of a bucket iterator for the controlled sequence.
<code>mapped_type</code>	The type of a mapped value associated with each key.
<code>pointer</code>	The type of a pointer to an element.
<code>reference</code>	The type of a reference to an element.
<code>size_type</code>	The type of an unsigned distance between two elements.
<code>value_type</code>	The type of an element.

### Public Constructors

NAME	DESCRIPTION
<a href="#">concurrent_unordered_multimap</a>	Overloaded. Constructs a concurrent unordered multimap.

### Public Methods

NAME	DESCRIPTION
<a href="#">hash_function</a>	Returns the stored hash function object.
<a href="#">insert</a>	Overloaded. Adds elements to the <code>concurrent_unordered_multimap</code> object.
<a href="#">key_eq</a>	Returns the stored equality comparison function object.

NAME	DESCRIPTION
<code>swap</code>	Swaps the contents of two <code>concurrent_unordered_multimap</code> objects. This method is not concurrency-safe.
<code>unsafe_erase</code>	Overloaded. Removes elements from the <code>concurrent_unordered_multimap</code> at specified positions. This method is not concurrency-safe.

### Public Operators

NAME	DESCRIPTION
<code>operator=</code>	Overloaded. Assigns the contents of another <code>concurrent_unordered_multimap</code> object to this one. This method is not concurrency-safe.

## Remarks

For detailed information on the `concurrent_unordered_multimap` class, see [Parallel Containers and Objects](#).

## Inheritance Hierarchy

`_Traits`

`_Concurrent_hash`

`concurrent_unordered_multimap`

## Requirements

**Header:** `concurrent_unordered_map.h`

**Namespace:** `concurrency`

## `begin`

Returns an iterator pointing to the first element in the concurrent container. This method is concurrency safe.

```
iterator begin();
const_iterator begin() const;
```

### Return Value

An iterator to the first element in the concurrent container.

## `cbegin`

Returns a const iterator pointing to the first element in the concurrent container. This method is concurrency safe.

```
const_iterator cbegin() const;
```

### Return Value

A const iterator to the first element in the concurrent container.

## cend

Returns a const iterator pointing to the location succeeding the last element in the concurrent container. This method is concurrency safe.

```
const_iterator cend() const;
```

### Return Value

A const iterator to the location succeeding the last element in the concurrent container.

## clear

Erases all the elements in the concurrent container. This function is not concurrency safe.

```
void clear();
```

## concurrent\_unordered\_multimap

Constructs a concurrent unordered multimap.

```
explicit concurrent_unordered_multimap(
    size_type _Number_of_buckets = 8,
    const hasher& _Hasher = hasher(),
    const key_equal& key_equality = key_equal(),
    const allocator_type& _Allocator = allocator_type());

concurrent_unordered_multimap(
    const allocator_type& _Allocator);

template <typename _Iterator>
concurrent_unordered_multimap(_Iterator _Begin,
    _Iterator _End,
    size_type _Number_of_buckets = 8,
    const hasher& _Hasher = hasher(),
    const key_equal& key_equality = key_equal(),
    const allocator_type& _Allocator = allocator_type());

concurrent_unordered_multimap(
    const concurrent_unordered_multimap& _Umap);

concurrent_unordered_multimap(
    const concurrent_unordered_multimap& _Umap,
    const allocator_type& _Allocator);

concurrent_unordered_multimap(
    concurrent_unordered_multimap&& _Umap);
```

### Parameters

*\_Iterator*

The type of the input iterator.

*\_Number\_of\_buckets*

The initial number of buckets for this unordered multimap.

*\_Hasher*

The hash function for this unordered multimap.

*key\_equality*

The equality comparison function for this unordered multimap.

*\_Allocator*

The allocator for this unordered multimap.

*\_Begin*

The position of the first element in the range of elements to be copied.

*\_End*

The position of the first element beyond the range of elements to be copied.

*\_Umap*

The source `concurrent_unordered_multimap` object to copy elements from.

### Remarks

All constructors store an allocator object `_Allocator` and initialize the unordered multimap.

The first constructor specifies an empty initial multimap and explicitly specifies the number of buckets, hash function, equality function and allocator type to be used.

The second constructor specifies an allocator for the unordered multimap.

The third constructor specifies values supplied by the iterator range [ `_Begin` , `_End` ).

The fourth and fifth constructors specify a copy of the concurrent unordered multimap `_Umap` .

The last constructor specifies a move of the concurrent unordered multimap `_Umap` .

## count

Counts the number of elements matching a specified key. This function is concurrency safe.

```
size_type count(const key_type& KVal) const;
```

### Parameters

*KVal*

The key to search for.

### Return Value

The number of times number of times the key appears in the container.

## empty

Tests whether no elements are present. This method is concurrency safe.

```
bool empty() const;
```

### Return Value

**true** if the concurrent container is empty, **false** otherwise.

### Remarks

In the presence of concurrent inserts, whether or not the concurrent container is empty may change immediately after calling this function, before the return value is even read.

## end

Returns an iterator pointing to the location succeeding the last element in the concurrent container. This method is concurrency safe.

```
iterator end();

const_iterator end() const;
```

### Return Value

An iterator to the location succeeding the last element in the concurrent container.

## equal\_range

Finds a range that matches a specified key. This function is concurrency safe.

```
std::pair<iterator,
iterator> equal_range(
    const key_type& KVal);

std::pair<const_iterator,
const_iterator> equal_range(
    const key_type& KVal) const;
```

### Parameters

*KVal*

The key value to search for.

### Return Value

A [pair](#) where the first element is an iterator to the beginning and the second element is an iterator to the end of the range.

### Remarks

It is possible for concurrent inserts to cause additional keys to be inserted after the begin iterator and before the end iterator.

## find

Finds an element that matches a specified key. This function is concurrency safe.

```
iterator find(const key_type& KVal);

const_iterator find(const key_type& KVal) const;
```

### Parameters

*KVal*

The key value to search for.

### Return Value

An iterator pointing to the location of the first element that matched the key provided, or the iterator `end()` if no such element exists.

## get\_allocator

Returns the stored allocator object for this concurrent container. This method is concurrency safe.

```
allocator_type get_allocator() const;
```

### Return Value

The stored allocator object for this concurrent container.

## hash\_function

Returns the stored hash function object.

```
hasher hash_function() const;
```

### Return Value

The stored hash function object.

## insert

Adds elements to the `concurrent_unordered_multimap` object.

```
iterator insert(
    const value_type& value);

iterator insert(
    const_iterator _where,
    const value_type& value);

template<class _Iterator>
void insert(_Iterator first,
    _Iterator last);

template<class V>
iterator insert(
    V&& value);

template<class V>
typename std::enable_if<!std::is_same<const_iterator,
    typename std::remove_reference<V>::type>::value,
    iterator>::type insert(
    const_iterator _where,
    V&& value);
```

### Parameters

*\_Iterator*

The iterator type used for insertion.

*V*

The type of the value inserted into the map.

*value*

The value to be inserted.

*\_Where*

The starting location to search for an insertion point.

*first*

The beginning of the range to insert.

*last*

The end of the range to insert.

### Return Value

An iterator pointing to the insertion location.

### Remarks

The first member function inserts the element `value` in the controlled sequence, then returns the iterator that designates the inserted element.

The second member function returns `insert(value)`, using `_where` as a starting place within the controlled sequence to search for the insertion point.

The third member function inserts the sequence of element values from the range `[first, last)`.

The last two member functions behave the same as the first two, except that `value` is used to construct the inserted value.

## key\_eq

Returns the stored equality comparison function object.

```
key_equal key_eq() const;
```

### Return Value

The stored equality comparison function object.

## load\_factor

Computes and returns the current load factor of the container. The load factor is the number of elements in the container divided by the number of buckets.

```
float load_factor() const;
```

### Return Value

The load factor for the container.

## max\_load\_factor

Gets or sets the maximum load factor of the container. The maximum load factor is the largest number of elements than can be in any bucket before the container grows its internal table.

```
float max_load_factor() const;  
void max_load_factor(float _Newmax);
```

### Parameters

`_Newmax`

### Return Value

The first member function returns the stored maximum load factor. The second member function does not return a value but throws an [out\\_of\\_range](#) exception if the supplied load factor is invalid..

## max\_size

Returns the maximum size of the concurrent container, determined by the allocator. This method is concurrency safe.

```
size_type max_size() const;
```

### Return Value

The maximum number of elements that can be inserted into this concurrent container.

### Remarks

This upper bound value may actually be higher than what the container can actually hold.

## operator=

Assigns the contents of another `concurrent_unordered_multimap` object to this one. This method is not concurrency-safe.

```
concurrent_unordered_multimap& operator= (const concurrent_unordered_multimap& _Umap);  
  
concurrent_unordered_multimap& operator= (concurrent_unordered_multimap&& _Umap);
```

### Parameters

*\_Umap*

The source `concurrent_unordered_multimap` object.

### Return Value

A reference to this `concurrent_unordered_multimap` object.

### Remarks

After erasing any existing elements in a concurrent unordered multimap, `operator=` either copies or moves the contents of `_Umap` into the concurrent unordered multimap.

## rehash

Rebuilds the hash table.

```
void rehash(size_type _Buckets);
```

### Parameters

*\_Buckets*

The desired number of buckets.

### Remarks

The member function alters the number of buckets to be at least `_Buckets` and rebuilds the hash table as needed. The number of buckets must be a power of 2. If not a power of 2, it will be rounded up to the next largest power of 2.

It throws an `out_of_range` exception if the number of buckets is invalid (either 0 or greater than the maximum number of buckets).

## size

Returns the number of elements in this concurrent container. This method is concurrency safe.

```
size_type size() const;
```

### Return Value

The number of items in the container.

### Remarks

In the presence of concurrent inserts, the number of elements in the concurrent container may change immediately after calling this function, before the return value is even read.

## swap

Swaps the contents of two `concurrent_unordered_multimap` objects. This method is not concurrency-safe.

```
void swap(concurrent_unordered_multimap& _Umap);
```

### Parameters

*\_Umap*

The `concurrent_unordered_multimap` object to swap with.

## unsafe\_begin

Returns an iterator to the first element in this container for a specific bucket.

```
local_iterator unsafe_begin(size_type _Bucket);  
  
const_local_iterator unsafe_begin(size_type _Bucket) const;
```

### Parameters

*\_Bucket*

The bucket index.

### Return Value

An iterator pointing to the beginning of the bucket.

## unsafe\_bucket

Returns the bucket index that a specific key maps to in this container.

```
size_type unsafe_bucket(const key_type& KVal) const;
```

### Parameters

*KVal*

The element key being searched for.

### Return Value

The bucket index for the key in this container.

## unsafe\_bucket\_count

Returns the current number of buckets in this container.

```
size_type unsafe_bucket_count() const;
```

#### Return Value

The current number of buckets in this container.

## unsafe\_bucket\_size

Returns the number of items in a specific bucket of this container.

```
size_type unsafe_bucket_size(size_type _Bucket);
```

#### Parameters

*\_Bucket*

The bucket to search for.

#### Return Value

The current number of buckets in this container.

## unsafe\_cbegin

Returns an iterator to the first element in this container for a specific bucket.

```
const_local_iterator unsafe_cbegin(size_type _Bucket) const;
```

#### Parameters

*\_Bucket*

The bucket index.

#### Return Value

An iterator pointing to the beginning of the bucket.

## unsafe\_cend

Returns an iterator to the location succeeding the last element in a specific bucket.

```
const_local_iterator unsafe_cend(size_type _Bucket) const;
```

#### Parameters

*\_Bucket*

The bucket index.

#### Return Value

An iterator pointing to the beginning of the bucket.

## unsafe\_end

Returns an iterator to the last element in this container for a specific bucket.

```
local_iterator unsafe_end(size_type _Bucket);

const_local_iterator unsafe_end(size_type _Bucket) const;
```

### Parameters

*\_Bucket*

The bucket index.

### Return Value

An iterator pointing to the end of the bucket.

## unsafe\_erase

Removes elements from the `concurrent_unordered_multimap` at specified positions. This method is not concurrency-safe.

```
iterator unsafe_erase(
    const_iterator _Where);

size_type unsafe_erase(
    const key_type& KVal);

iterator unsafe_erase(
    const_iterator first,
    const_iterator last);
```

### Parameters

*\_Where*

The iterator position to erase from.

*KVal*

The key value to erase.

*first*

*last*

Iterators.

### Return Value

The first two member functions return an iterator that designates the first element remaining beyond any elements removed, or `concurrent_unordered_multimap::end()` if no such element exists. The third member function returns the number of elements it removes.

### Remarks

The first member function removes the element of the controlled sequence pointed to by `_Where`. The second member function removes the elements in the range `[_Begin, _End)`.

The third member function removes the elements in the range delimited by

`concurrent_unordered_multimap::equal_range(KVal)`.

## unsafe\_max\_bucket\_count

Returns the maximum number of buckets in this container.

```
size_type unsafe_max_bucket_count() const;
```

**Return Value**

The maximum number of buckets in this container.

## See also

[concurrency Namespace](#)

[Parallel Containers and Objects](#)

# concurrent\_unordered\_multiset Class

3/4/2019 • 10 minutes to read • [Edit Online](#)

The `concurrent_unordered_multiset` class is an concurrency-safe container that controls a varying-length sequence of elements of type `K`. The sequence is represented in a way that enables concurrency-safe append, element access, iterator access and iterator traversal operations.

## Syntax

```
template <typename K,  
    typename _Hasher = std::hash<K>,  
    typename key_equality = std::equal_to<K>,  
    typename _Allocator_type = std::allocator<K>  
>,  
    typename key_equality = std::equal_to<K>,  
    typename _Allocator_type = std::allocator<K>> class concurrent_unordered_multiset : public  
    details::_Concurrent_hash<details::_Concurrent_unordered_set_traits<K,  
        details::_Hash_compare<K,  
            _Hasher,  
                key_equality>,  
            _Allocator_type,  
                true>>;
```

### Parameters

*K*

The key type.

*\_Hasher*

The hash function object type. This argument is optional and the default value is `std::hash<K>`.

*key\_equality*

The equality comparison function object type. This argument is optional and the default value is `std::equal_to<K>`.

*\_Allocator\_type*

The type that represents the stored allocator object that encapsulates details about the allocation and deallocation of memory for the concurrent vector. This argument is optional and the default value is `std::allocator<K>`.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>allocator_type</code>	The type of an allocator for managing storage.
<code>const_iterator</code>	The type of a constant iterator for the controlled sequence.
<code>const_local_iterator</code>	The type of a constant bucket iterator for the controlled sequence.
<code>const_pointer</code>	The type of a constant pointer to an element.

NAME	DESCRIPTION
<code>const_reference</code>	The type of a constant reference to an element.
<code>difference_type</code>	The type of a signed distance between two elements.
<code>hasher</code>	The type of the hash function.
<code>iterator</code>	The type of an iterator for the controlled sequence.
<code>key_equal</code>	The type of the comparison function.
<code>key_type</code>	The type of an ordering key.
<code>local_iterator</code>	The type of a bucket iterator for the controlled sequence.
<code>pointer</code>	The type of a pointer to an element.
<code>reference</code>	The type of a reference to an element.
<code>size_type</code>	The type of an unsigned distance between two elements.
<code>value_type</code>	The type of an element.

### Public Constructors

NAME	DESCRIPTION
<a href="#">concurrent_unordered_multiset</a>	Overloaded. Constructs a concurrent unordered multiset.

### Public Methods

NAME	DESCRIPTION
<a href="#">hash_function</a>	Returns the stored hash function object.
<a href="#">insert</a>	Overloaded. Adds elements to the <code>concurrent_unordered_multiset</code> object.
<a href="#">key_eq</a>	The stored equality comparison function object.
<a href="#">swap</a>	Swaps the contents of two <code>concurrent_unordered_multiset</code> objects. This method is not concurrency-safe.
<a href="#">unsafe_erase</a>	Overloaded. Removes elements from the <code>concurrent_unordered_multiset</code> at specified positions. This method is not concurrency-safe.

### Public Operators

NAME	DESCRIPTION
<code>operator=</code>	Overloaded. Assigns the contents of another <code>concurrent_unordered_multiset</code> object to this one. This method is not concurrency-safe.

## Remarks

For detailed information on the `concurrent_unordered_multiset` class, see [Parallel Containers and Objects](#).

## Inheritance Hierarchy

`_Traits`

`_Concurrent_hash`

`concurrent_unordered_multiset`

## Requirements

**Header:** `concurrent_unordered_set.h`

**Namespace:** `concurrency`

## `begin`

Returns an iterator pointing to the first element in the concurrent container. This method is concurrency safe.

```
iterator begin();
const_iterator begin() const;
```

### Return Value

An iterator to the first element in the concurrent container.

## `cbegin`

Returns a const iterator pointing to the first element in the concurrent container. This method is concurrency safe.

```
const_iterator cbegin() const;
```

### Return Value

A const iterator to the first element in the concurrent container.

## `cend`

Returns a const iterator pointing to the location succeeding the last element in the concurrent container. This method is concurrency safe.

```
const_iterator cend() const;
```

### Return Value

A const iterator to the location succeeding the last element in the concurrent container.

## clear

Erases all the elements in the concurrent container. This function is not concurrency safe.

```
void clear();
```

## concurrent\_unordered\_multiset

Constructs a concurrent unordered multiset.

```
explicit concurrent_unordered_multiset(
    size_type _Number_of_buckets = 8,
    const hasher& _Hasher = hasher(),
    const key_equal& key_equality = key_equal(),
    const allocator_type& _Allocator = allocator_type());

concurrent_unordered_multiset(
    const allocator_type& _Allocator);

template <typename _Iterator>
concurrent_unordered_multiset(_Iterator first,
    _Iterator last,
    size_type _Number_of_buckets = 8,
    const hasher& _Hasher = hasher(),
    const key_equal& key_equality = key_equal(),
    const allocator_type& _Allocator = allocator_type());

concurrent_unordered_multiset(
    const concurrent_unordered_multiset& _Uset);

concurrent_unordered_multiset(
    const concurrent_unordered_multiset& _Uset,
    const allocator_type& _Allocator);

concurrent_unordered_multiset(
    concurrent_unordered_multiset&& _Uset);
```

### Parameters

#### *\_Iterator*

The type of the input iterator.

#### *\_Number\_of\_buckets*

The initial number of buckets for this unordered multiset.

#### *\_Hasher*

The hash function for this unordered multiset.

#### *key\_equality*

The equality comparison function for this unordered multiset.

#### *\_Allocator*

The allocator for this unordered multiset.

#### *first*

#### *last*

#### *\_Uset*

The source `concurrent_unordered_multiset` object to move elements from.

## Remarks

All constructors store an allocator object `_Allocator` and initialize the unordered multiset.

The first constructor specifies an empty initial multiset and explicitly specifies the number of buckets, hash function, equality function and allocator type to be used.

The second constructor specifies an allocator for the unordered multiset.

The third constructor specifies values supplied by the iterator range [ `_Begin` , `_End` ).

The fourth and fifth constructors specify a copy of the concurrent unordered multiset `_Uset` .

The last constructor specifies a move of the concurrent unordered multiset `_Uset` .

## count

Counts the number of elements matching a specified key. This function is concurrency safe.

```
size_type count(const key_type& KVal) const;
```

### Parameters

*KVal*

The key to search for.

### Return Value

The number of times number of times the key appears in the container.

## empty

Tests whether no elements are present. This method is concurrency safe.

```
bool empty() const;
```

### Return Value

**true** if the concurrent container is empty, **false** otherwise.

### Remarks

In the presence of concurrent inserts, whether or not the concurrent container is empty may change immediately after calling this function, before the return value is even read.

## end

Returns an iterator pointing to the location succeeding the last element in the concurrent container. This method is concurrency safe.

```
iterator end();  
  
const_iterator end() const;
```

### Return Value

An iterator to the location succeeding the last element in the concurrent container.

## equal\_range

Finds a range that matches a specified key. This function is concurrency safe.

```
std::pair<iterator,
iterator> equal_range(
    const key_type& KVal);

std::pair<const_iterator,
const_iterator> equal_range(
    const key_type& KVal) const;
```

### Parameters

*KVal*

The key value to search for.

### Return Value

A [pair](#) where the first element is an iterator to the beginning and the second element is an iterator to the end of the range.

### Remarks

It is possible for concurrent inserts to cause additional keys to be inserted after the begin iterator and before the end iterator.

## find

Finds an element that matches a specified key. This function is concurrency safe.

```
iterator find(const key_type& KVal);

const_iterator find(const key_type& KVal) const;
```

### Parameters

*KVal*

The key value to search for.

### Return Value

An iterator pointing to the location of the first element that matched the key provided, or the iterator `end()` if no such element exists.

## get\_allocator

Returns the stored allocator object for this concurrent container. This method is concurrency safe.

```
allocator_type get_allocator() const;
```

### Return Value

The stored allocator object for this concurrent container.

## hash\_function

Returns the stored hash function object.

```
hasher hash_function() const;
```

## Return Value

The stored hash function object.

# insert

Adds elements to the `concurrent_unordered_multiset` object.

```
iterator insert(
    const value_type& value);

iterator insert(
    const_iterator _where,
    const value_type& value);

template<class _Iterator>
void insert(_Iterator first,
    _Iterator last);

template<class V>
iterator insert(
    V&& value);

template<class V>
typename std::enable_if<!std::is_same<const_iterator,
    typename std::remove_reference<V>::type>::value,
    iterator>::type insert(
    const_iterator _where,
    V&& value);
```

## Parameters

*\_Iterator*

The iterator type used for insertion.

*V*

The type of the value inserted.

*value*

The value to be inserted.

*\_Where*

The starting location to search for an insertion point.

*first*

The beginning of the range to insert.

*last*

The end of the range to insert.

## Return Value

An iterator pointing to the insertion location.

## Remarks

The first member function inserts the element `value` in the controlled sequence, then returns the iterator that designates the inserted element.

The second member function returns `insert(value)`, using `_where` as a starting place within the controlled sequence to search for the insertion point.

The third member function inserts the sequence of element values from the range `[ first, last )`.

The last two member functions behave the same as the first two, except that `value` is used to construct the inserted value.

## key\_eq

The stored equality comparison function object.

```
key_equal key_eq() const;
```

### Return Value

The stored equality comparison function object.

## load\_factor

Computes and returns the current load factor of the container. The load factor is the number of elements in the container divided by the number of buckets.

```
float load_factor() const;
```

### Return Value

The load factor for the container.

## max\_load\_factor

Gets or sets the maximum load factor of the container. The maximum load factor is the largest number of elements than can be in any bucket before the container grows its internal table.

```
float max_load_factor() const;  
  
void max_load_factor(float _Newmax);
```

### Parameters

`_Newmax`

### Return Value

The first member function returns the stored maximum load factor. The second member function does not return a value but throws an [out\\_of\\_range](#) exception if the supplied load factor is invalid..

## max\_size

Returns the maximum size of the concurrent container, determined by the allocator. This method is concurrency safe.

```
size_type max_size() const;
```

### Return Value

The maximum number of elements that can be inserted into this concurrent container.

### Remarks

This upper bound value may actually be higher than what the container can actually hold.

## operator=

Assigns the contents of another `concurrent_unordered_multiset` object to this one. This method is not concurrency-safe.

```
concurrent_unordered_multiset& operator= (const concurrent_unordered_multiset& _Uset);  
  
concurrent_unordered_multiset& operator= (concurrent_unordered_multiset&& _Uset);
```

### Parameters

*\_Uset*

The source `concurrent_unordered_multiset` object.

### Return Value

A reference to this `concurrent_unordered_multiset` object.

### Remarks

After erasing any existing elements in a concurrent unordered multiset, `operator=` either copies or moves the contents of `_Uset` into the concurrent unordered multiset.

## rehash

Rebuilds the hash table.

```
void rehash(size_type _Buckets);
```

### Parameters

*\_Buckets*

The desired number of buckets.

### Remarks

The member function alters the number of buckets to be at least `_Buckets` and rebuilds the hash table as needed. The number of buckets must be a power of 2. If not a power of 2, it will be rounded up to the next largest power of 2.

It throws an `out_of_range` exception if the number of buckets is invalid (either 0 or greater than the maximum number of buckets).

## size

Returns the number of elements in this concurrent container. This method is concurrency safe.

```
size_type size() const;
```

### Return Value

The number of items in the container.

### Remarks

In the presence of concurrent inserts, the number of elements in the concurrent container may change immediately after calling this function, before the return value is even read.

## swap

Swaps the contents of two `concurrent_unordered_multiset` objects. This method is not concurrency-safe.

```
void swap(concurrent_unordered_multiset& _Uset);
```

#### Parameters

*\_Uset*

The `concurrent_unordered_multiset` object to swap with.

## unsafe\_begin

Returns an iterator to the first element in this container for a specific bucket.

```
local_iterator unsafe_begin(size_type _Bucket);  
  
const_local_iterator unsafe_begin(size_type _Bucket) const;
```

#### Parameters

*\_Bucket*

The bucket index.

#### Return Value

An iterator pointing to the beginning of the bucket.

## unsafe\_bucket

Returns the bucket index that a specific key maps to in this container.

```
size_type unsafe_bucket(const key_type& KVal) const;
```

#### Parameters

*KVal*

The element key being searched for.

#### Return Value

The bucket index for the key in this container.

## unsafe\_bucket\_count

Returns the current number of buckets in this container.

```
size_type unsafe_bucket_count() const;
```

#### Return Value

The current number of buckets in this container.

## unsafe\_bucket\_size

Returns the number of items in a specific bucket of this container.

```
size_type unsafe_bucket_size(size_type _Bucket);
```

## Parameters

*\_Bucket*

The bucket to search for.

## Return Value

The current number of buckets in this container.

# unsafe\_cbegin

Returns an iterator to the first element in this container for a specific bucket.

```
const_local_iterator unsafe_cbegin(size_type _Bucket) const;
```

## Parameters

*\_Bucket*

The bucket index.

## Return Value

An iterator pointing to the beginning of the bucket.

# unsafe\_cend

Returns an iterator to the location succeeding the last element in a specific bucket.

```
const_local_iterator unsafe_cend(size_type _Bucket) const;
```

## Parameters

*\_Bucket*

The bucket index.

## Return Value

An iterator pointing to the beginning of the bucket.

# unsafe\_end

Returns an iterator to the last element in this container for a specific bucket.

```
local_iterator unsafe_end(size_type _Bucket);  
const_local_iterator unsafe_end(size_type _Bucket) const;
```

## Parameters

*\_Bucket*

The bucket index.

## Return Value

An iterator pointing to the end of the bucket.

# unsafe\_erase

Removes elements from the `concurrent_unordered_multiset` at specified positions. This method is not concurrency-safe.

```
iterator unsafe_erase(  
    const_iterator _Where);  
  
iterator unsafe_erase(  
    const_iterator first,  
    const_iterator last);  
  
size_type unsafe_erase(  
    const key_type& KVal);
```

### Parameters

*\_Where*

The iterator position to erase from.

*first*

*last*

*KVal*

The key value to erase.

### Return Value

The first two member functions return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists. The third member function returns the number of elements it removes.

### Remarks

The first member function removes the element pointed to by `_Where`. The second member function removes the elements in the range `[_Begin, _End)`.

The third member function removes the elements in the range delimited by `equal_range(KVal)`.

## unsafe\_max\_bucket\_count

Returns the maximum number of buckets in this container.

```
size_type unsafe_max_bucket_count() const;
```

### Return Value

The maximum number of buckets in this container.

## See also

[concurrency Namespace](#)

[Parallel Containers and Objects](#)

# concurrent\_unordered\_set Class

3/4/2019 • 10 minutes to read • [Edit Online](#)

The `concurrent_unordered_set` class is an concurrency-safe container that controls a varying-length sequence of elements of type `K`. The sequence is represented in a way that enables concurrency-safe append, element access, iterator access and iterator traversal operations.

## Syntax

```
template <typename K,  
    typename _Hasher = std::hash<K>,  
    typename key_equality = std::equal_to<K>,  
    typename _Allocator_type = std::allocator<K>  
>,  
    typename key_equality = std::equal_to<K>,  
    typename _Allocator_type = std::allocator<K>> class concurrent_unordered_set : public  
    details::_Concurrent_hash<details::_Concurrent_unordered_set_traits<K,  
        details::_Hash_compare<K,  
        _Hasher,  
        key_equality>,  
        _Allocator_type,  
        false>>;
```

### Parameters

`K`

The key type.

`_Hasher`

The hash function object type. This argument is optional and the default value is `std::hash<K>`.

`key_equality`

The equality comparison function object type. This argument is optional and the default value is `std::equal_to<K>`.

`_Allocator_type`

The type that represents the stored allocator object that encapsulates details about the allocation and deallocation of memory for the concurrent unordered set. This argument is optional and the default value is `std::allocator<K>`.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>allocator_type</code>	The type of an allocator for managing storage.
<code>const_iterator</code>	The type of a constant iterator for the controlled sequence.
<code>const_local_iterator</code>	The type of a constant bucket iterator for the controlled sequence.
<code>const_pointer</code>	The type of a constant pointer to an element.

NAME	DESCRIPTION
<code>const_reference</code>	The type of a constant reference to an element.
<code>difference_type</code>	The type of a signed distance between two elements.
<code>hasher</code>	The type of the hash function.
<code>iterator</code>	The type of an iterator for the controlled sequence.
<code>key_equal</code>	The type of the comparison function.
<code>key_type</code>	The type of an ordering key.
<code>local_iterator</code>	The type of a bucket iterator for the controlled sequence.
<code>pointer</code>	The type of a pointer to an element.
<code>reference</code>	The type of a reference to an element.
<code>size_type</code>	The type of an unsigned distance between two elements.
<code>value_type</code>	The type of an element.

## Public Constructors

NAME	DESCRIPTION
<a href="#">concurrent_unordered_set</a>	Overloaded. Constructs a concurrent unordered set.

## Public Methods

NAME	DESCRIPTION
<a href="#">hash_function</a>	Returns the stored hash function object.
<a href="#">insert</a>	Overloaded. Adds elements to the <code>concurrent_unordered_set</code> object.
<a href="#">key_eq</a>	Returns the stored equality comparison function object.
<a href="#">swap</a>	Swaps the contents of two <code>concurrent_unordered_set</code> objects. This method is not concurrency-safe.
<a href="#">unsafe_erase</a>	Overloaded. Removes elements from the <code>concurrent_unordered_set</code> at specified positions. This method is not concurrency-safe.

## Public Operators

NAME	DESCRIPTION
<code>operator=</code>	Overloaded. Assigns the contents of another <code>concurrent_unordered_set</code> object to this one. This method is not concurrency-safe.

## Remarks

For detailed information on the `concurrent_unordered_set` class, see [Parallel Containers and Objects](#).

## Inheritance Hierarchy

`_Traits`

`_Concurrent_hash`

`concurrent_unordered_set`

## Requirements

**Header:** `concurrent_unordered_set.h`

**Namespace:** `concurrency`

## `begin`

Returns an iterator pointing to the first element in the concurrent container. This method is concurrency safe.

```
iterator begin();  
const_iterator begin() const;
```

### Return Value

An iterator to the first element in the concurrent container.

## `cbegin`

Returns a const iterator pointing to the first element in the concurrent container. This method is concurrency safe.

```
const_iterator cbegin() const;
```

### Return Value

A const iterator to the first element in the concurrent container.

## `cend`

Returns a const iterator pointing to the location succeeding the last element in the concurrent container. This method is concurrency safe.

```
const_iterator cend() const;
```

### Return Value

A const iterator to the location succeeding the last element in the concurrent container.

## clear

Erases all the elements in the concurrent container. This function is not concurrency safe.

```
void clear();
```

## concurrent\_unordered\_set

Constructs a concurrent unordered set.

```
explicit concurrent_unordered_set(
    size_type _Number_of_buckets = 8,
    const hasher& _Hasher = hasher(),
    const key_equal& key_equality = key_equal(),
    const allocator_type& _Allocator = allocator_type());

concurrent_unordered_set(
    const allocator_type& _Allocator);

template <typename _Iterator>
concurrent_unordered_set(_Iterator first,
    _Iterator last,
    size_type _Number_of_buckets = 8,
    const hasher& _Hasher = hasher(),
    const key_equal& key_equality = key_equal(),
    const allocator_type& _Allocator = allocator_type());

concurrent_unordered_set(
    const concurrent_unordered_set& _Uset);

concurrent_unordered_set(
    const concurrent_unordered_set& _Uset,
    const allocator_type& _Allocator);

concurrent_unordered_set(
    concurrent_unordered_set&& _Uset);
```

### Parameters

#### *\_Iterator*

The type of the input iterator.

#### *\_Number\_of\_buckets*

The initial number of buckets for this unordered set.

#### *\_Hasher*

The hash function for this unordered set.

#### *key\_equality*

The equality comparison function for this unordered set.

#### *\_Allocator*

The allocator for this unordered set.

#### *first*

#### *last*

#### *\_Uset*

The source `concurrent_unordered_set` object to copy or move elements from.

## Remarks

All constructors store an allocator object `_Allocator` and initialize the unordered set.

The first constructor specifies an empty initial set and explicitly specifies the number of buckets, hash function, equality function and allocator type to be used.

The second constructor specifies an allocator for the unordered set.

The third constructor specifies values supplied by the iterator range [ `_Begin` , `_End` ).

The fourth and fifth constructors specify a copy of the concurrent unordered set `_Uset` .

The last constructor specifies a move of the concurrent unordered set `_Uset` .

## count

Counts the number of elements matching a specified key. This function is concurrency safe.

```
size_type count(const key_type& KVal) const;
```

### Parameters

*KVal*

The key to search for.

### Return Value

The number of times number of times the key appears in the container.

## empty

Tests whether no elements are present. This method is concurrency safe.

```
bool empty() const;
```

### Return Value

**true** if the concurrent container is empty, **false** otherwise.

### Remarks

In the presence of concurrent inserts, whether or not the concurrent container is empty may change immediately after calling this function, before the return value is even read.

## end

Returns an iterator pointing to the location succeeding the last element in the concurrent container. This method is concurrency safe.

```
iterator end();  
  
const_iterator end() const;
```

### Return Value

An iterator to the location succeeding the last element in the concurrent container.

## equal\_range

Finds a range that matches a specified key. This function is concurrency safe.

```
std::pair<iterator,
iterator> equal_range(
    const key_type& KVal);

std::pair<const_iterator,
const_iterator> equal_range(
    const key_type& KVal) const;
```

### Parameters

*KVal*

The key value to search for.

### Return Value

A [pair](#) where the first element is an iterator to the beginning and the second element is an iterator to the end of the range.

### Remarks

It is possible for concurrent inserts to cause additional keys to be inserted after the begin iterator and before the end iterator.

## find

Finds an element that matches a specified key. This function is concurrency safe.

```
iterator find(const key_type& KVal);

const_iterator find(const key_type& KVal) const;
```

### Parameters

*KVal*

The key value to search for.

### Return Value

An iterator pointing to the location of the first element that matched the key provided, or the iterator `end()` if no such element exists.

## get\_allocator

Returns the stored allocator object for this concurrent container. This method is concurrency safe.

```
allocator_type get_allocator() const;
```

### Return Value

The stored allocator object for this concurrent container.

## hash\_function

Returns the stored hash function object.

```
hasher hash_function() const;
```

## Return Value

The stored hash function object.

# insert

Adds elements to the `concurrent_unordered_set` object.

```
std::pair<iterator,
bool> insert(
    const value_type& value);

iterator insert(
    const_iterator _where,
    const value_type& value);

template<class _Iterator>
void insert(_Iterator first,
    _Iterator last);

template<class V>
std::pair<iterator,
bool> insert(
    V&& value);

template<class V>
typename std::enable_if<!std::is_same<const_iterator,
    typename std::remove_reference<V>::type>::value,
    iterator>::type insert(
    const_iterator _where,
    V&& value);
```

## Parameters

*\_Iterator*

The iterator type used for insertion.

*V*

The type of the value inserted into the set.

*value*

The value to be inserted.

*\_Where*

The starting location to search for an insertion point.

*first*

The beginning of the range to insert.

*last*

The end of the range to insert.

## Return Value

A pair that contains an iterator and a boolean value. See the Remarks section for more details.

## Remarks

The first member function determines whether an element *X* exists in the sequence whose key has equivalent ordering to that of `value`. If not, it creates such an element *X* and initializes it with `value`. The function then determines the iterator `where` that designates *X*. If an insertion occurred, the function returns

```
std::pair(where, true)
```

. Otherwise, it returns `std::pair(where, false)`.

The second member function returns `insert( value )`, using `_where` as a starting place within the controlled sequence to search for the insertion point.

The third member function inserts the sequence of element values from the range [ `first`, `last` ).

The last two member functions behave the same as the first two, except that `value` is used to construct the inserted value.

## key\_eq

Returns the stored equality comparison function object.

```
key_equal key_eq() const;
```

### Return Value

The stored equality comparison function object.

## load\_factor

Computes and returns the current load factor of the container. The load factor is the number of elements in the container divided by the number of buckets.

```
float load_factor() const;
```

### Return Value

The load factor for the container.

## max\_load\_factor

Gets or sets the maximum load factor of the container. The maximum load factor is the largest number of elements than can be in any bucket before the container grows its internal table.

```
float max_load_factor() const;  
void max_load_factor(float _Newmax);
```

### Parameters

`_Newmax`

### Return Value

The first member function returns the stored maximum load factor. The second member function does not return a value but throws an [out\\_of\\_range](#) exception if the supplied load factor is invalid..

## max\_size

Returns the maximum size of the concurrent container, determined by the allocator. This method is concurrency safe.

```
size_type max_size() const;
```

### Return Value

The maximum number of elements that can be inserted into this concurrent container.

### Remarks

This upper bound value may actually be higher than what the container can actually hold.

## operator=

Assigns the contents of another `concurrent_unordered_set` object to this one. This method is not concurrency-safe.

```
concurrent_unordered_set& operator= (const concurrent_unordered_set& _Uset);  
  
concurrent_unordered_set& operator= (concurrent_unordered_set&& _Uset);
```

### Parameters

*\_Uset*

The source `concurrent_unordered_set` object.

### Return Value

A reference to this `concurrent_unordered_set` object.

### Remarks

After erasing any existing elements in a concurrent unordered set, `operator=` either copies or moves the contents of `_Uset` into the concurrent unordered set.

## rehash

Rebuilds the hash table.

```
void rehash(size_type _Buckets);
```

### Parameters

*\_Buckets*

The desired number of buckets.

### Remarks

The member function alters the number of buckets to be at least `_Buckets` and rebuilds the hash table as needed. The number of buckets must be a power of 2. If not a power of 2, it will be rounded up to the next largest power of 2.

It throws an [out\\_of\\_range](#) exception if the number of buckets is invalid (either 0 or greater than the maximum number of buckets).

## size

Returns the number of elements in this concurrent container. This method is concurrency safe.

```
size_type size() const;
```

### Return Value

The number of items in the container.

### Remarks

In the presence of concurrent inserts, the number of elements in the concurrent container may change immediately after calling this function, before the return value is even read.

## swap

Swaps the contents of two `concurrent_unordered_set` objects. This method is not concurrency-safe.

```
void swap(concurrent_unordered_set& _Uset);
```

### Parameters

*\_Uset*

The `concurrent_unordered_set` object to swap with.

## unsafe\_begin

Returns an iterator to the first element in this container for a specific bucket.

```
local_iterator unsafe_begin(size_type _Bucket);  
  
const_local_iterator unsafe_begin(size_type _Bucket) const;
```

### Parameters

*\_Bucket*

The bucket index.

### Return Value

An iterator pointing to the beginning of the bucket.

## unsafe\_bucket

Returns the bucket index that a specific key maps to in this container.

```
size_type unsafe_bucket(const key_type& KVal) const;
```

### Parameters

*KVal*

The element key being searched for.

### Return Value

The bucket index for the key in this container.

## unsafe\_bucket\_count

Returns the current number of buckets in this container.

```
size_type unsafe_bucket_count() const;
```

### Return Value

The current number of buckets in this container.

## unsafe\_bucket\_size

Returns the number of items in a specific bucket of this container.

```
size_type unsafe_bucket_size(size_type _Bucket);
```

### Parameters

*\_Bucket*

The bucket to search for.

### Return Value

The current number of buckets in this container.

## unsafe\_cbegin

Returns an iterator to the first element in this container for a specific bucket.

```
const_local_iterator unsafe_cbegin(size_type _Bucket) const;
```

### Parameters

*\_Bucket*

The bucket index.

### Return Value

An iterator pointing to the beginning of the bucket.

## unsafe\_cend

Returns an iterator to the location succeeding the last element in a specific bucket.

```
const_local_iterator unsafe_cend(size_type _Bucket) const;
```

### Parameters

*\_Bucket*

The bucket index.

### Return Value

An iterator pointing to the beginning of the bucket.

## unsafe\_end

Returns an iterator to the last element in this container for a specific bucket.

```
local_iterator unsafe_end(size_type _Bucket);  
const_local_iterator unsafe_end(size_type _Bucket) const;
```

### Parameters

*\_Bucket*

The bucket index.

### Return Value

An iterator pointing to the end of the bucket.

## unsafe\_erase

Removes elements from the `concurrent_unordered_set` at specified positions. This method is not concurrency-safe.

```
iterator unsafe_erase(
    const_iterator _where);

size_type unsafe_erase(
    const key_type& KVal);

iterator unsafe_erase(
    const_iterator first,
    const_iterator last);
```

### Parameters

*\_Where*

The iterator position to erase from.

*KVal*

The key value to erase.

*first*

*last*

Iterators.

### Return Value

The first two member functions return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists. The third member function returns the number of elements it removes.

### Remarks

The first member function removes the element pointed to by `_where`. The second member function removes the elements in the range [ `_Begin`, `_End` ).

The third member function removes the elements in the range delimited by `equal_range(KVal)`.

## unsafe\_max\_bucket\_count

Returns the maximum number of buckets in this container.

```
size_type unsafe_max_bucket_count() const;
```

### Return Value

The maximum number of buckets in this container.

## See also

[concurrency Namespace](#)

[Parallel Containers and Objects](#)

# concurrent\_vector Class

3/4/2019 • 14 minutes to read • [Edit Online](#)

The `concurrent_vector` class is a sequence container class that allows random access to any element. It enables concurrency-safe append, element access, iterator access, and iterator traversal operations.

## Syntax

```
template<typename T, class _Ax>
class concurrent_vector: protected details::_Allocator_base<T,
    _Ax>,
private details::_Concurrent_vector_base_v4;
```

### Parameters

*T*

The data type of the elements to be stored in the vector.

*\_Ax*

The type that represents the stored allocator object that encapsulates details about the allocation and deallocation of memory for the concurrent vector. This argument is optional and the default value is

`allocator<T>`.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>allocator_type</code>	A type that represents the allocator class for the concurrent vector.
<code>const_iterator</code>	A type that provides a random-access iterator that can read a <code>const</code> element in a concurrent vector.
<code>const_pointer</code>	A type that provides a pointer to a <code>const</code> element in a concurrent vector.
<code>const_reference</code>	A type that provides a reference to a <code>const</code> element stored in a concurrent vector for reading and performing <code>const</code> operations.
<code>const_reverse_iterator</code>	A type that provides a random-access iterator that can read any <code>const</code> element in the concurrent vector.
<code>difference_type</code>	A type that provides the signed distance between two elements in a concurrent vector.
<code>iterator</code>	A type that provides a random-access iterator that can read any element in a concurrent vector. Modification of an element using the iterator is not concurrency-safe.

NAME	DESCRIPTION
<code>pointer</code>	A type that provides a pointer to an element in a concurrent vector.
<code>reference</code>	A type that provides a reference to an element stored in a concurrent vector.
<code>reverse_iterator</code>	A type that provides a random-access iterator that can read any element in a reversed concurrent vector. Modification of an element using the iterator is not concurrency-safe.
<code>size_type</code>	A type that counts the number of elements in a concurrent vector.
<code>value_type</code>	A type that represents the data type stored in a concurrent vector.

### Public Constructors

NAME	DESCRIPTION
<code>concurrent_vector</code>	Overloaded. Constructs a concurrent vector.
<code>~concurrent_vector</code> Destructor	Erases all elements and destroys this concurrent vector.

### Public Methods

NAME	DESCRIPTION
<code>assign</code>	Overloaded. Erases the elements of the concurrent vector and assigns to it either <code>_N</code> copies of <code>_Item</code> , or values specified by the iterator range [ <code>_Begin</code> , <code>_End</code> ). This method is not concurrency-safe.
<code>at</code>	Overloaded. Provides access to the element at the given index in the concurrent vector. This method is concurrency-safe for read operations, and also while growing the vector, as long as you have ensured that the value <code>_Index</code> is less than the size of the concurrent vector.
<code>back</code>	Overloaded. Returns a reference or a <code>const</code> reference to the last element in the concurrent vector. If the concurrent vector is empty, the return value is undefined. This method is concurrency-safe.
<code>begin</code>	Overloaded. Returns an iterator of type <code>iterator</code> or <code>const_iterator</code> to the beginning of the concurrent vector. This method is concurrency-safe.
<code>capacity</code>	Returns the maximum size to which the concurrent vector can grow without having to allocate more memory. This method is concurrency-safe.

NAME	DESCRIPTION
<code>cbegin</code>	Returns an iterator of type <code>const_iterator</code> to the beginning of the concurrent vector. This method is concurrency-safe.
<code>cend</code>	Returns an iterator of type <code>const_iterator</code> to the end of the concurrent vector. This method is concurrency-safe.
<code>clear</code>	Erases all elements in the concurrent vector. This method is not concurrency-safe.
<code>crbegin</code>	Returns an iterator of type <code>const_reverse_iterator</code> to the beginning of the concurrent vector. This method is concurrency-safe.
<code>crend</code>	Returns an iterator of type <code>const_reverse_iterator</code> to the end of the concurrent vector. This method is concurrency-safe.
<code>empty</code>	Tests if the concurrent vector is empty at the time this method is called. This method is concurrency-safe.
<code>end</code>	Overloaded. Returns an iterator of type <code>iterator</code> or <code>const_iterator</code> to the end of the concurrent vector. This method is concurrency-safe.
<code>front</code>	Overloaded. Returns a reference or a <code>const</code> reference to the first element in the concurrent vector. If the concurrent vector is empty, the return value is undefined. This method is concurrency-safe.
<code>get_allocator</code>	Returns a copy of the allocator used to construct the concurrent vector. This method is concurrency-safe.
<code>grow_by</code>	Overloaded. Grows this concurrent vector by <code>_Delta</code> elements. This method is concurrency-safe.
<code>grow_to_at_least</code>	Grows this concurrent vector until it has at least <code>_N</code> elements. This method is concurrency-safe.
<code>max_size</code>	Returns the maximum number of elements the concurrent vector can hold. This method is concurrency-safe.
<code>push_back</code>	Overloaded. Appends the given item to the end of the concurrent vector. This method is concurrency-safe.
<code>rbegin</code>	Overloaded. Returns an iterator of type <code>reverse_iterator</code> or <code>const_reverse_iterator</code> to the beginning of the concurrent vector. This method is concurrency-safe.
<code>rend</code>	Overloaded. Returns an iterator of type <code>reverse_iterator</code> or <code>const_reverse_iterator</code> to the end of the concurrent vector. This method is concurrency-safe.

NAME	DESCRIPTION
<a href="#">reserve</a>	Allocates enough space to grow the concurrent vector to size <code>_N</code> without having to allocate more memory later. This method is not concurrency-safe.
<a href="#">resize</a>	Overloaded. Changes the size of the concurrent vector to the requested size, deleting or adding elements as necessary. This method is not concurrency-safe.
<a href="#">shrink_to_fit</a>	Compacts the internal representation of the concurrent vector to reduce fragmentation and optimize memory usage. This method is not concurrency-safe.
<a href="#">size</a>	Returns the number of elements in the concurrent vector. This method is concurrency-safe.
<a href="#">swap</a>	Swaps the contents of two concurrent vectors. This method is not concurrency-safe.

## Public Operators

NAME	DESCRIPTION
<a href="#">operator[]</a>	Overloaded. Provides access to the element at the given index in the concurrent vector. This method is concurrency-safe for read operations, and also while growing the vector, as long as the you have ensured that the value <code>_Index</code> is less than the size of the concurrent vector.
<a href="#">operator=</a>	Overloaded. Assigns the contents of another <code>concurrent_vector</code> object to this one. This method is not concurrency-safe.

## Remarks

For detailed information on the `concurrent_vector` class, see [Parallel Containers and Objects](#).

## Inheritance Hierarchy

`_Concurrent_vector_base_v4`

`_Allocator_base`

`concurrent_vector`

## Requirements

**Header:** `concurrent_vector.h`

**Namespace:** `concurrency`

## assign

Erases the elements of the concurrent vector and assigns to it either `_N` copies of `_Item`, or values specified by the iterator range [ `_Begin`, `_End` ). This method is not concurrency-safe.

```
void assign(
    size_type _N,
    const_reference _Item);

template<class _InputIterator>
void assign(_InputIterator _Begin,
    _InputIterator _End);
```

### Parameters

*\_InputIterator*

The type of the specified iterator.

*\_N*

The number of items to copy into the concurrent vector.

*\_Item*

Reference to a value used to fill the concurrent vector.

*\_Begin*

An iterator to the first element of the source range.

*\_End*

An iterator to one past the last element of the source range.

### Remarks

`assign` is not concurrency-safe. You must ensure that no other threads are invoking methods on the concurrent vector when you call this method.

## at

Provides access to the element at the given index in the concurrent vector. This method is concurrency-safe for read operations, and also while growing the vector, as long as you have ensured that the value `_Index` is less than the size of the concurrent vector.

```
reference at(size_type _Index);

const_reference at(size_type _Index) const;
```

### Parameters

*\_Index*

The index of the element to be retrieved.

### Return Value

A reference to the item at the given index.

### Remarks

The version of the function `at` that returns a non-`const` reference cannot be used to concurrently write to the element from different threads. A different synchronization object should be used to synchronize concurrent read and write operations to the same data element.

The method throws `out_of_range` if `_Index` is greater than or equal to the size of the concurrent vector, and `range_error` if the index is for a broken portion of the vector. For details on how a vector can become broken, see [Parallel Containers and Objects](#).

## back

Returns a reference or a `const` reference to the last element in the concurrent vector. If the concurrent vector is empty, the return value is undefined. This method is concurrency-safe.

```
reference back();  
  
const_reference back() const;
```

### Return Value

A reference or a `const` reference to the last element in the concurrent vector.

## begin

Returns an iterator of type `iterator` or `const_iterator` to the beginning of the concurrent vector. This method is concurrency-safe.

```
iterator begin();  
  
const_iterator begin() const;
```

### Return Value

An iterator of type `iterator` or `const_iterator` to the beginning of the concurrent vector.

## capacity

Returns the maximum size to which the concurrent vector can grow without having to allocate more memory. This method is concurrency-safe.

```
size_type capacity() const;
```

### Return Value

The maximum size to which the concurrent vector can grow without having to allocate more memory.

### Remarks

Unlike a C++ Standard Library `vector`, a `concurrent_vector` object does not move existing elements if it allocates more memory.

## cbegin

Returns an iterator of type `const_iterator` to the beginning of the concurrent vector. This method is concurrency-safe.

```
const_iterator cbegin() const;
```

### Return Value

An iterator of type `const_iterator` to the beginning of the concurrent vector.

## cend

Returns an iterator of type `const_iterator` to the end of the concurrent vector. This method is concurrency-safe.

```
const_iterator cend() const;
```

### Return Value

An iterator of type `const_iterator` to the end of the concurrent vector.

## clear

Erases all elements in the concurrent vector. This method is not concurrency-safe.

```
void clear();
```

### Remarks

`clear` is not concurrency-safe. You must ensure that no other threads are invoking methods on the concurrent vector when you call this method. `clear` does not free internal arrays. To free internal arrays, call the function `shrink_to_fit` after `clear`.

## concurrent\_vector

Constructs a concurrent vector.

```
explicit concurrent_vector(
    const allocator_type& _Al = allocator_type());

concurrent_vector(
    const concurrent_vector& _Vector);

template<class M>
concurrent_vector(
    const concurrent_vector<T,
    M>& _Vector,
    const allocator_type& _Al = allocator_type());

concurrent_vector(
    concurrent_vector&& _Vector);

explicit concurrent_vector(
    size_type _N);

concurrent_vector(
    size_type _N,
    const_reference _Item,
    const allocator_type& _Al = allocator_type());

template<class _InputIterator>
concurrent_vector(_InputIterator _Begin,
    _InputIterator _End,
    const allocator_type& _Al = allocator_type());
```

### Parameters

*M*

The allocator type of the source vector.

*\_InputIterator*

The type of the input iterator.

*\_Al*

The allocator class to use with this object.

*\_Vector*

The source `concurrent_vector` object to copy or move elements from.

*\_N*

The initial capacity of the `concurrent_vector` object.

*\_Item*

The value of elements in the constructed object.

*\_Begin*

Position of the first element in the range of elements to be copied.

*\_End*

Position of the first element beyond the range of elements to be copied.

### Remarks

All constructors store an allocator object `_A1` and initialize the vector.

The first constructor specify an empty initial vector and explicitly specifies the allocator type. to be used.

The second and third constructors specify a copy of the concurrent vector `_Vector`.

The fourth constructor specifies a move of the concurrent vector `_Vector`.

The fifth constructor specifies a repetition of a specified number ( `_N` ) of elements of the default value for class `T`.

The sixth constructor specifies a repetition of ( `_N` ) elements of value `_Item`.

The last constructor specifies values supplied by the iterator range [ `_Begin` , `_End` ).

## ~concurrent\_vector

Erases all elements and destroys this concurrent vector.

```
~concurrent_vector();
```

## crbegin

Returns an iterator of type `const_reverse_iterator` to the beginning of the concurrent vector. This method is concurrency-safe.

```
const_reverse_iterator crbegin() const;
```

### Return Value

An iterator of type `const_reverse_iterator` to the beginning of the concurrent vector.

## crend

Returns an iterator of type `const_reverse_iterator` to the end of the concurrent vector. This method is concurrency-safe.

```
const_reverse_iterator crend() const;
```

### Return Value

An iterator of type `const_reverse_iterator` to the end of the concurrent vector.

## empty

Tests if the concurrent vector is empty at the time this method is called. This method is concurrency-safe.

```
bool empty() const;
```

### Return Value

**true** if the vector was empty at the moment the function was called, **false** otherwise.

## end

Returns an iterator of type `iterator` or `const_iterator` to the end of the concurrent vector. This method is concurrency-safe.

```
iterator end();  
const_iterator end() const;
```

### Return Value

An iterator of type `iterator` or `const_iterator` to the end of the concurrent vector.

## front

Returns a reference or a `const` reference to the first element in the concurrent vector. If the concurrent vector is empty, the return value is undefined. This method is concurrency-safe.

```
reference front();  
const_reference front() const;
```

### Return Value

A reference or a `const` reference to the first element in the concurrent vector.

## get\_allocator

Returns a copy of the allocator used to construct the concurrent vector. This method is concurrency-safe.

```
allocator_type get_allocator() const;
```

### Return Value

A copy of the allocator used to construct the `concurrent_vector` object.

## grow\_by

Grows this concurrent vector by `_Delta` elements. This method is concurrency-safe.

```
iterator grow_by(  
    size_type _Delta);  
  
iterator grow_by(  
    size_type _Delta,  
    const_reference _Item);
```

### Parameters

*\_Delta*

The number of elements to append to the object.

*\_Item*

The value to initialize the new elements with.

### Return Value

An iterator to first item appended.

### Remarks

If `_Item` is not specified, the new elements are default constructed.

## grow\_to\_at\_least

Grows this concurrent vector until it has at least `_N` elements. This method is concurrency-safe.

```
iterator grow_to_at_least(size_type _N);
```

### Parameters

*\_N*

The new minimum size for the `concurrent_vector` object.

### Return Value

An iterator that points to beginning of appended sequence, or to the element at index `_N` if no elements were appended.

## max\_size

Returns the maximum number of elements the concurrent vector can hold. This method is concurrency-safe.

```
size_type max_size() const;
```

### Return Value

The maximum number of elements the `concurrent_vector` object can hold.

## operator=

Assigns the contents of another `concurrent_vector` object to this one. This method is not concurrency-safe.

```

concurrent_vector& operator= (
    const concurrent_vector& _Vector);

template<class M>
concurrent_vector& operator= (
    const concurrent_vector<T, M>& _Vector);

concurrent_vector& operator= (
    concurrent_vector&& _Vector);

```

## Parameters

*M*

The allocator type of the source vector.

*\_Vector*

The source `concurrent_vector` object.

## Return Value

A reference to this `concurrent_vector` object.

## operator[]

Provides access to the element at the given index in the concurrent vector. This method is concurrency-safe for read operations, and also while growing the vector, as long as the you have ensured that the value `_Index` is less than the size of the concurrent vector.

```

reference operator[](size_type _index);

const_reference operator[](size_type _index) const;

```

## Parameters

*\_Index*

The index of the element to be retrieved.

## Return Value

A reference to the item at the given index.

## Remarks

The version of `operator []` that returns a non-`const` reference cannot be used to concurrently write to the element from different threads. A different synchronization object should be used to synchronize concurrent read and write operations to the same data element.

No bounds checking is performed to ensure that `_Index` is a valid index into the concurrent vector.

## push\_back

Appends the given item to the end of the concurrent vector. This method is concurrency-safe.

```

iterator push_back(const_reference _Item);

iterator push_back(T&& _Item);

```

## Parameters

*\_Item*

The value to be appended.

### Return Value

An iterator to item appended.

## rbegin

Returns an iterator of type `reverse_iterator` or `const_reverse_iterator` to the beginning of the concurrent vector. This method is concurrency-safe.

```
reverse_iterator rbegin();  
  
const_reverse_iterator rbegin() const;
```

### Return Value

An iterator of type `reverse_iterator` or `const_reverse_iterator` to the beginning of the concurrent vector.

## rend

Returns an iterator of type `reverse_iterator` or `const_reverse_iterator` to the end of the concurrent vector. This method is concurrency-safe.

```
reverse_iterator rend();  
  
const_reverse_iterator rend() const;
```

### Return Value

An iterator of type `reverse_iterator` or `const_reverse_iterator` to the end of the concurrent vector.

## reserve

Allocates enough space to grow the concurrent vector to size `_N` without having to allocate more memory later. This method is not concurrency-safe.

```
void reserve(size_type _N);
```

### Parameters

`_N`

The number of elements to reserve space for.

### Remarks

`reserve` is not concurrency-safe. You must ensure that no other threads are invoking methods on the concurrent vector when you call this method. The capacity of the concurrent vector after the method returns may be bigger than the requested reservation.

## resize

Changes the size of the concurrent vector to the requested size, deleting or adding elements as necessary. This method is not concurrency-safe.

```
void resize(
    size_type _N);

void resize(
    size_type _N,
    const T& val);
```

### Parameters

*\_N*

The new size of the `concurrent_vector`.

*val*

The value of new elements added to the vector if the new size is larger than the original size. If the value is omitted, the new objects are assigned the default value for their type.

### Remarks

If the size of the container is less than the requested size, elements are added to the vector until it reaches the requested size. If the size of the container is larger than the requested size, the elements closest to the end of the container are deleted until the container reaches the size `_N`. If the present size of the container is the same as the requested size, no action is taken.

`resize` is not concurrency safe. You must ensure that no other threads are invoking methods on the concurrent vector when you call this method.

## shrink\_to\_fit

Compacts the internal representation of the concurrent vector to reduce fragmentation and optimize memory usage. This method is not concurrency-safe.

```
void shrink_to_fit();
```

### Remarks

This method will internally re-allocate memory move elements around, invalidating all the iterators.

`shrink_to_fit` is not concurrency-safe. You must ensure that no other threads are invoking methods on the concurrent vector when you call this function.

## size

Returns the number of elements in the concurrent vector. This method is concurrency-safe.

```
size_type size() const;
```

### Return Value

The number of elements in this `concurrent_vector` object.

### Remarks

The returned size is guaranteed to include all elements appended by calls to the function `push_back`, or grow operations that have completed prior to invoking this method. However, it may also include elements that are allocated but still under construction by concurrent calls to any of the growth methods.

## swap

Swaps the contents of two concurrent vectors. This method is not concurrency-safe.

```
void swap(concurrent_vector& _Vector);
```

### Parameters

*\_Vector*

The `concurrent_vector` object to swap contents with.

## See also

[concurrency Namespace](#)

[Parallel Containers and Objects](#)

# Context Class

3/4/2019 • 8 minutes to read • [Edit Online](#)

Represents an abstraction for an execution context.

## Syntax

```
class Context;
```

## Members

### Protected Constructors

NAME	DESCRIPTION
<a href="#">~Context Destructor</a>	

### Public Methods

NAME	DESCRIPTION
<a href="#">Block</a>	Blocks the current context.
<a href="#">CurrentContext</a>	Returns a pointer to the current context.
<a href="#">GetId</a>	Returns an identifier for the context that is unique within the scheduler to which the context belongs.
<a href="#">GetScheduleGroupId</a>	Returns an identifier for the schedule group that the context is currently working on.
<a href="#">GetVirtualProcessorId</a>	Returns an identifier for the virtual processor that the context is currently executing on.
<a href="#">Id</a>	Returns an identifier for the current context that is unique within the scheduler to which the current context belongs.
<a href="#">IsCurrentTaskCollectionCanceling</a>	Returns an indication of whether the task collection which is currently executing inline on the current context is in the midst of an active cancellation (or will be shortly).
<a href="#">IsSynchronouslyBlocked</a>	Determines whether or not the context is synchronously blocked. A context is considered to be synchronously blocked if it explicitly performed an action which led to blocking.
<a href="#">Oversubscribe</a>	Injects an additional virtual processor into a scheduler for the duration of a block of code when invoked on a context executing on one of the virtual processors in that scheduler.

NAME	DESCRIPTION
<a href="#">ScheduleGroupId</a>	Returns an identifier for the schedule group that the current context is working on.
<a href="#">Unblock</a>	Unblocks the context and causes it to become runnable.
<a href="#">VirtualProcessorId</a>	Returns an identifier for the virtual processor that the current context is executing on.
<a href="#">Yield</a>	Yields execution so that another context can execute. If no other context is available to yield to, the scheduler can yield to another operating system thread.

## Remarks

The Concurrency Runtime scheduler (see [Scheduler](#)) uses execution contexts to execute the work queued to it by your application. A Win32 thread is an example of an execution context on a Windows operating system.

At any time, the concurrency level of a scheduler is equal to the number of virtual processors granted to it by the Resource Manager. A virtual processor is an abstraction for a processing resource and maps to a hardware thread on the underlying system. Only a single scheduler context can execute on a virtual processor at a given time.

The scheduler is cooperative in nature and an executing context can yield its virtual processor to a different context at any time if it wishes to enter a wait state. When its wait is satisfied, it cannot resume until an available virtual processor from the scheduler begins executing it.

## Inheritance Hierarchy

Context

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## Block

Blocks the current context.

```
static void __cdecl Block();
```

### Remarks

This method will result in the process' default scheduler being created and/or attached to the calling context if there is no scheduler currently associated with the calling context.

If the calling context is running on a virtual processor, the virtual processor will find another runnable context to execute or can potentially create a new one.

After the `Block` method has been called or will be called, you must pair it with a call to the [Unblock](#) method from another execution context in order for it to run again. Be aware that there is a critical period between the point where your code publishes its context for another thread to be able to call the `Unblock` method and the point where the actual method call to `Block` is made. During this period, you must not call any method which can in

turn block and unblock for its own reasons (for example, acquiring a lock). Calls to the `Block` and `Unblock` method do not track the reason for the blocking and unblocking. Only one object should have ownership of a `Block - Unblock` pair.

This method can throw a variety of exceptions, including [scheduler\\_resource\\_allocation\\_error](#).

## ~Context

```
virtual ~Context();
```

## CurrentContext

Returns a pointer to the current context.

```
static Context* __cdecl CurrentContext();
```

### Return Value

A pointer to the current context.

### Remarks

This method will result in the process' default scheduler being created and/or attached to the calling context if there is no scheduler currently associated with the calling context.

## GetId

Returns an identifier for the context that is unique within the scheduler to which the context belongs.

```
virtual unsigned int GetId() const = 0;
```

### Return Value

An identifier for the context that is unique within the scheduler to which the context belongs.

## GetScheduleGroupId

Returns an identifier for the schedule group that the context is currently working on.

```
virtual unsigned int GetScheduleGroupId() const = 0;
```

### Return Value

An identifier for the schedule group the context is currently working on.

### Remarks

The return value from this method is an instantaneous sampling of the schedule group that the context is executing on. If this method is called on a context other than the current context, the value can be stale the moment it is returned and cannot be relied upon. Typically, this method is used for debugging or tracing purposes only.

## GetVirtualProcessorId

Returns an identifier for the virtual processor that the context is currently executing on.

```
virtual unsigned int GetVirtualProcessorId() const = 0;
```

### Return Value

If the context is currently executing on a virtual processor, an identifier for the virtual processor that the context is currently executing on; otherwise, the value `-1`.

### Remarks

The return value from this method is an instantaneous sampling of the virtual processor that the context is executing on. This value can be stale the moment it is returned and cannot be relied upon. Typically, this method is used for debugging or tracing purposes only.

## Id

Returns an identifier for the current context that is unique within the scheduler to which the current context belongs.

```
static unsigned int __cdecl Id();
```

### Return Value

If the current context is attached to a scheduler, an identifier for the current context that is unique within the scheduler to which the current context belongs; otherwise, the value `-1`.

## IsCurrentTaskCollectionCanceling

Returns an indication of whether the task collection which is currently executing inline on the current context is in the midst of an active cancellation (or will be shortly).

```
static bool __cdecl IsCurrentTaskCollectionCanceling();
```

### Return Value

If a scheduler is attached to the calling context and a task group is executing a task inline on that context, an indication of whether that task group is in the midst of an active cancellation (or will be shortly); otherwise, the value `false`.

## IsSynchronouslyBlocked

Determines whether or not the context is synchronously blocked. A context is considered to be synchronously blocked if it explicitly performed an action which led to blocking.

```
virtual bool IsSynchronouslyBlocked() const = 0;
```

### Return Value

Whether the context is synchronously blocked.

### Remarks

A context is considered to be synchronously blocked if it explicitly performed an action which led to blocking. On the thread scheduler, this would indicate a direct call to the `Context::Block` method or a synchronization object which was built using the `Context::Block` method.

The return value from this method is an instantaneous sample of whether the context is synchronously blocked.

This value may be stale the moment it is returned and can only be used under very specific circumstances.

## operator delete

A `Context` object is destroyed internally by the runtime. It can not be explicitly deleted.

```
void operator delete(void* _PObject);
```

### Parameters

*\_PObject*

A pointer to the object to be deleted.

## Oversubscribe

Injects an additional virtual processor into a scheduler for the duration of a block of code when invoked on a context executing on one of the virtual processors in that scheduler.

```
static void __cdecl Oversubscribe(bool _BeginOversubscription);
```

### Parameters

*\_BeginOversubscription*

If **true**, an indication that an extra virtual processor should be added for the duration of the oversubscription. If **false**, an indication that the oversubscription should end and the previously added virtual processor should be removed.

## ScheduleGroupId

Returns an identifier for the schedule group that the current context is working on.

```
static unsigned int __cdecl ScheduleGroupId();
```

### Return Value

If the current context is attached to a scheduler and working on a schedule group, an identifier for the scheduler group that the current context is working on; otherwise, the value `-1`.

## Unblock

Unblocks the context and causes it to become runnable.

```
virtual void Unblock() = 0;
```

### Remarks

It is perfectly legal for a call to the `Unblock` method to come before a corresponding call to the `Block` method. As long as calls to the `Block` and `Unblock` methods are properly paired, the runtime properly handles the natural race of either ordering. An `Unblock` call coming before a `Block` call simply negates the effect of the `Block` call.

There are several exceptions which can be thrown from this method. If a context attempts to call the `Unblock` method on itself, a `context_self_unblock` exception will be thrown. If calls to `Block` and `Unblock` are not properly paired (for example, two calls to `Unblock` are made for a context which is currently running), a `context_unblock_unbalanced` exception will be thrown.

Be aware that there is a critical period between the point where your code publishes its context for another thread to be able to call the `Unlock` method and the point where the actual method call to `Block` is made. During this period, you must not call any method which can in turn block and unblock for its own reasons (for example, acquiring a lock). Calls to the `Block` and `Unlock` method do not track the reason for the blocking and unblocking. Only one object should have ownership of a `Block` and `Unlock` pair.

## VirtualProcessorId

Returns an identifier for the virtual processor that the current context is executing on.

```
static unsigned int __cdecl VirtualProcessorId();
```

### Return Value

If the current context is attached to a scheduler, an identifier for the virtual processor that the current context is executing on; otherwise, the value `-1`.

### Remarks

The return value from this method is an instantaneous sampling of the virtual processor that the current context is executing on. This value can be stale the moment it is returned and cannot be relied upon. Typically, this method is used for debugging or tracing purposes only.

## Yield

Yields execution so that another context can execute. If no other context is available to yield to, the scheduler can yield to another operating system thread.

```
static void __cdecl Yield();
```

### Remarks

This method will result in the process' default scheduler being created and/or attached to the calling context if there is no scheduler currently associated with the calling context.

## YieldExecution

Yields execution so that another context can execute. If no other context is available to yield to, the scheduler can yield to another operating system thread.

```
static void __cdecl YieldExecution();
```

### Remarks

This method will result in the process' default scheduler being created and/or attached to the calling context if there is no scheduler currently associated with the calling context.

This function is new in Visual Studio 2015 and is identical to the [Yield](#) function but does not conflict with the `Yield` macro in `Windows.h`.

## See also

[concurrency Namespace](#)

[Scheduler Class](#)

[Task Scheduler](#)

# context\_self\_unblock Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when the `Unblock` method of a `Context` object is called from the same context. This would indicate an attempt by a given context to unblock itself.

## Syntax

```
class context_self_unblock : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">context_self_unblock</a>	Overloaded. Constructs a <code>context_self_unblock</code> object.

## Inheritance Hierarchy

`exception`

`context_self_unblock`

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## context\_self\_unblock

Constructs a `context_self_unblock` object.

```
explicit _CRTIMP context_self_unblock(_In_z_ const char* _Message) throw();  
context_self_unblock() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

# context\_unblock\_unbalanced Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when calls to the `Block` and `Unblock` methods of a `Context` object are not properly paired.

## Syntax

```
class context_unblock_unbalanced : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<code>context_unblock_unbalanced</code>	Overloaded. Constructs a <code>context_unblock_unbalanced</code> object.

## Remarks

Calls to the `Block` and `Unblock` methods of a `Context` object must always be properly paired. The Concurrency Runtime allows the operations to happen in either order. For example, a call to `Block` can be followed by a call to `Unblock`, or vice-versa. This exception would be thrown if, for instance, two calls to the `Unblock` method were made in a row, on a `Context` object which was not blocked.

## Inheritance Hierarchy

`exception`

`context_unblock_unbalanced`

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## context\_unblock\_unbalanced

Constructs a `context_unblock_unbalanced` object.

```
explicit _CRTIMP context_unblock_unbalanced(_In_z_ const char* _Message) throw();  
context_unblock_unbalanced() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

# critical\_section Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

A non-reentrant mutex which is explicitly aware of the Concurrency Runtime.

## Syntax

```
class critical_section;
```

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>native_handle_type</code>	A reference to a <code>critical_section</code> object.

### Public Classes

NAME	DESCRIPTION
<code>critical_section::scoped_lock</code> Class	An exception safe RAII wrapper for a <code>critical_section</code> object.

### Public Constructors

NAME	DESCRIPTION
<code>critical_section</code>	Constructs a new critical section.
<code>~critical_section</code> Destructor	Destroys a critical section.

### Public Methods

NAME	DESCRIPTION
<code>lock</code>	Acquires this critical section.
<code>native_handle</code>	Returns a platform specific native handle, if one exists.
<code>try_lock</code>	Tries to acquire the lock without blocking.
<code>try_lock_for</code>	Tries to acquire the lock without blocking for a specific number of milliseconds.
<code>unlock</code>	Unlocks the critical section.

## Remarks

For more information, see [Synchronization Data Structures](#).

## Inheritance Hierarchy

```
critical_section
```

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## `critical_section`

Constructs a new critical section.

```
critical_section();
```

## `~critical_section`

Destroys a critical section.

```
~critical_section();
```

### Remarks

It is expected that the lock is no longer held when the destructor runs. Allowing the critical section to destruct with the lock still held results in undefined behavior.

## `lock`

Acquires this critical section.

```
void lock();
```

### Remarks

It is often safer to utilize the [scoped\\_lock](#) construct to acquire and release a `critical_section` object in an exception safe way.

If the lock is already held by the calling context, an [improper\\_lock](#) exception will be thrown.

## `native_handle`

Returns a platform specific native handle, if one exists.

```
native_handle_type native_handle();
```

### Return Value

A reference to the critical section.

### Remarks

A `critical_section` object is not associated with a platform specific native handle for the Windows operating

system. The method simply returns a reference to the object itself.

## critical\_section::scoped\_lock Class

An exception safe RAII wrapper for a `critical_section` object.

```
class scoped_lock;
```

## scoped\_lock::scoped\_lock

Constructs a `scoped_lock` object and acquires the `critical_section` object passed in the `_Critical_section` parameter. If the critical section is held by another thread, this call will block.

```
explicit _CRTIMP scoped_lock(critical_section& _Critical_section);
```

### Parameters

*\_Critical\_section*

The critical section to lock.

## scoped\_lock::~~scoped\_lock

Destroys a `scoped_lock` object and releases the critical section supplied in its constructor.

```
~scoped_lock();
```

## try\_lock

Tries to acquire the lock without blocking.

```
bool try_lock();
```

### Return Value

If the lock was acquired, the value **true**; otherwise, the value **false**.

## try\_lock\_for

Tries to acquire the lock without blocking for a specific number of milliseconds.

```
bool try_lock_for(unsigned int _Timeout);
```

### Parameters

*\_Timeout*

The number of milliseconds to wait before timing out.

### Return Value

If the lock was acquired, the value **true**; otherwise, the value **false**.

## unlock

Unlocks the critical section.

```
void unlock();
```

## See also

[concurrency Namespace](#)

[reader\\_writer\\_lock Class](#)

# CurrentScheduler Class

3/4/2019 • 7 minutes to read • [Edit Online](#)

Represents an abstraction for the current scheduler associated with the calling context.

## Syntax

```
class CurrentScheduler;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">Create</a>	Creates a new scheduler whose behavior is described by the <code>_Policy</code> parameter and attaches it to the calling context. The newly created scheduler will become the current scheduler for the calling context.
<a href="#">CreateScheduleGroup</a>	Overloaded. Creates a new schedule group within the scheduler associated with the calling context. The version that takes the parameter <code>_Placement</code> causes tasks within the newly created schedule group to be biased towards executing at the location specified by that parameter.
<a href="#">Detach</a>	Detaches the current scheduler from the calling context and restores the previously attached scheduler as the current scheduler, if one exists. After this method returns, the calling context is then managed by the scheduler that was previously attached to the context using either the <code>CurrentScheduler::Create</code> or <code>Scheduler::Attach</code> method.
<a href="#">Get</a>	Returns a pointer to the scheduler associated with the calling context, also referred to as the current scheduler.
<a href="#">GetNumberOfVirtualProcessors</a>	Returns the current number of virtual processors for the scheduler associated with the calling context.
<a href="#">GetPolicy</a>	Returns a copy of the policy that the current scheduler was created with.
<a href="#">Id</a>	Returns a unique identifier for the current scheduler.
<a href="#">IsAvailableLocation</a>	Determines whether a given location is available on the current scheduler.

NAME	DESCRIPTION
<a href="#">RegisterShutdownEvent</a>	Causes the Windows event handle passed in the <code>_ShutdownEvent</code> parameter to be signaled when the scheduler associated with the current context shuts down and destroys itself. At the time the event is signaled, all work that had been scheduled to the scheduler is complete. Multiple shutdown events can be registered through this method.
<a href="#">ScheduleTask</a>	Overloaded. Schedules a light-weight task within the scheduler associated with the calling context. The light-weight task will be placed in a schedule group determined by the runtime. The version that takes the parameter <code>_Placement</code> causes the task to be biased towards executing at the specified location.

## Remarks

If there is no scheduler (see [Scheduler](#)) associated with the calling context, many methods within the `CurrentScheduler` class will result in attachment of the process' default scheduler. This may also imply that the process' default scheduler is created during such a call.

## Inheritance Hierarchy

`CurrentScheduler`

## Requirements

**Header:** `concr.t.h`

**Namespace:** `concurrency`

## Create

Creates a new scheduler whose behavior is described by the `_Policy` parameter and attaches it to the calling context. The newly created scheduler will become the current scheduler for the calling context.

```
static void __cdecl Create(const SchedulerPolicy& _Policy);
```

### Parameters

`_Policy`

The scheduler policy that describes the behavior of the newly created scheduler.

### Remarks

The attachment of the scheduler to the calling context implicitly places a reference count on the scheduler.

After a scheduler is created with the `Create` method, you must call the `CurrentScheduler::Detach` method at some point in the future in order to allow the scheduler to shut down.

If this method is called from a context that is already attached to a different scheduler, the existing scheduler is remembered as the previous scheduler, and the newly created scheduler becomes the current scheduler. When you call the `CurrentScheduler::Detach` method at a later point, the previous scheduler is restored as the current scheduler.

This method can throw a variety of exceptions, including [scheduler\\_resource\\_allocation\\_error](#) and

[invalid\\_scheduler\\_policy\\_value](#).

## CreateScheduleGroup

Creates a new schedule group within the scheduler associated with the calling context. The version that takes the parameter `_Placement` causes tasks within the newly created schedule group to be biased towards executing at the location specified by that parameter.

```
static ScheduleGroup* __cdecl CreateScheduleGroup();  
  
static ScheduleGroup* __cdecl CreateScheduleGroup(location& _Placement);
```

### Parameters

*\_Placement*

A reference to a location where the tasks within the schedule group will be biased towards executing at.

### Return Value

A pointer to the newly created schedule group. This `ScheduleGroup` object has an initial reference count placed on it.

### Remarks

This method will result in the process' default scheduler being created and/or attached to the calling context if there is no scheduler currently associated with the calling context.

You must invoke the [Release](#) method on a schedule group when you are done scheduling work to it. The scheduler will destroy the schedule group when all work queued to it has completed.

Note that if you explicitly created this scheduler, you must release all references to schedule groups within it, before you release your reference on the scheduler, by detaching the current context from it.

## Detach

Detaches the current scheduler from the calling context and restores the previously attached scheduler as the current scheduler, if one exists. After this method returns, the calling context is then managed by the scheduler that was previously attached to the context using either the `CurrentScheduler::Create` or `Scheduler::Attach` method.

```
static void __cdecl Detach();
```

### Remarks

The `Detach` method implicitly removes a reference count from the scheduler.

If there is no scheduler attached to the calling context, calling this method will result in a [scheduler\\_not\\_attached](#) exception being thrown.

Calling this method from a context that is internal to and managed by a scheduler, or a context that was attached using a method other than the `Scheduler::Attach` or `CurrentScheduler::Create` methods, will result in an [improper\\_scheduler\\_detach](#) exception being thrown.

## Get

Returns a pointer to the scheduler associated with the calling context, also referred to as the current scheduler.

```
static Scheduler* __cdecl Get();
```

### Return Value

A pointer to the scheduler associated with the calling context (the current scheduler).

### Remarks

This method will result in the process' default scheduler being created and/or attached to the calling context if there is no scheduler currently associated with the calling context. No additional reference is placed on the `Scheduler` object returned by this method.

## GetNumberOfVirtualProcessors

Returns the current number of virtual processors for the scheduler associated with the calling context.

```
static unsigned int __cdecl GetNumberOfVirtualProcessors();
```

### Return Value

If a scheduler is associated with the calling context, the current number of virtual processors for that scheduler; otherwise, the value `-1`.

### Remarks

This method will not result in scheduler attachment if the calling context is not already associated with a scheduler.

The return value from this method is an instantaneous sampling of the number of virtual processors for the scheduler associated with the calling context. This value can be stale the moment it is returned.

## GetPolicy

Returns a copy of the policy that the current scheduler was created with.

```
static SchedulerPolicy __cdecl GetPolicy();
```

### Return Value

A copy of the policy that the current scheduler was created with.

### Remarks

This method will result in the process' default scheduler being created and/or attached to the calling context if there is no scheduler currently associated with the calling context.

## Id

Returns a unique identifier for the current scheduler.

```
static unsigned int __cdecl Id();
```

### Return Value

If a scheduler is associated with the calling context, a unique identifier for that scheduler; otherwise, the value `-1`.

### Remarks

This method will not result in scheduler attachment if the calling context is not already associated with a scheduler.

# IsAvailableLocation

Determines whether a given location is available on the current scheduler.

```
static bool __cdecl IsAvailableLocation(const location& _Placement);
```

## Parameters

*\_Placement*

A reference to the location to query the current scheduler about.

## Return Value

An indication of whether or not the location specified by the `_Placement` argument is available on the current scheduler.

## Remarks

This method will not result in scheduler attachment if the calling context is not already associated with a scheduler.

Note that the return value is an instantaneous sampling of whether the given location is available. In the presence of multiple schedulers, dynamic resource management can add or take away resources from schedulers at any point. Should this happen, the given location can change availability.

# RegisterShutdownEvent

Causes the Windows event handle passed in the `_ShutdownEvent` parameter to be signaled when the scheduler associated with the current context shuts down and destroys itself. At the time the event is signaled, all work that had been scheduled to the scheduler is complete. Multiple shutdown events can be registered through this method.

```
static void __cdecl RegisterShutdownEvent(HANDLE _ShutdownEvent);
```

## Parameters

*\_ShutdownEvent*

A handle to a Windows event object which will be signaled by the runtime when the scheduler associated with the current context shuts down and destroys itself.

## Remarks

If there is no scheduler attached to the calling context, calling this method will result in a [scheduler\\_not\\_attached](#) exception being thrown.

# ScheduleTask

Schedules a light-weight task within the scheduler associated with the calling context. The light-weight task will be placed in a schedule group determined by the runtime. The version that takes the parameter `_Placement` causes the task to be biased towards executing at the specified location.

```
static void __cdecl ScheduleTask(  
    TaskProc _Proc,  
    _Inout_opt_ void* _Data);  
  
static void __cdecl ScheduleTask(  
    TaskProc _Proc,  
    _Inout_opt_ void* _Data,  
    location& _Placement);
```

**Parameters**

*\_Proc*

A pointer to the function to execute to perform the body of the light-weight task.

*\_Data*

A void pointer to the data that will be passed as a parameter to the body of the task.

*\_Placement*

A reference to a location where the light-weight task will be biased towards executing at.

**Remarks**

This method will result in the process' default scheduler being created and/or attached to the calling context if there is no scheduler currently associated with the calling context.

## See also

[concurrency Namespace](#)

[Scheduler Class](#)

[PolicyElementKey](#)

[Task Scheduler](#)

# default\_scheduler\_exists Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when the `Scheduler::SetDefaultSchedulerPolicy` method is called when a default scheduler already exists within the process.

## Syntax

```
class default_scheduler_exists : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">default_scheduler_exists</a>	Overloaded. Constructs a <code>default_scheduler_exists</code> object.

## Inheritance Hierarchy

`exception`

`default_scheduler_exists`

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## default\_scheduler\_exists

Constructs a `default_scheduler_exists` object.

```
explicit _CRTIMP default_scheduler_exists(_In_z_ const char* _Message) throw();  
default_scheduler_exists() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

# DispatchState Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `DispatchState` structure is used to transfer state to the `IExecutionContext::Dispatch` method. It describes the circumstances under which the `Dispatch` method is invoked on an `IExecutionContext` interface.

## Syntax

```
struct DispatchState;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">DispatchState::DispatchState</a>	Constructs a new <code>DispatchState</code> object.

### Public Data Members

NAME	DESCRIPTION
<a href="#">DispatchState::m_dispatchStateSize</a>	Size of this structure, which is used for versioning.
<a href="#">DispatchState::m_flPreviousContextAsynchronouslyBlocked</a>	Tells whether this context has entered the <code>Dispatch</code> method because the previous context asynchronously blocked. This is used only on the UMS scheduling context, and is set to the value <code>0</code> for all other execution contexts.
<a href="#">DispatchState::m_reserved</a>	Bits reserved for future information passing.

## Inheritance Hierarchy

`DispatchState`

## Requirements

**Header:** `concrtrm.h`

**Namespace:** `concurrency`

## DispatchState::DispatchState Constructor

Constructs a new `DispatchState` object.

```
DispatchState();
```

## DispatchState::m\_dispatchStateSize Data Member

Size of this structure, which is used for versioning.

```
unsigned long m_dispatchStateSize;
```

## DispatchState::m\_fIsPreviousContextAsynchronouslyBlocked Data Member

Tells whether this context has entered the `Dispatch` method because the previous context asynchronously blocked. This is used only on the UMS scheduling context, and is set to the value `0` for all other execution contexts.

```
unsigned int m_fIsPreviousContextAsynchronouslyBlocked : 1;
```

## DispatchState::m\_reserved Data Member

Bits reserved for future information passing.

```
unsigned int m_reserved : 31;
```

## See also

[concurrency Namespace](#)

# event Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

A manual reset event which is explicitly aware of the Concurrency Runtime.

## Syntax

```
class event;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">~event Destructor</a>	Destroys an event.

### Public Methods

NAME	DESCRIPTION
<a href="#">reset</a>	Resets the event to a non-signaled state.
<a href="#">set</a>	Signals the event.
<a href="#">wait</a>	Waits for the event to become signaled.
<a href="#">wait_for_multiple</a>	Waits for multiple events to become signaled.

### Public Constants

NAME	DESCRIPTION
<a href="#">timeout_infinite</a>	Value indicating that a wait should never time out.

## Remarks

For more information, see [Synchronization Data Structures](#).

## Inheritance Hierarchy

event

## Requirements

**Header:** conctr.h

**Namespace:** concurrency

## event

Constructs a new event.

```
_CRTIMP event();
```

### Remarks

## ~event

Destroys an event.

```
~event();
```

### Remarks

It is expected that there are no threads waiting on the event when the destructor runs. Allowing the event to destruct with threads still waiting on it results in undefined behavior.

## reset

Resets the event to a non-signaled state.

```
void reset();
```

## set

Signals the event.

```
void set();
```

### Remarks

Signaling the event can cause an arbitrary number of contexts waiting on the event to become runnable.

## timeout\_infinite

Value indicating that a wait should never time out.

```
static const unsigned int timeout_infinite = COOPERATIVE_TIMEOUT_INFINITE;
```

## wait

Waits for the event to become signaled.

```
size_t wait(unsigned int _Timeout = COOPERATIVE_TIMEOUT_INFINITE);
```

### Parameters

*\_Timeout*

Indicates the number of milliseconds before the wait times out. The value `COOPERATIVE_TIMEOUT_INFINITE` signifies that there is no timeout.

## Return Value

If the wait was satisfied, the value `0` is returned; otherwise, the value `COOPERATIVE_WAIT_TIMEOUT` to indicate that the wait timed out without the event becoming signaled.

### IMPORTANT

In a Universal Windows Platform (UWP) app, do not call `wait` on the ASTA thread because this call can block the current thread and can cause the app to become unresponsive.

## wait\_for\_multiple

Waits for multiple events to become signaled.

```
static size_t __cdecl wait_for_multiple(  
    _In_reads_(count) event** _PPEvents,  
    size_t count,  
    bool _FWaitAll,  
    unsigned int _Timeout = COOPERATIVE_TIMEOUT_INFINITE);
```

### Parameters

#### *\_PPEvents*

An array of events to wait on. The number of events within the array is indicated by the `count` parameter.

#### *count*

The count of events within the array supplied in the `_PPEvents` parameter.

#### *\_FWaitAll*

If set to the value **true**, the parameter specifies that all events within the array supplied in the `_PPEvents` parameter must become signaled in order to satisfy the wait. If set to the value **false**, it specifies that any event within the array supplied in the `_PPEvents` parameter becoming signaled will satisfy the wait.

#### *\_Timeout*

Indicates the number of milliseconds before the wait times out. The value `COOPERATIVE_TIMEOUT_INFINITE` signifies that there is no timeout.

### Return Value

If the wait was satisfied, the index within the array supplied in the `_PPEvents` parameter which satisfied the wait condition; otherwise, the value `COOPERATIVE_WAIT_TIMEOUT` to indicate that the wait timed out without the condition being satisfied.

### Remarks

If the parameter `_FWaitAll` is set to the value `true` to indicate that all events must become signaled to satisfy the wait, the index returned by the function carries no special significance other than the fact that it is not the value `COOPERATIVE_WAIT_TIMEOUT`.

### IMPORTANT

In a Universal Windows Platform (UWP) app, do not call `wait_for_multiple` on the ASTA thread because this call can block the current thread and can cause the app to become unresponsive.

## See also

[concurrency Namespace](#)

# IExecutionContext Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

An interface to an execution context which can run on a given virtual processor and be cooperatively context switched.

## Syntax

```
struct IExecutionContext;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">IExecutionContext::Dispatch</a>	The method that is called when a thread proxy starts executing a particular execution context. This should be the main worker routine for your scheduler.
<a href="#">IExecutionContext::GetId</a>	Returns a unique identifier for the execution context.
<a href="#">IExecutionContext::GetProxy</a>	Returns an interface to the thread proxy that is executing this context.
<a href="#">IExecutionContext::GetScheduler</a>	Returns an interface to the scheduler this execution context belongs to.
<a href="#">IExecutionContext::SetProxy</a>	Associates a thread proxy with this execution context. The associated thread proxy invokes this method right before it starts executing the context's <a href="#">Dispatch</a> method.

## Remarks

If you are implementing a custom scheduler that interfaces with the Concurrency Runtime's Resource Manager, you will need to implement the [IExecutionContext](#) interface. The threads created by the Resource Manager perform work on behalf of your scheduler by executing the [IExecutionContext::Dispatch](#) method.

## Inheritance Hierarchy

[IExecutionContext](#)

## Requirements

**Header:** conctrm.h

**Namespace:** concurrency

## IExecutionContext::Dispatch Method

The method that is called when a thread proxy starts executing a particular execution context. This should be the main worker routine for your scheduler.

```
virtual void Dispatch(_Inout_ DispatchState* pDispatchState) = 0;
```

### Parameters

*pDispatchState*

A pointer to the state under which this execution context is being dispatched. For more information on dispatch state, see [DispatchState](#).

## IExecutionContext::GetId Method

Returns a unique identifier for the execution context.

```
virtual unsigned int GetId() const = 0;
```

### Return Value

A unique integer identifier.

### Remarks

You should use the method `GetExecutionContextId` to obtain a unique identifier for the object that implements the `IExecutionContext` interface, before you use the interface as a parameter to methods supplied by the Resource Manager. You are expected to return the same identifier when the `GetId` function is invoked.

An identifier obtained from a different source could result in undefined behavior.

## IExecutionContext::GetProxy Method

Returns an interface to the thread proxy that is executing this context.

```
virtual IThreadProxy* GetProxy() = 0;
```

### Return Value

An `IThreadProxy` interface. If the execution context's thread proxy has not been initialized with a call to `SetProxy`, the function must return `NULL`.

### Remarks

The Resource Manager will invoke the `SetProxy` method on an execution context, with an `IThreadProxy` interface as a parameter, prior to entering the `Dispatch` method on the on the context. You are expected to store this argument and return it on calls to `GetProxy()`.

## IExecutionContext::GetScheduler Method

Returns an interface to the scheduler this execution context belongs to.

```
virtual IScheduler* GetScheduler() = 0;
```

### Return Value

An `IScheduler` interface.

### Remarks

You are required to initialize the execution context with a valid `IScheduler` interface before you use it as a parameter to methods supplied by the Resource Manager.

## IExecutionContext::SetProxy Method

Associates a thread proxy with this execution context. The associated thread proxy invokes this method right before it starts executing the context's `Dispatch` method.

```
virtual void SetProxy(_Inout_ IThreadProxy* pThreadProxy) = 0;
```

### Parameters

*pThreadProxy*

An interface to the thread proxy that is about to enter the `Dispatch` method on this execution context.

### Remarks

You are expected to save the parameter `pThreadProxy` and return it on a call to the `GetProxy` method. The Resource Manager guarantees that the thread proxy associated with the execution context will not change while the thread proxy is executing the `Dispatch` method.

## See also

[concurrency Namespace](#)

[IScheduler Structure](#)

[IThreadProxy Structure](#)

# IExecutionResource Structure

3/4/2019 • 4 minutes to read • [Edit Online](#)

An abstraction for a hardware thread.

## Syntax

```
struct IExecutionResource;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">IExecutionResource::CurrentSubscriptionLevel</a>	Returns the number of activated virtual processor roots and subscribed external threads currently associated with the underlying hardware thread this execution resource represents.
<a href="#">IExecutionResource::GetExecutionResourceId</a>	Returns a unique identifier for the hardware thread that this execution resource represents.
<a href="#">IExecutionResource::GetNodeId</a>	Returns a unique identifier for the processor node that this execution resource belongs to.
<a href="#">IExecutionResource::Remove</a>	Returns this execution resource to the Resource Manager.

## Remarks

Execution resources can be standalone or associated with virtual processor roots. A standalone execution resource is created when a thread in your application creates a thread subscription. The methods [ISchedulerProxy::SubscribeThread](#) and [ISchedulerProxy::RequestInitialVirtualProcessors](#) create thread subscriptions, and return an `IExecutionResource` interface representing the subscription. Creating a thread subscription is a way to inform the Resource Manager that a given thread will participate in the work queued to a scheduler, along with the virtual processor roots Resource Manager assigns to the scheduler. The Resource Manager uses the information to avoid oversubscribing hardware threads where it can.

## Inheritance Hierarchy

`IExecutionResource`

## Requirements

**Header:** conctrm.h

**Namespace:** concurrency

## IExecutionResource::CurrentSubscriptionLevel Method

Returns the number of activated virtual processor roots and subscribed external threads currently associated with the underlying hardware thread this execution resource represents.

```
virtual unsigned int CurrentSubscriptionLevel() const = 0;
```

### Return Value

The current subscription level.

### Remarks

The subscription level tells you how many running threads are associated with the hardware thread. This only includes threads the Resource Manager is aware of in the form of subscribed threads, and virtual processor roots that are actively executing thread proxies.

Calling the method [ISchedulerProxy::SubscribeCurrentThread](#), or the method [ISchedulerProxy::RequestInitialVirtualProcessors](#) with the parameter `doSubscribeCurrentThread` set to the value **true** increments the subscription level of a hardware thread by one. They also return an [IExecutionResource](#) interface representing the subscription. A corresponding call to the [IExecutionResource::Remove](#) decrements the hardware thread's subscription level by one.

The act of activating a virtual processor root using the method [IVirtualProcessorRoot::Activate](#) increments the subscription level of a hardware thread by one. The methods [IVirtualProcessorRoot::Deactivate](#), or [IExecutionResource::Remove](#) decrement the subscription level by one when invoked on an activated virtual processor root.

The Resource Manager uses subscription level information as one of the ways in which to determine when to move resources between schedulers.

## IExecutionResource::GetExecutionResourceId Method

Returns a unique identifier for the hardware thread that this execution resource represents.

```
virtual unsigned int GetExecutionResourceId() const = 0;
```

### Return Value

A unique identifier for the hardware thread underlying this execution resource.

### Remarks

Each hardware thread is assigned a unique identifier by the Concurrency Runtime. If multiple execution resources are associated hardware thread, they will all have the same execution resource identifier.

## IExecutionResource::GetNodeId Method

Returns a unique identifier for the processor node that this execution resource belongs to.

```
virtual unsigned int GetNodeId() const = 0;
```

### Return Value

A unique identifier for a processor node.

### Remarks

The Concurrency Runtime represents hardware threads on the system in groups of processor nodes. Nodes are usually derived from the hardware topology of the system. For example, all processors on a specific socket or a

specific NUMA node may belong to the same processor node. The Resource Manager assigns unique identifiers to these nodes starting with `0` up to and including `nodeCount - 1`, where `nodeCount` represents the total number of processor nodes on the system.

The count of nodes can be obtained from the function [GetProcessorNodeCount](#).

## IExecutionResource::Remove Method

Returns this execution resource to the Resource Manager.

```
virtual void Remove(_Inout_ IScheduler* pScheduler) = 0;
```

### Parameters

*pScheduler*

An interface to the scheduler making the request to remove this execution resource.

### Remarks

Use this method to return standalone execution resources as well as execution resources associated with virtual processor roots to the Resource Manager.

If this is a standalone execution resource you received from either of the methods [ISchedulerProxy::SubscribeCurrentThread](#) or [ISchedulerProxy::RequestInitialVirtualProcessors](#), calling the method `Remove` will end the thread subscription that the resource was created to represent. You are required to end all thread subscriptions before shutting down a scheduler proxy, and must call `Remove` from the thread that created the subscription.

Virtual processor roots, too, can be returned to the Resource Manager by invoking the `Remove` method, because the interface `IVirtualProcessorRoot` inherits from the `IExecutionResource` interface. You may need to return a virtual processor root either in response to a call to the [IScheduler::RemoveVirtualProcessors](#) method, or when you are done with an oversubscribed virtual processor root you obtained from the [ISchedulerProxy::CreateOversubscriber](#) method. For virtual processor roots, there are no restrictions on which thread can invoke the `Remove` method.

`invalid_argument` is thrown if the parameter `pScheduler` is set to `NULL`.

`invalid_operation` is thrown if the parameter `pScheduler` is different from the scheduler that this execution resource was created for, or, with a standalone execution resource, if the current thread is different from the thread that created the thread subscription.

## See also

[concurrency Namespace](#)

[IVirtualProcessorRoot Structure](#)

# improper\_lock Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when a lock is acquired improperly.

## Syntax

```
class improper_lock : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">improper_lock</a>	Overloaded. Constructs an <code>improper_lock</code> exception .

## Remarks

Typically, this exception is thrown when an attempt is made to acquire a non-reentrant lock recursively on the same context.

## Inheritance Hierarchy

`exception`

`improper_lock`

## Requirements

**Header:** `concrct.h`

**Namespace:** `concurrency`

## improper\_lock

Constructs an `improper_lock` exception .

```
explicit _CRTIMP improper_lock(_In_z_ const char* _Message) throw();  
  
improper_lock() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

critical\_section Class  
reader\_writer\_lock Class

# improper\_scheduler\_attach Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when the `Attach` method is called on a `Scheduler` object which is already attached to the current context.

## Syntax

```
class improper_scheduler_attach : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">improper_scheduler_attach</a>	Overloaded. Constructs an <code>improper_scheduler_attach</code> object.

## Inheritance Hierarchy

`exception`

`improper_scheduler_attach`

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## improper\_scheduler\_attach

Constructs an `improper_scheduler_attach` object.

```
explicit _CRTIMP improper_scheduler_attach(_In_z_ const char* _Message) throw();  
  
improper_scheduler_attach() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)  
[Scheduler Class](#)

# improper\_scheduler\_detach Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when the `CurrentScheduler::Detach` method is called on a context which has not been attached to any scheduler using the `Attach` method of a `Scheduler` object.

## Syntax

```
class improper_scheduler_detach : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">improper_scheduler_detach</a>	Overloaded. Constructs an <code>improper_scheduler_detach</code> object.

## Inheritance Hierarchy

`exception`

`improper_scheduler_detach`

## Requirements

**Header:** `concrth`

**Namespace:** `concurrency`

## improper\_scheduler\_detach

Constructs an `improper_scheduler_detach` object.

```
explicit _CRTIMP improper_scheduler_detach(_In_z_ const char* _Message) throw();  
  
improper_scheduler_detach() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)  
[Scheduler Class](#)

# improper\_scheduler\_reference Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when the `Reference` method is called on a `Scheduler` object that is shutting down, from a context that is not part of that scheduler.

## Syntax

```
class improper_scheduler_reference : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">improper_scheduler_reference</a>	Overloaded. Constructs an <code>improper_scheduler_reference</code> object.

## Inheritance Hierarchy

`exception`

`improper_scheduler_reference`

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## `improper_scheduler_reference`

Constructs an `improper_scheduler_reference` object.

```
explicit _CRTIMP improper_scheduler_reference(_In_z_ const char* _Message) throw();  
  
improper_scheduler_reference() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

[Scheduler Class](#)

# invalid\_link\_target Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when the `link_target` method of a messaging block is called and the messaging block is unable to link to the target. This can be the result of exceeding the number of links the messaging block is allowed or attempting to link a specific target twice to the same source.

## Syntax

```
class invalid_link_target : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">invalid_link_target</a>	Overloaded. Constructs an <code>invalid_link_target</code> object.

## Inheritance Hierarchy

`exception`

`invalid_link_target`

## Requirements

**Header:** `concrth`

**Namespace:** `concurrency`

## invalid\_link\_target

Constructs an `invalid_link_target` object.

```
explicit _CRTIMP invalid_link_target(_In_z_ const char* _Message) throw();  
  
invalid_link_target() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

[Asynchronous Message Blocks](#)

# invalid\_multiple\_scheduling Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when a `task_handle` object is scheduled multiple times using the `run` method of a `task_group` or `structured_task_group` object without an intervening call to either the `wait` or `run_and_wait` methods.

## Syntax

```
class invalid_multiple_scheduling : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">invalid_multiple_scheduling</a>	Overloaded. Constructs an <code>invalid_multiple_scheduling</code> object.

## Inheritance Hierarchy

`exception`

`invalid_multiple_scheduling`

## Requirements

**Header:** `concrth`

**Namespace:** `concurrency`

## invalid\_multiple\_scheduling

Constructs an `invalid_multiple_scheduling` object.

```
explicit _CRTIMP invalid_multiple_scheduling(_In_z_ const char* _Message) throw();  
  
invalid_multiple_scheduling() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

[task\\_handle Class](#)

[task\\_group Class](#)

run

wait

run\_and\_wait

structured\_task\_group Class

# invalid\_operation Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when an invalid operation is performed that is not more accurately described by another exception type thrown by the Concurrency Runtime.

## Syntax

```
class invalid_operation : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">invalid_operation</a>	Overloaded. Constructs an <code>invalid_operation</code> object.

## Remarks

The various methods which throw this exception will generally document under what circumstances they will throw it.

## Inheritance Hierarchy

`exception`

`invalid_operation`

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## invalid\_operation

Constructs an `invalid_operation` object.

```
explicit _CRTIMP invalid_operation(_In_z_ const char* _Message) throw();
```

```
invalid_operation() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also



# invalid\_oversubscribe\_operation Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when the `Context::Oversubscribe` method is called with the `_BeginOversubscription` parameter set to **false** without a prior call to the `Context::Oversubscribe` method with the `_BeginOversubscription` parameter set to **true**.

## Syntax

```
class invalid_oversubscribe_operation : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">invalid_oversubscribe_operation</a>	Overloaded. Constructs an <code>invalid_oversubscribe_operation</code> object.

## Inheritance Hierarchy

`exception`

`invalid_oversubscribe_operation`

## Requirements

**Header:** `concrth`

**Namespace:** `concurrency`

## invalid\_oversubscribe\_operation

Constructs an `invalid_oversubscribe_operation` object.

```
explicit _CRTIMP invalid_oversubscribe_operation(_In_z_ const char* _Message) throw();  
  
invalid_oversubscribe_operation() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

# invalid\_scheduler\_policy\_key Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when an invalid or unknown key is passed to a `SchedulerPolicy` object constructor, or the `SetPolicyValue` method of a `SchedulerPolicy` object is passed a key that must be changed using other means such as the `SetConcurrencyLimits` method.

## Syntax

```
class invalid_scheduler_policy_key : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<code>invalid_scheduler_policy_key</code>	Overloaded. Constructs an <code>invalid_scheduler_policy_key</code> object.

## Inheritance Hierarchy

`exception`

`invalid_scheduler_policy_key`

## Requirements

**Header:** `concrth`

**Namespace:** `concurrency`

## invalid\_scheduler\_policy\_key

Constructs an `invalid_scheduler_policy_key` object.

```
explicit _CRTIMP invalid_scheduler_policy_key(_In_z_ const char* _Message) throw();  
  
invalid_scheduler_policy_key() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)  
[SchedulerPolicy Class](#)

# invalid\_scheduler\_policy\_thread\_specification Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when an attempt is made to set the concurrency limits of a

`SchedulerPolicy` object such that the value of the `MinConcurrency` key is less than the value of the `MaxConcurrency` key.

## Syntax

```
class invalid_scheduler_policy_thread_specification : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<code>[invalid_scheduler_policy_thread_specification](invalid-scheduler-policy-value-class.md#ctor</code>	Overloaded. Constructs an <code>invalid_scheduler_policy_value</code> object.

## Inheritance Hierarchy

`exception`

`invalid_scheduler_policy_thread_specification`

## Requirements

**Header:** `concrth`

**Namespace:** `concurrency`

## invalid\_scheduler\_policy\_thread\_specification

Constructs an `invalid_scheduler_policy_value` object.

```
explicit _CRTIMP invalid_scheduler_policy_thread_specification(_In_z_ const char* _Message) throw();  
  
invalid_scheduler_policy_thread_specification() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)  
[SchedulerPolicy Class](#)

# invalid\_scheduler\_policy\_value Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when a policy key of a `SchedulerPolicy` object is set to an invalid value for that key.

## Syntax

```
class invalid_scheduler_policy_value : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<code>[invalid_scheduler_policy_value](invalid-scheduler-policy-thread-specification-class.md#ctor</code>	Overloaded. Constructs an <code>invalid_scheduler_policy_value</code> object.

## Inheritance Hierarchy

`exception`

`invalid_scheduler_policy_value`

## Requirements

**Header:** `concrth`

**Namespace:** `concurrency`

## invalid\_scheduler\_policy\_value

Constructs an `invalid_scheduler_policy_value` object.

```
explicit _CRTIMP invalid_scheduler_policy_value(_In_z_ const char* _Message) throw();  
  
invalid_scheduler_policy_value() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

[SchedulerPolicy Class](#)

# IResourceManager Structure

3/28/2019 • 4 minutes to read • [Edit Online](#)

An interface to the Concurrency Runtime's Resource Manager. This is the interface by which schedulers communicate with the Resource Manager.

## Syntax

```
struct IResourceManager;
```

## Members

### Public Enumerations

NAME	DESCRIPTION
<a href="#">IResourceManager::OSVersion</a>	An enumerated type that represents the operating system version.

### Public Methods

NAME	DESCRIPTION
<a href="#">IResourceManager::CreateNodeTopology</a>	Present only in debug builds of the runtime, this method is a test hook designed to facilitate testing of the Resource Manager on varying hardware topologies, without requiring actual hardware matching the configuration. With retail builds of the runtime, this method will return without performing any action.
<a href="#">IResourceManager::GetAvailableNodeCount</a>	Returns the number of nodes available to the Resource Manager.
<a href="#">IResourceManager::GetFirstNode</a>	Returns the first node in enumeration order as defined by the Resource Manager.
<a href="#">IResourceManager::Reference</a>	Increments the reference count on the Resource Manager instance.
<a href="#">IResourceManager::RegisterScheduler</a>	Registers a scheduler with the Resource Manager. Once the scheduler is registered, it should communicate with the Resource Manager using the <a href="#">ISchedulerProxy</a> interface that is returned.
<a href="#">IResourceManager::Release</a>	Decrements the reference count on the Resource Manager instance. The Resource Manager is destroyed when its reference count goes to 0.

## Remarks

Use the [CreateResourceManager](#) function to obtain an interface to the singleton Resource Manager instance. The

method increments a reference count on the Resource Manager, and you should invoke the [IResourceManager::Release](#) method to release the reference when you are done with Resource Manager. Typically, each scheduler you create will invoke this method during creation, and release the reference to the Resource Manager after it shuts down.

## Inheritance Hierarchy

IResourceManager

## Requirements

**Header:** conctrm.h

**Namespace:** concurrency

## IResourceManager::CreateNodeTopology Method

Present only in debug builds of the runtime, this method is a test hook designed to facilitate testing of the Resource Manager on varying hardware topologies, without requiring actual hardware matching the configuration. With retail builds of the runtime, this method will return without performing any action.

```
virtual void CreateNodeTopology(  
    unsigned int nodeCount,  
    _In_reads_(nodeCount) unsigned int* pCoreCount,  
    _In_reads_opt_(nodeCount) unsigned int** pNodeDistance,  
    _In_reads_(nodeCount) unsigned int* pProcessorGroups) = 0;
```

### Parameters

*nodeCount*

The number of processor nodes being simulated.

*pCoreCount*

An array that specifies the number of cores on each node.

*pNodeDistance*

A matrix specifying the node distance between any two nodes. This parameter can have the value `NULL`.

*pProcessorGroups*

An array that specifies the processor group each node belongs to.

### Remarks

[invalid\\_argument](#) is thrown if the parameter `nodeCount` has the value `0` was passed in, or if the parameter `pCoreCount` has the value `NULL`.

[invalid\\_operation](#) is thrown if this method is called while other schedulers exist in the process.

## IResourceManager::GetAvailableNodeCount Method

Returns the number of nodes available to the Resource Manager.

```
virtual unsigned int GetAvailableNodeCount() const = 0;
```

### Return Value

The number of nodes available to the Resource Manager.

## IResourceManager::GetFirstNode Method

Returns the first node in enumeration order as defined by the Resource Manager.

```
virtual ITopologyNode* GetFirstNode() const = 0;
```

### Return Value

The first node in enumeration order as defined by the Resource Manager.

## IResourceManager::OSVersion Enumeration

An enumerated type that represents the operating system version.

```
enum OSVersion;
```

## IResourceManager::Reference Method

Increments the reference count on the Resource Manager instance.

```
virtual unsigned int Reference() = 0;
```

### Return Value

The resulting reference count.

## IResourceManager::RegisterScheduler Method

Registers a scheduler with the Resource Manager. Once the scheduler is registered, it should communicate with the Resource Manager using the `ISchedulerProxy` interface that is returned.

```
virtual ISchedulerProxy *RegisterScheduler(  
    _Inout_ IScheduler* pScheduler,  
    unsigned int version) = 0;
```

### Parameters

*pScheduler*

An `IScheduler` interface to the scheduler to be registered.

*version*

The version of communication interface the scheduler is using to communicate with the Resource Manager. Using a version allows the Resource Manager to evolve the communication interface while allowing schedulers to obtain access to older features. Schedulers that wish to use Resource Manager features present in Visual Studio 2010 should use the version `CONCRT_RM_VERSION_1`.

### Return Value

The `ISchedulerProxy` interface the Resource Manager has associated with your scheduler. Your scheduler should use this interface to communicate with Resource Manager from this point on.

### Remarks

Use this method to initiate communication with the Resource Manager. The method associates the `IScheduler` interface for your scheduler with an `ISchedulerProxy` interface and hands it back to you. You can use the returned interface to request execution resources for use by your scheduler, or to subscribe threads with the Resource

Manager. The Resource Manager will use policy elements from the scheduler policy returned by the [IScheduler::GetPolicy](#) method to determine what type of threads the scheduler will need to execute work. If your `SchedulerKind` policy key has the value `UmsThreadDefault` and the value is read back out of the policy as the value `UmsThreadDefault`, the `IScheduler` interface passed to the method must be an `IUMScheduler` interface.

The method throws an `invalid_argument` exception if the parameter `pScheduler` has the value `NULL` or if the parameter `version` is not a valid version for the communication interface.

## IResourceManager::Release Method

Decrements the reference count on the Resource Manager instance. The Resource Manager is destroyed when its reference count goes to `0`.

```
virtual unsigned int Release() = 0;
```

### Return Value

The resulting reference count.

## See also

[concurrency Namespace](#)

[ISchedulerProxy Structure](#)

[IScheduler Structure](#)

# IScheduler Structure

3/4/2019 • 7 minutes to read • [Edit Online](#)

An interface to an abstraction of a work scheduler. The Concurrency Runtime's Resource Manager uses this interface to communicate with work schedulers.

## Syntax

```
struct IScheduler;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">IScheduler::AddVirtualProcessors</a>	Provides a scheduler with a set of virtual processor roots for its use. Each <code>IVirtualProcessorRoot</code> interface represents the right to execute a single thread that can perform work on behalf of the scheduler.
<a href="#">IScheduler::GetId</a>	Returns a unique identifier for the scheduler.
<a href="#">IScheduler::GetPolicy</a>	Returns a copy of the scheduler's policy. For more information on scheduler policies, see <a href="#">SchedulerPolicy</a> .
<a href="#">IScheduler::NotifyResourcesExternallyBusy</a>	Notifies this scheduler that the hardware threads represented by the set of virtual processor roots in the array <code>ppVirtualProcessorRoots</code> are now being used by other schedulers.
<a href="#">IScheduler::NotifyResourcesExternallyIdle</a>	Notifies this scheduler that the hardware threads represented by the set of virtual processor roots in the array <code>ppVirtualProcessorRoots</code> are not being used by other schedulers.
<a href="#">IScheduler::RemoveVirtualProcessors</a>	Initiates the removal of virtual processor roots that were previously allocated to this scheduler.
<a href="#">IScheduler::Statistics</a>	Provides information related to task arrival and completion rates, and change in queue length for a scheduler.

## Remarks

If you are implementing a custom scheduler that communicates with the Resource Manager, you should provide an implementation of the `IScheduler` interface. This interface is one end of a two-way channel of communication between a scheduler and the Resource Manager. The other end is represented by the `IResourceManager` and `ISchedulerProxy` interfaces which are implemented by the Resource Manager.

# Inheritance Hierarchy

IScheduler

## Requirements

**Header:** conctrm.h

**Namespace:** concurrency

## IScheduler::AddVirtualProcessors Method

Provides a scheduler with a set of virtual processor roots for its use. Each `IVirtualProcessorRoot` interface represents the right to execute a single thread that can perform work on behalf of the scheduler.

```
virtual void AddVirtualProcessors(  
    _In_reads_(count) IVirtualProcessorRoot** ppVirtualProcessorRoots,  
    unsigned int count) = 0;
```

### Parameters

*ppVirtualProcessorRoots*

An array of `IVirtualProcessorRoot` interfaces representing the virtual processor roots being added to the scheduler.

*count*

The number of `IVirtualProcessorRoot` interfaces in the array.

### Remarks

The Resource Manager invokes the `AddVirtualProcessor` method to grant an initial set of virtual processor roots to a scheduler. It could also invoke the method to add virtual processor roots to the scheduler when it rebalances resources among schedulers.

## IScheduler::GetId Method

Returns a unique identifier for the scheduler.

```
virtual unsigned int GetId() const = 0;
```

### Return Value

A unique integer identifier.

### Remarks

You should use the [GetSchedulerId](#) function to obtain a unique identifier for the object that implements the `IScheduler` interface, before you use the interface as a parameter to methods supplied by the Resource Manager. You are expected to return the same identifier when the `GetId` function is invoked.

An identifier obtained from a different source could result in undefined behavior.

## IScheduler::GetPolicy Method

Returns a copy of the scheduler's policy. For more information on scheduler policies, see [SchedulerPolicy](#).

```
virtual SchedulerPolicy GetPolicy() const = 0;
```

## Return Value

A copy of the scheduler's policy.

# IScheduler::NotifyResourcesExternallyBusy Method

Notifies this scheduler that the hardware threads represented by the set of virtual processor roots in the array `ppVirtualProcessorRoots` are now being used by other schedulers.

```
virtual void NotifyResourcesExternallyBusy(  
    _In_reads_(count) IVirtualProcessorRoot** ppVirtualProcessorRoots,  
    unsigned int count) = 0;
```

## Parameters

*ppVirtualProcessorRoots*

An array of `IVirtualProcessorRoot` interfaces associated with the hardware threads on which other schedulers have become busy.

*count*

The number of `IVirtualProcessorRoot` interfaces in the array.

## Remarks

It is possible for a particular hardware thread to be assigned to multiple schedulers at the same time. One reason for this could be that there are not enough hardware threads on the system to satisfy the minimum concurrency for all schedulers, without sharing resources. Another possibility is that resources are temporarily assigned to other schedulers when the owning scheduler is not using them, by way of all its virtual processor roots on that hardware thread being deactivated.

The subscription level of a hardware thread is denoted by the number of subscribed threads and activated virtual processor roots associated with that hardware thread. From a particular scheduler's point of view, the external subscription level of a hardware thread is the portion of the subscription other schedulers contribute to.

Notifications that resources are externally busy are sent to a scheduler when the external subscription level for a hardware thread moves from zero into positive territory.

Notifications via this method are only sent to schedulers that have a policy where the value for the `MinConcurrency` policy key is equal to the value for the `MaxConcurrency` policy key. For more information on scheduler policies, see [SchedulerPolicy](#).

A scheduler that qualifies for notifications gets a set of initial notifications when it is created, informing it whether the resources it was just assigned are externally busy or idle.

# IScheduler::NotifyResourcesExternallyIdle Method

Notifies this scheduler that the hardware threads represented by the set of virtual processor roots in the array `ppVirtualProcessorRoots` are not being used by other schedulers.

```
virtual void NotifyResourcesExternallyIdle(  
    _In_reads_(count) IVirtualProcessorRoot** ppVirtualProcessorRoots,  
    unsigned int count) = 0;
```

## Parameters

*ppVirtualProcessorRoots*

An array of `IVirtualProcessorRoot` interfaces associated with hardware threads on which other schedulers have become idle.

*count*

The number of `IVirtualProcessorRoot` interfaces in the array.

### Remarks

It is possible for a particular hardware thread to be assigned to multiple schedulers at the same time. One reason for this could be that there are not enough hardware threads on the system to satisfy the minimum concurrency for all schedulers, without sharing resources. Another possibility is that resources are temporarily assigned to other schedulers when the owning scheduler is not using them, by way of all its virtual processor roots on that hardware thread being deactivated.

The subscription level of a hardware thread is denoted by the number of subscribed threads and activated virtual processor roots associated with that hardware thread. From a particular scheduler's point of view, the external subscription level of a hardware thread is the portion of the subscription other schedulers contribute to.

Notifications that resources are externally busy are sent to a scheduler when the external subscription level for a hardware thread falls to zero from a previous positive value.

Notifications via this method are only sent to schedulers that have a policy where the value for the `MinConcurrency` policy key is equal to the value for the `MaxConcurrency` policy key. For more information on scheduler policies, see [SchedulerPolicy](#).

A scheduler that qualifies for notifications gets a set of initial notifications when it is created, informing it whether the resources it was just assigned are externally busy or idle.

## IScheduler::RemoveVirtualProcessors Method

Initiates the removal of virtual processor roots that were previously allocated to this scheduler.

```
virtual void RemoveVirtualProcessors(
    _In_reads_(count) IVirtualProcessorRoot** ppVirtualProcessorRoots,
    unsigned int count) = 0;
```

### Parameters

*ppVirtualProcessorRoots*

An array of `IVirtualProcessorRoot` interfaces representing the virtual processor roots to be removed.

*count*

The number of `IVirtualProcessorRoot` interfaces in the array.

### Remarks

The Resource Manager invokes the `RemoveVirtualProcessors` method to take back a set of virtual processor roots from a scheduler. The scheduler is expected to invoke the `Remove` method on each interface when it is done with the virtual processor roots. Do not use an `IVirtualProcessorRoot` interface once you have invoked the `Remove` method on it.

The parameter `ppVirtualProcessorRoots` points to an array of interfaces. Among the set of virtual processor roots to be removed, the roots that have never been activated can be returned immediately using the `Remove` method. The roots that have been activated and are either executing work, or have been deactivated and are waiting for work to arrive, should be returned asynchronously. The scheduler must make every attempt to remove the virtual processor root as quickly as possible. Delaying removal of the virtual processor roots may result in unintentional oversubscription within the scheduler.

## IScheduler::Statistics Method

Provides information related to task arrival and completion rates, and change in queue length for a scheduler.

```
virtual void Statistics(  
    _Out_ unsigned int* pTaskCompletionRate,  
    _Out_ unsigned int* pTaskArrivalRate,  
    _Out_ unsigned int* pNumberOfTasksEnqueued) = 0;
```

## Parameters

*pTaskCompletionRate*

The number of tasks that have been completed by the scheduler since the last call to this method.

*pTaskArrivalRate*

The number of tasks that have arrived in the scheduler since the last call to this method.

*pNumberOfTasksEnqueued*

The total number of tasks in all scheduler queues.

## Remarks

This method is invoked by the Resource Manager in order to gather statistics for a scheduler. The statistics gathered here will be used to drive dynamic feedback algorithms to determine when it is appropriate to assign more resources to the scheduler and when to take resources away. The values provided by the scheduler can be optimistic and do not necessarily have to reflect the current count accurately.

You should implement this method if you want the Resource Manager to use feedback about such things as task arrival to determine how to balance resource between your scheduler and other schedulers registered with the Resource Manager. If you choose not to gather statistics, you can set the policy key `DynamicProgressFeedback` to the value `DynamicProgressFeedbackDisabled` in your scheduler's policy, and the Resource Manager will not invoke this method on your scheduler.

In the absence of statistical information, the Resource Manager will use hardware thread subscription levels to make resource allocation and migration decisions. For more information on subscription levels, see [IExecutionResource::CurrentSubscriptionLevel](#).

## See also

[concurrency Namespace](#)

[PolicyElementKey](#)

[SchedulerPolicy Class](#)

[IExecutionContext Structure](#)

[IThreadProxy Structure](#)

[IVirtualProcessorRoot Structure](#)

[IResourceManager Structure](#)

# ISchedulerProxy Structure

3/4/2019 • 6 minutes to read • [Edit Online](#)

The interface by which schedulers communicate with the Concurrency Runtime's Resource Manager to negotiate resource allocation.

## Syntax

```
struct ISchedulerProxy;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">ISchedulerProxy::BindContext</a>	Associates an execution context with a thread proxy, if it is not already associated with one.
<a href="#">ISchedulerProxy::CreateOversubscriber</a>	Creates a new virtual processor root on the hardware thread associated with an existing execution resource.
<a href="#">ISchedulerProxy::RequestInitialVirtualProcessors</a>	Requests an initial allocation of virtual processor roots. Every virtual processor root represents the ability to execute one thread that can perform work for the scheduler.
<a href="#">ISchedulerProxy::Shutdown</a>	Notifies the Resource Manager that the scheduler is shutting down. This will cause the Resource Manager to immediately reclaim all resources granted to the scheduler.
<a href="#">ISchedulerProxy::SubscribeCurrentThread</a>	Registers the current thread with the Resource Manager, associating it with this scheduler.
<a href="#">ISchedulerProxy::UnbindContext</a>	Disassociates a thread proxy from the execution context specified by the <code>pContext</code> parameter and returns it to the thread proxy factory's free pool. This method may only be called on an execution context which was bound via the <a href="#">ISchedulerProxy::BindContext</a> method and has not yet been started via being the <code>pContext</code> parameter of an <a href="#">IThreadProxy::SwitchTo</a> method call.

## Remarks

The Resource Manager hands an `ISchedulerProxy` interface to every scheduler that registers with it using the [IResourceManager::RegisterScheduler](#) method.

## Inheritance Hierarchy

`ISchedulerProxy`

# Requirements

**Header:** conctrm.h

**Namespace:** concurrency

## ISchedulerProxy::BindContext Method

Associates an execution context with a thread proxy, if it is not already associated with one.

```
virtual void BindContext(_Inout_ IExecutionContext* pContext) = 0;
```

### Parameters

*pContext*

An interface to the execution context to associate with a thread proxy.

### Remarks

Normally, the [IThreadProxy::SwitchTo](#) method will bind a thread proxy to an execution context on demand. There are, however, circumstances where it is necessary to bind a context in advance to ensure that the [SwitchTo](#) method switches to an already bound context. This is the case on a UMS scheduling context as it cannot call methods that allocate memory, and binding a thread proxy may involve memory allocation if a thread proxy is not readily available in the free pool of the thread proxy factory.

[invalid\\_argument](#) is thrown if the parameter [pContext](#) has the value [NULL](#).

## ISchedulerProxy::CreateOversubscriber Method

Creates a new virtual processor root on the hardware thread associated with an existing execution resource.

```
virtual IVirtualProcessorRoot* CreateOversubscriber(_Inout_ IExecutionResource* pExecutionResource) = 0;
```

### Parameters

*pExecutionResource*

An [IExecutionResource](#) interface that represents the hardware thread you want to oversubscribe.

### Return Value

An [IVirtualProcessorRoot](#) interface.

### Remarks

Use this method when your scheduler wants to oversubscribe a particular hardware thread for a limited amount of time. Once you are done with the virtual processor root, you should return it to the resource manager by calling the [Remove](#) method on the [IVirtualProcessorRoot](#) interface.

You can even oversubscribe an existing virtual processor root, because the [IVirtualProcessorRoot](#) interface inherits from the [IExecutionResource](#) interface.

## ISchedulerProxy::RequestInitialVirtualProcessors Method

Requests an initial allocation of virtual processor roots. Every virtual processor root represents the ability to execute one thread that can perform work for the scheduler.

```
virtual IExecutionResource* RequestInitialVirtualProcessors(bool doSubscribeCurrentThread) = 0;
```

## Parameters

### *doSubscribeCurrentThread*

Whether to subscribe the current thread and account for it during resource allocation.

## Return Value

The `IExecutionResource` interface for the current thread, if the parameter `doSubscribeCurrentThread` has the value **true**. If the value is **false**, the method returns NULL.

## Remarks

Before a scheduler executes any work, it should use this method to request virtual processor roots from the Resource Manager. The Resource Manager will access the scheduler's policy using `IScheduler::GetPolicy` and use the values for the policy keys `MinConcurrency`, `MaxConcurrency` and `TargetOversubscriptionFactor` to determine how many hardware threads to assign to the scheduler initially and how many virtual processor roots to create for every hardware thread. For more information on how scheduler policies are used to determine a scheduler's initial allocation, see [PolicyElementKey](#).

The Resource Manager grants resources to a scheduler by calling the method `IScheduler::AddVirtualProcessors` with a list of virtual processor roots. The method is invoked as a callback into the scheduler before this method returns.

If the scheduler requested subscription for the current thread by setting the parameter `doSubscribeCurrentThread` to **true**, the method returns an `IExecutionResource` interface. The subscription must be terminated at a later point by using the `IExecutionResource::Remove` method.

When determining which hardware threads are selected, the Resource Manager will attempt to optimize for processor node affinity. If subscription is requested for the current thread, it is an indication that the current thread intends to participate in the work assigned to this scheduler. In such a case, the allocated virtual processors roots are located on the processor node the current thread is executing on, if possible.

The act of subscribing a thread increases the subscription level of the underlying hardware thread by one. The subscription level is reduced by one when the subscription is terminated. For more information on subscription levels, see `IExecutionResource::CurrentSubscriptionLevel`.

## ISchedulerProxy::Shutdown Method

Notifies the Resource Manager that the scheduler is shutting down. This will cause the Resource Manager to immediately reclaim all resources granted to the scheduler.

```
virtual void Shutdown() = 0;
```

## Remarks

All `IExecutionContext` interfaces the scheduler received as a result of subscribing an external thread using the methods `ISchedulerProxy::RequestInitialVirtualProcessors` or `ISchedulerProxy::SubscribeCurrentThread` must be returned to the Resource Manager using `IExecutionResource::Remove` before a scheduler shuts itself down.

If your scheduler had any deactivated virtual processor roots, you must activate them using `IVirtualProcessorRoot::Activate`, and have the thread proxies executing on them leave the `Dispatch` method of the execution contexts they are dispatching before you invoke `Shutdown` on a scheduler proxy.

It is not necessary for the scheduler to individually return all of the virtual processor roots the Resource Manager granted to it via calls to the `Remove` method because all virtual processors roots will be returned to the Resource Manager at shutdown.

## ISchedulerProxy::SubscribeCurrentThread Method

Registers the current thread with the Resource Manager, associating it with this scheduler.

```
virtual IExecutionResource* SubscribeCurrentThread() = 0;
```

### Return Value

The `IExecutionResource` interfacing representing the current thread in the runtime.

### Remarks

Use this method if you want the Resource Manager to account for the current thread while allocating resources to your scheduler and other schedulers. It is especially useful when the thread plans to participate in the work queued to your scheduler, along with the virtual processor roots the scheduler receives from the Resource Manager. The Resource Manager uses information to prevent unnecessary oversubscription of hardware threads on the system.

The execution resource received via this method should be returned to the Resource Manager using the [IExecutionResource::Remove](#) method. The thread that calls the `Remove` method must be the same thread that previously called the `SubscribeCurrentThread` method.

The act of subscribing a thread increases the subscription level of the underlying hardware thread by one. The subscription level is reduced by one when the subscription is terminated. For more information on subscription levels, see [IExecutionResource::CurrentSubscriptionLevel](#).

## ISchedulerProxy::UnbindContext Method

Disassociates a thread proxy from the execution context specified by the `pContext` parameter and returns it to the thread proxy factory's free pool. This method may only be called on an execution context which was bound via the [ISchedulerProxy::BindContext](#) method and has not yet been started via being the `pContext` parameter of an [IThreadProxy::SwitchTo](#) method call.

```
virtual void UnbindContext(_Inout_ IExecutionContext* pContext) = 0;
```

### Parameters

*pContext*

The execution context to disassociate from its thread proxy.

## See also

[concurrency Namespace](#)

[IScheduler Structure](#)

[IThreadProxy Structure](#)

[IVirtualProcessorRoot Structure](#)

[IResourceManager Structure](#)

# ISource Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

The `ISource` class is the interface for all source blocks. Source blocks propagate messages to `ITarget` blocks.

## Syntax

```
template<class T>
class ISource;
```

### Parameters

`T`

The data type of the payload within the messages produced by the source block.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>source_type</code>	A type alias for <code>T</code> .

### Public Constructors

NAME	DESCRIPTION
<code>~ISource Destructor</code>	Destroys the <code>ISource</code> object.

### Public Methods

NAME	DESCRIPTION
<code>accept</code>	When overridden in a derived class, accepts a message that was offered by this <code>ISource</code> block, transferring ownership to the caller.
<code>acquire_ref</code>	When overridden in a derived class, acquires a reference count on this <code>ISource</code> block, to prevent deletion.
<code>consume</code>	When overridden in a derived class, consumes a message previously offered by this <code>ISource</code> block and successfully reserved by the target, transferring ownership to the caller.
<code>link_target</code>	When overridden in a derived class, links a target block to this <code>ISource</code> block.
<code>release</code>	When overridden in a derived class, releases a previous successful message reservation.

NAME	DESCRIPTION
<a href="#">release_ref</a>	When overridden in a derived class, releases a reference count on this <code>ISource</code> block.
<a href="#">reserve</a>	When overridden in a derived class, reserves a message previously offered by this <code>ISource</code> block.
<a href="#">unlink_target</a>	When overridden in a derived class, unlinks a target block from this <code>ISource</code> block, if found to be previously linked.
<a href="#">unlink_targets</a>	When overridden in a derived class, unlinks all target blocks from this <code>ISource</code> block.

## Remarks

For more information, see [Asynchronous Message Blocks](#).

## Inheritance Hierarchy

`ISource`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## accept

When overridden in a derived class, accepts a message that was offered by this `ISource` block, transferring ownership to the caller.

```
virtual message<T>* accept(
    runtime_object_identity _MsgId,
    _Inout_ ITarget<T>* _PTarget) = 0;
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the offered `message` object.

*\_PTarget*

A pointer to the target block that is calling the `accept` method.

### Return Value

A pointer to the message that the caller now has ownership of.

### Remarks

The `accept` method is called by a target while a message is being offered by this `ISource` block. The message pointer returned may be different from the one passed into the `propagate` method of the `ITarget` block, if this source decides to make a copy of the message.

## acquire\_ref

When overridden in a derived class, acquires a reference count on this `ISource` block, to prevent deletion.

```
virtual void acquire_ref(_Inout_ ITarget<T>* _PTarget) = 0;
```

### Parameters

*\_PTarget*

A pointer to the target block that is calling this method.

### Remarks

This method is called by an `ITarget` object that is being linked to this source during the `link_target` method.

## consume

When overridden in a derived class, consumes a message previously offered by this `ISource` block and successfully reserved by the target, transferring ownership to the caller.

```
virtual message<T>* consume(  
    runtime_object_identity _MsgId,  
    _Inout_ ITarget<T>* _PTarget) = 0;
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the reserved `message` object.

*\_PTarget*

A pointer to the target block that is calling the `consume` method.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

The `consume` method is similar to `accept`, but must always be preceded by a call to `reserve` that returned **true**.

## ~ISource

Destroys the `ISource` object.

```
virtual ~ISource();
```

## link\_target

When overridden in a derived class, links a target block to this `ISource` block.

```
virtual void link_target(_Inout_ ITarget<T>* _PTarget) = 0;
```

### Parameters

*\_PTarget*

A pointer to the target block being linked to this `ISource` block.

## release

When overridden in a derived class, releases a previous successful message reservation.

```
virtual void release(  
    runtime_object_identity _MsgId,  
    _Inout_ ITarget<T>* _PTarget) = 0;
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the reserved `message` object.

*\_PTarget*

A pointer to the target block that is calling the `release` method.

## release\_ref

When overridden in a derived class, releases a reference count on this `ISource` block.

```
virtual void release_ref(_Inout_ ITarget<T>* _PTarget) = 0;
```

### Parameters

*\_PTarget*

A pointer to the target block that is calling this method.

### Remarks

This method is called by an `ITarget` object that is being unlinked from this source. The source block is allowed to release any resources reserved for the target block.

## reserve

When overridden in a derived class, reserves a message previously offered by this `ISource` block.

```
virtual bool reserve(  
    runtime_object_identity _MsgId,  
    _Inout_ ITarget<T>* _PTarget) = 0;
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the offered `message` object.

*\_PTarget*

A pointer to the target block that is calling the `reserve` method.

### Return Value

**true** if the message was successfully reserved, **false** otherwise. Reservations can fail for many reasons, including: the message was already reserved or accepted by another target, the source could deny reservations, and so forth.

### Remarks

After you call `reserve`, if it succeeds, you must call either `consume` or `release` in order to take or give up possession of the message, respectively.

## unlink\_target

When overridden in a derived class, unlinks a target block from this `ISource` block, if found to be previously linked.

```
virtual void unlink_target(_Inout_ ITarget<T>* _PTarget) = 0;
```

#### Parameters

*\_PTarget*

A pointer to the target block being unlinked from this `ISource` block.

## unlink\_targets

When overridden in a derived class, unlinks all target blocks from this `ISource` block.

```
virtual void unlink_targets() = 0;
```

## See also

[concurrency Namespace](#)

[ITarget Class](#)

# ITarget Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

The `ITarget` class is the interface for all target blocks. Target blocks consume messages offered to them by `ISource` blocks.

## Syntax

```
template<class T>
class ITarget;
```

### Parameters

`T`

The data type of the payload within the messages accepted by the target block.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>filter_method</code>	The signature of any method used by the block that returns a <code>bool</code> value to determine whether an offered message should be accepted.
<code>type</code>	A type alias for <code>T</code> .

### Public Constructors

NAME	DESCRIPTION
<code>~ITarget Destructor</code>	Destroys the <code>ITarget</code> object.

### Public Methods

NAME	DESCRIPTION
<code>propagate</code>	When overridden in a derived class, asynchronously passes a message from a source block to this target block.
<code>send</code>	When overridden in a derived class, synchronously passes a message to the target block.
<code>supports_anonymous_source</code>	When overridden in a derived class, returns true or false depending on whether the message block accepts messages offered by a source that is not linked to it. If the overridden method returns <b>true</b> , the target cannot postpone an offered message, as consumption of a postponed message at a later time requires the source to be identified in its source link registry.

## Protected Methods

NAME	DESCRIPTION
<a href="#">link_source</a>	When overridden in a derived class, links a specified source block to this <code>ITarget</code> block.
<a href="#">unlink_source</a>	When overridden in a derived class, unlinks a specified source block from this <code>ITarget</code> block.
<a href="#">unlink_sources</a>	When overridden in a derived class, unlinks all source blocks from this <code>ITarget</code> block.

## Remarks

For more information, see [Asynchronous Message Blocks](#).

## Inheritance Hierarchy

`ITarget`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## ~ITarget

Destroys the `ITarget` object.

```
virtual ~ITarget();
```

## link\_source

When overridden in a derived class, links a specified source block to this `ITarget` block.

```
virtual void link_source(_Inout_ ISource<T>* _PSource) = 0;
```

### Parameters

*\_PSource*

The `ISource` block being linked to this `ITarget` block.

### Remarks

This function should not be called directly on an `ITarget` block. Blocks should be connected together using the `link_target` method on `ISource` blocks, which will invoke the `link_source` method on the corresponding target.

## propagate

When overridden in a derived class, asynchronously passes a message from a source block to this target block.

```
virtual message_status propagate(  
    _Inout_opt_ message<T>* _PMessage,  
    _Inout_opt_ ISource<T>* _PSource) = 0;
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A `message_status` indication of what the target decided to do with the message.

### Remarks

The method throws an `invalid_argument` exception if either the `_PMessage` or `_PSource` parameter is `NULL`.

## send

When overridden in a derived class, synchronously passes a message to the target block.

```
virtual message_status send(  
    _Inout_ message<T>* _PMessage,  
    _Inout_ ISource<T>* _PSource) = 0;
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A `message_status` indication of what the target decided to do with the message.

### Remarks

The method throws an `invalid_argument` exception if either the `_PMessage` or `_PSource` parameter is `NULL`.

Using the `send` method outside of message initiation and to propagate messages within a network is dangerous and can lead to deadlock.

When `send` returns, the message has either already been accepted, and transferred into the target block, or it has been declined by the target.

## supports\_anonymous\_source

When overridden in a derived class, returns true or false depending on whether the message block accepts messages offered by a source that is not linked to it. If the overridden method returns **true**, the target cannot postpone an offered message, as consumption of a postponed message at a later time requires the source to be identified in its source link registry.

```
virtual bool supports_anonymous_source();
```

## Return Value

**true** if the block can accept message from a source that is not linked to it **false** otherwise.

## unlink\_source

When overridden in a derived class, unlinks a specified source block from this `ITarget` block.

```
virtual void unlink_source(_Inout_ ISource<T>* _PSource) = 0;
```

## Parameters

*\_PSource*

The `ISource` block being unlinked from this `ITarget` block.

## Remarks

This function should not be called directly on an `ITarget` block. Blocks should be disconnected using the `unlink_target` or `unlink_targets` methods on `ISource` blocks, which will invoke the `unlink_source` method on the corresponding target.

## unlink\_sources

When overridden in a derived class, unlinks all source blocks from this `ITarget` block.

```
virtual void unlink_sources() = 0;
```

## See also

[concurrency Namespace](#)

[ISource Class](#)

# IThreadProxy Structure

3/4/2019 • 6 minutes to read • [Edit Online](#)

An abstraction for a thread of execution. Depending on the `SchedulerType` policy key of the scheduler you create, the Resource Manager will grant you a thread proxy that is backed by either a regular Win32 thread or a user-mode schedulable (UMS) thread. UMS threads are supported on 64-bit operating systems with version Windows 7 and higher.

## Syntax

```
struct IThreadProxy;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">IThreadProxy::GetId</a>	Returns a unique identifier for the thread proxy.
<a href="#">IThreadProxy::SwitchOut</a>	Disassociates the context from the underlying virtual processor root.
<a href="#">IThreadProxy::SwitchTo</a>	Performs a cooperative context switch from the currently executing context to a different one.
<a href="#">IThreadProxy::YieldToSystem</a>	Causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the next thread to be executed.

## Remarks

Thread proxies are coupled to execution contexts represented by the interface `IExecutionContext` as a means of dispatching work.

## Inheritance Hierarchy

`IThreadProxy`

## Requirements

**Header:** conctrm.h

**Namespace:** concurrency

## IThreadProxy::GetId Method

Returns a unique identifier for the thread proxy.

```
virtual unsigned int GetId() const = 0;
```

## Return Value

A unique integer identifier.

# IThreadProxy::SwitchOut Method

Disassociates the context from the underlying virtual processor root.

```
virtual void SwitchOut(SwitchingProxyState switchState = Blocking) = 0;
```

## Parameters

*switchState*

Indicates the state of the thread proxy that is executing the switch. The parameter is of type `SwitchingProxyState`.

## Remarks

Use `SwitchOut` if you need to disassociate a context from the virtual processor root it is executing on, for any reason. Depending on the value you pass in to the parameter `switchState`, and whether or not it is executing on a virtual processor root, the call will either return immediately or block the thread proxy associated with the context. It is an error to call `SwitchOut` with the parameter set to `Idle`. Doing so will result in an `invalid_argument` exception.

`SwitchOut` is useful when you want to reduce the number of virtual processor roots your scheduler has, either because the Resource Manager has instructed you to do so, or because you requested a temporary oversubscribed virtual processor root, and are done with it. In this case you should invoke the method [IVirtualProcessorRoot::Remove](#) on the virtual processor root, before making a call to `SwitchOut` with the parameter `switchState` set to `Blocking`. This will block the thread proxy and it will resume execution when a different virtual processor root in the scheduler is available to execute it. The blocking thread proxy can be resumed by calling the function `SwitchTo` to switch to this thread proxy's execution context. You can also resume the thread proxy, by using its associated context to activate a virtual processor root. For more information on how to do this, see [IVirtualProcessorRoot::Activate](#).

`SwitchOut` may also be used when you want reinitialize the virtual processor so it may be activated in the future while either blocking the thread proxy or temporarily detaching it from the virtual processor root it is running on, and the scheduler it is dispatching work for. Use `SwitchOut` with the parameter `switchState` set to `Blocking` if you wish to block the thread proxy. It can later be resumed using either `SwitchTo` or [IVirtualProcessorRoot::Activate](#) as noted above. Use `SwitchOut` with the parameter set to `Nesting` when you want to temporarily detach this thread proxy from the virtual processor root it is running on, and the scheduler the virtual processor is associated with. Calling `SwitchOut` with the parameter `switchState` set to `Nesting` while it is executing on a virtual processor root will cause the root to be reinitialized and the current thread proxy to continue executing without the need for one. The thread proxy is considered to have left the scheduler until it calls the [IThreadProxy::SwitchOut](#) method with `Blocking` at a later point in time. The second call to `SwitchOut` with the parameter set to `Blocking` is intended to return the context to a blocked state so that it can be resumed by either `SwitchTo` or [IVirtualProcessorRoot::Activate](#) in the scheduler it detached from. Because it was not executing on a virtual processor root, no reinitialization takes place.

A reinitialized virtual processor root is no different from a brand new virtual processor root your scheduler has been granted by the Resource Manager. You can use it for execution by activating it with an execution context using [IVirtualProcessorRoot::Activate](#).

`SwitchOut` must be called on the `IThreadProxy` interface that represents the currently executing thread or the results are undefined.

In the libraries and headers that shipped with Visual Studio 2010, this method did not take a parameter and did not reinitialize the virtual processor root. To preserve old behavior, the default parameter value of `Blocking` is supplied.

## IThreadProxy::SwitchTo Method

Performs a cooperative context switch from the currently executing context to a different one.

```
virtual void SwitchTo(  
    _Inout_ IExecutionContext* pContext,  
    SwitchingProxyState switchState) = 0;
```

### Parameters

*pContext*

The execution context to cooperatively switch to.

*switchState*

Indicates the state of the thread proxy that is executing the switch. The parameter is of type `SwitchingProxyState`.

### Remarks

Use this method to switch from one execution context to another, from the [IExecutionContext::Dispatch](#) method of the first execution context. The method associates the execution context `pContext` with a thread proxy if it is not already associated with one. The ownership of the current thread proxy is determined by the value you specify for the `switchState` argument.

Use the value `Idle` when you want to return the currently executing thread proxy to the Resource Manager. Calling `SwitchTo` with the parameter `switchState` set to `Idle` will cause the execution context `pContext` to start executing on the underlying execution resource. Ownership of this thread proxy is transferred to the Resource Manager, and you are expected to return from the execution context's `Dispatch` method soon after `SwitchTo` returns, in order to complete the transfer. The execution context that the thread proxy was dispatching is disassociated from the thread proxy, and the scheduler is free to reuse it or destroy it as it sees fit.

Use the value `Blocking` when you want this thread proxy to enter a blocked state. Calling `SwitchTo` with the parameter `switchState` set to `Blocking` will cause the execution context `pContext` to start executing, and block the current thread proxy until it is resumed. The scheduler retains ownership of the thread proxy when the thread proxy is in the `Blocking` state. The blocking thread proxy can be resumed by calling the function `SwitchTo` to switch to this thread proxy's execution context. You can also resume the thread proxy, by using its associated context to activate a virtual processor root. For more information on how to do this, see [IVirtualProcessorRoot::Activate](#).

Use the value `Nesting` when you want to temporarily detach this thread proxy from the virtual processor root it is running on, and the scheduler it is dispatching work for. Calling `SwitchTo` with the parameter `switchState` set to `Nesting` will cause the execution context `pContext` to start executing and the current thread proxy also continues executing without the need for a virtual processor root. The thread proxy is considered to have left the scheduler until it calls the [IThreadProxy::SwitchOut](#) method at a later point in time. The `IThreadProxy::SwitchOut` method could block the thread proxy until a virtual processor root is available to reschedule it.

`SwitchTo` must be called on the `IThreadProxy` interface that represents the currently executing thread or the results are undefined. The function throws `invalid_argument` if the parameter `pContext` is set to `NULL`.

## IThreadProxy::YieldToSystem Method

Causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the next thread to be executed.

```
virtual void YieldToSystem() = 0;
```

### Remarks

When called by a thread proxy backed by a regular Windows thread, `YieldToSystem` behaves exactly like the Windows function `SwitchToThread`. However, when called from user-mode schedulable (UMS) threads, the `SwitchToThread` function delegates the task of picking the next thread to run to the user mode scheduler, not the operating system. To achieve the desired effect of switching to a different ready thread in the system, use `YieldToSystem`.

`YieldToSystem` must be called on the `IThreadProxy` interface that represents the currently executing thread or the results are undefined.

## See also

[concurrency Namespace](#)

[IExecutionContext Structure](#)

[IScheduler Structure](#)

[IVirtualProcessorRoot Structure](#)

# ITopologyExecutionResource Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

An interface to an execution resource as defined by the Resource Manager.

## Syntax

```
struct ITopologyExecutionResource;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">ITopologyExecutionResource::GetId</a>	Returns the Resource Manager's unique identifier for this execution resource.
<a href="#">ITopologyExecutionResource::GetNext</a>	Returns an interface to the next execution resource in enumeration order.

## Remarks

This interface is typically utilized to walk the topology of the system as observed by the Resource Manager.

## Inheritance Hierarchy

```
ITopologyExecutionResource
```

## Requirements

**Header:** conctrm.h

**Namespace:** concurrency

## ITopologyExecutionResource::GetId Method

Returns the Resource Manager's unique identifier for this execution resource.

```
virtual unsigned int GetId() const = 0;
```

### Return Value

The Resource Manager's unique identifier for this execution resource.

## ITopologyExecutionResource::GetNext Method

Returns an interface to the next execution resource in enumeration order.

```
virtual ITopologyExecutionResource *GetNext() const = 0;
```

### Return Value

An interface to the next execution resource in enumeration order. If there are no more nodes in enumeration order of the node to which this execution resource belongs, this method will return the value `NULL`.

## See also

[concurrency Namespace](#)

# ITopologyNode Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

An interface to a topology node as defined by the Resource Manager. A node contains one or more execution resources.

## Syntax

```
struct ITopologyNode;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">ITopologyNode::GetExecutionResourceCount</a>	Returns the number of execution resources grouped together under this node.
<a href="#">ITopologyNode::GetFirstExecutionResource</a>	Returns the first execution resource grouped under this node in enumeration order.
<a href="#">ITopologyNode::GetId</a>	Returns the Resource Manager's unique identifier for this node.
<a href="#">ITopologyNode::GetNext</a>	Returns an interface to the next topology node in enumeration order.
<a href="#">ITopologyNode::GetNumaNode</a>	Returns the Windows assigned NUMA node number to which this Resource Manager node belongs.

## Remarks

This interface is typically utilized to walk the topology of the system as observed by the Resource Manager.

## Inheritance Hierarchy

```
ITopologyNode
```

## Requirements

**Header:** conctrm.h

**Namespace:** concurrency

## ITopologyNode::GetExecutionResourceCount Method

Returns the number of execution resources grouped together under this node.

```
virtual unsigned int GetExecutionResourceCount() const = 0;
```

### Return Value

The number of execution resources grouped together under this node.

## ITopologyNode::GetFirstExecutionResource Method

Returns the first execution resource grouped under this node in enumeration order.

```
virtual ITopologyExecutionResource *GetFirstExecutionResource() const = 0;
```

### Return Value

The first execution resource grouped under this node in enumeration order.

## ITopologyNode::GetId Method

Returns the Resource Manager's unique identifier for this node.

```
virtual unsigned int GetId() const = 0;
```

### Return Value

The Resource Manager's unique identifier for this node.

### Remarks

The Concurrency Runtime represents hardware threads on the system in groups of processor nodes. Nodes are usually derived from the hardware topology of the system. For example, all processors on a specific socket or a specific NUMA node may belong to the same processor node. The Resource Manager assigns unique identifiers to these nodes starting with `0` up to and including `nodeCount - 1`, where `nodeCount` represents the total number of processor nodes on the system.

The count of nodes can be obtained from the function [GetProcessorNodeCount](#).

## ITopologyNode::GetNext Method

Returns an interface to the next topology node in enumeration order.

```
virtual ITopologyNode *GetNext() const = 0;
```

### Return Value

An interface to the next node in enumeration order. If there are no more nodes in enumeration order of the system topology, this method will return the value `NULL`.

## ITopologyNode::GetNumaNode Method

Returns the Windows assigned NUMA node number to which this Resource Manager node belongs.

```
virtual unsigned long GetNumaNode() const = 0;
```

### Return Value

The Windows assigned NUMA node number to which this Resource Manager node belongs.

**Remarks**

A thread proxy running on a virtual processor root belonging to this node will have affinity to at least the NUMA node level for the NUMA node returned by this method.

## See also

[concurrency Namespace](#)

# IUMSCompletionList Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a UMS completion list. When a UMS thread blocks, the scheduler's designated scheduling context is dispatched in order to make a decision of what to schedule on the underlying virtual processor root while the original thread is blocked. When the original thread unblocks, the operating system queues it to the completion list which is accessible through this interface. The scheduler can query the completion list on the designated scheduling context or any other place it searches for work.

## Syntax

```
struct IUMSCompletionList;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">IUMSCompletionList::GetUnblockNotifications</a>	Retrieves a chain of <a href="#">IUMSUnblockNotification</a> interfaces representing execution contexts whose associated thread proxies have unblocked since the last time this method was invoked.

## Remarks

A scheduler must be extraordinarily careful about what actions are performed after utilizing this interface to dequeue items from the completion list. The items should be placed on the scheduler's list of runnable contexts and be generally accessible as soon as possible. It is entirely possible that one of the dequeued items has been given ownership of an arbitrary lock. The scheduler can make no arbitrary function calls that may block between the call to dequeue items and the placement of those items on a list that can be generally accessed from within the scheduler.

## Inheritance Hierarchy

[IUMSCompletionList](#)

## Requirements

**Header:** conctrm.h

**Namespace:** concurrency

## IUMSCompletionList::GetUnblockNotifications Method

Retrieves a chain of [IUMSUnblockNotification](#) interfaces representing execution contexts whose associated thread proxies have unblocked since the last time this method was invoked.

```
virtual IUMSUnlockNotification *GetUnlockNotifications() = 0;
```

### Return Value

A chain of `IUMSUnlockNotification` interfaces.

### Remarks

The returned notifications are invalid once the execution contexts are rescheduled.

## See also

[concurrency Namespace](#)

[IUMSScheduler Structure](#)

[IUMSUnlockNotification Structure](#)

# IUMSScheduler Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

An interface to an abstraction of a work scheduler that wants the Concurrency Runtime's Resource Manager to hand it user-mode schedulable (UMS) threads. The Resource Manager uses this interface to communicate with UMS thread schedulers. The `IUMSScheduler` interface inherits from the `IScheduler` interface.

## Syntax

```
struct IUMSScheduler : public IScheduler;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">IUMSScheduler::SetCompletionList</a>	Assigns an <code>IUMSCompletionList</code> interface to a UMS thread scheduler.

## Remarks

If you are implementing a custom scheduler that communicates with the Resource Manager, and you want UMS threads to be handed to your scheduler instead of ordinary Win32 threads, you should provide an implementation of the `IUMSScheduler` interface. In addition, you should set the policy value for the scheduler policy key `SchedulerKind` to be `UmsThreadDefault`. If the policy specifies UMS thread, the `IScheduler` interface that is passed as a parameter to the `IResourceManager::RegisterScheduler` method must be an `IUMSScheduler` interface.

The Resource Manager is able to hand you UMS threads only on operating systems that have the UMS feature. 64-bit operating systems with version Windows 7 and higher support UMS threads. If you create a scheduler policy with the `SchedulerKind` key set to the value `UmsThreadDefault` and the underlying platform does not support UMS, the value of the `SchedulerKind` key on that policy will be changed to the value `ThreadScheduler`. You should always read back this policy value before expecting to receive UMS threads.

The `IUMSScheduler` interface is one end of a two-way channel of communication between a scheduler and the Resource Manager. The other end is represented by the `IResourceManager` and `ISchedulerProxy` interfaces, which are implemented by the Resource Manager.

## Inheritance Hierarchy

[IScheduler](#)

`IUMSScheduler`

## Requirements

**Header:** `concrtrm.h`

**Namespace:** `concurrency`

# IUMSScheduler::SetCompletionList Method

Assigns an `IUMSCompletionList` interface to a UMS thread scheduler.

```
virtual void SetCompletionList(_Inout_ IUMSCompletionList* pCompletionList) = 0;
```

## Parameters

*pCompletionList*

The completion list interface for the scheduler. There is a single list per scheduler.

## Remarks

The Resource Manager will invoke this method on a scheduler that specifies it wants UMS threads, after the scheduler has requested an initial allocation of resources. The scheduler can use the `IUMSCompletionList` interface to determine when UMS thread proxies have unblocked. It is only valid to access this interface from a thread proxy running on a virtual processor root assigned to the UMS scheduler.

## See also

[concurrency Namespace](#)

[PolicyElementKey](#)

[IScheduler Structure](#)

[IUMSCompletionList Structure](#)

[IResourceManager Structure](#)

# IUMSThreadProxy Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

An abstraction for a thread of execution. If you want your scheduler to be granted user-mode schedulable (UMS) threads, set the value for the scheduler policy element `SchedulerKind` to `UmsThreadDefault`, and implement the `IUMSScheduler` interface. UMS threads are only supported on 64-bit operating systems with version Windows 7 and higher.

## Syntax

```
struct IUMSThreadProxy : public IThreadProxy;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">IUMSThreadProxy::EnterCriticalRegion</a>	Called in order to enter a critical region. When inside a critical region, the scheduler will not observe asynchronous blocking operations that happen during the region. This means that the scheduler will not be reentered for page faults, thread suspensions, kernel asynchronous procedure calls (APCs), and so forth, for a UMS thread.
<a href="#">IUMSThreadProxy::EnterHyperCriticalRegion</a>	Called in order to enter a hyper-critical region. When inside a hyper-critical region, the scheduler will not observe any blocking operations that happen during the region. This means the scheduler will not be reentered for blocking function calls, lock acquisition attempts which block, page faults, thread suspensions, kernel asynchronous procedure calls (APCs), and so forth, for a UMS thread.
<a href="#">IUMSThreadProxy::ExitCriticalRegion</a>	Called in order to exit a critical region.
<a href="#">IUMSThreadProxy::ExitHyperCriticalRegion</a>	Called in order to exit a hyper-critical region.
<a href="#">IUMSThreadProxy::GetCriticalRegionType</a>	Returns what kind of critical region the thread proxy is within. Because hyper-critical regions are a superset of critical regions, if code has entered a critical region and then a hyper-critical region, <code>InsideHyperCriticalRegion</code> will be returned.

## Inheritance Hierarchy

[IThreadProxy](#)

`IUMSThreadProxy`

## Requirements

**Header:** conctrm.h

**Namespace:** concurrency

## IUMSThreadProxy::EnterCriticalRegion Method

Called in order to enter a critical region. When inside a critical region, the scheduler will not observe asynchronous blocking operations that happen during the region. This means that the scheduler will not be reentered for page faults, thread suspensions, kernel asynchronous procedure calls (APCs), and so forth, for a UMS thread.

```
virtual int EnterCriticalRegion() = 0;
```

### Return Value

The new depth of critical region. Critical regions are reentrant.

## IUMSThreadProxy::EnterHyperCriticalRegion Method

Called in order to enter a hyper-critical region. When inside a hyper-critical region, the scheduler will not observe any blocking operations that happen during the region. This means the scheduler will not be reentered for blocking function calls, lock acquisition attempts which block, page faults, thread suspensions, kernel asynchronous procedure calls (APCs), and so forth, for a UMS thread.

```
virtual int EnterHyperCriticalRegion() = 0;
```

### Return Value

The new depth of hyper-critical region. Hyper-critical regions are reentrant.

### Remarks

The scheduler must be extraordinarily careful about what methods it calls and what locks it acquires in such regions. If code in such a region blocks on a lock that is held by something the scheduler is responsible for scheduling, deadlock may ensue.

## IUMSThreadProxy::ExitCriticalRegion Method

Called in order to exit a critical region.

```
virtual int ExitCriticalRegion() = 0;
```

### Return Value

The new depth of critical region. Critical regions are reentrant.

## IUMSThreadProxy::ExitHyperCriticalRegion Method

Called in order to exit a hyper-critical region.

```
virtual int ExitHyperCriticalRegion() = 0;
```

### Return Value

The new depth of hyper-critical region. Hyper-critical regions are reentrant.

## IUMSThreadProxy::GetCriticalRegionType Method

Returns what kind of critical region the thread proxy is within. Because hyper-critical regions are a superset of critical regions, if code has entered a critical region and then a hyper-critical region, `InsideHyperCriticalRegion` will be returned.

```
virtual CriticalRegionType GetCriticalRegionType() const = 0;
```

### Return Value

The type of critical region the thread proxy is within.

## See also

[concurrency Namespace](#)

[IUMSScheduler Structure](#)

# IUMSUnlockNotification Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a notification from the Resource Manager that a thread proxy which blocked and triggered a return to the scheduler's designated scheduling context has unblocked and is ready to be scheduled. This interface is invalid once the thread proxy's associated execution context, returned from the `GetContext` method, is rescheduled.

## Syntax

```
struct IUMSUnlockNotification;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">IUMSUnlockNotification::GetContext</a>	Returns the <code>IExecutionContext</code> interface for the execution context associated with the thread proxy which has unblocked. Once this method returns and the underlying execution context has been rescheduled via a call to the <code>IThreadProxy::SwitchTo</code> method, this interface is no longer valid.
<a href="#">IUMSUnlockNotification::GetNextUnlockNotification</a>	Returns the next <code>IUMSUnlockNotification</code> interface in the chain returned from the method <code>IUMSCompletionList::GetUnlockNotifications</code> .

## Inheritance Hierarchy

`IUMSUnlockNotification`

## Requirements

**Header:** conctrm.h

**Namespace:** concurrency

## IUMSUnlockNotification::GetContext Method

Returns the `IExecutionContext` interface for the execution context associated with the thread proxy which has unblocked. Once this method returns and the underlying execution context has been rescheduled via a call to the `IThreadProxy::SwitchTo` method, this interface is no longer valid.

```
virtual IExecutionContext* GetContext() = 0;
```

### Return Value

An `IExecutionContext` interface for the execution context to a thread proxy which has unblocked.

# IUMSUnlockNotification::GetNextUnlockNotification Method

Returns the next `IUMSUnlockNotification` interface in the chain returned from the method

`IUMSCompletionList::GetUnlockNotifications` .

```
virtual IUMSUnlockNotification* GetNextUnlockNotification() = 0;
```

## Return Value

The next `IUMSUnlockNotification` interface in the chain returned from the method

`IUMSCompletionList::GetUnlockNotifications` .

## See also

[concurrency Namespace](#)

[IUMSScheduler Structure](#)

[IUMSCompletionList Structure](#)

# IVirtualProcessorRoot Structure

3/4/2019 • 6 minutes to read • [Edit Online](#)

An abstraction for a hardware thread on which a thread proxy can execute.

## Syntax

```
struct IVirtualProcessorRoot : public IExecutionResource;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">IVirtualProcessorRoot::Activate</a>	Causes the thread proxy associated with the execution context interface <code>pContext</code> to start executing on this virtual processor root.
<a href="#">IVirtualProcessorRoot::Deactivate</a>	Causes the thread proxy currently executing on this virtual processor root to stop dispatching the execution context. The thread proxy will resume executing on a call to the <code>Activate</code> method.
<a href="#">IVirtualProcessorRoot::EnsureAllTasksVisible</a>	Causes data stored in the memory hierarchy of individual processors to become visible to all processors on the system. It ensures that a full memory fence has been executed on all processors before the method returns.
<a href="#">IVirtualProcessorRoot::GetId</a>	Returns a unique identifier for the virtual processor root.

## Remarks

Every virtual processor root has an associated execution resource. The `IVirtualProcessorRoot` interface inherits from the [IExecutionResource](#) interface. Multiple virtual processor roots may correspond to the same underlying hardware thread.

The Resource Manager grants virtual processor roots to schedulers in response to requests for resources. A scheduler can use a virtual processor root to perform work by activating it with an execution context.

## Inheritance Hierarchy

[IExecutionResource](#)

`IVirtualProcessorRoot`

## Requirements

**Header:** `concrtrm.h`

**Namespace:** `concurrency`

# IVirtualProcessorRoot::Activate Method

Causes the thread proxy associated with the execution context interface `pContext` to start executing on this virtual processor root.

```
virtual void Activate(_Inout_ IExecutionContext* pContext) = 0;
```

## Parameters

*pContext*

An interface to the execution context that will be dispatched on this virtual processor root.

## Remarks

The Resource Manager will supply a thread proxy if one is not associated with the execution context interface `pContext`.

The `Activate` method can be used to start executing work on a new virtual processor root returned by the Resource Manager, or to resume the thread proxy on a virtual processor root that has deactivated or is about to deactivate. See [IVirtualProcessorRoot::Deactivate](#) for more information on deactivation. When you are resuming a deactivated virtual processor root, the parameter `pContext` must be the same as the parameter used to deactivate the virtual processor root.

Once a virtual processor root has been activated for the first time, subsequent pairs of calls to `Deactivate` and `Activate` may race with each other. This means it is acceptable for the Resource Manager to receive a call to `Activate` before it receives the `Deactivate` call it was meant for.

When you activate a virtual processor root, you signal to the Resource Manager that this virtual processor root is currently busy with work. If your scheduler cannot find any work to execute on this root, it is expected to invoke the `Deactivate` method informing the Resource Manager that the virtual processor root is idle. The Resource Manager uses this data to load balance the system.

`invalid_argument` is thrown if the argument `pContext` has the value `NULL`.

`invalid_operation` is thrown if the argument `pContext` does not represent the execution context that was most recently dispatched by this virtual processor root.

The act of activating a virtual processor root increases the subscription level of the underlying hardware thread by one. For more information on subscription levels, see [IExecutionResource::CurrentSubscriptionLevel](#).

# IVirtualProcessorRoot::Deactivate Method

Causes the thread proxy currently executing on this virtual processor root to stop dispatching the execution context. The thread proxy will resume executing on a call to the `Activate` method.

```
virtual bool Deactivate(_Inout_ IExecutionContext* pContext) = 0;
```

## Parameters

*pContext*

The context which is currently being dispatched by this root.

## Return Value

A boolean value. A value of **true** indicates that the thread proxy returned from the `Deactivate` method in response to a call to the `Activate` method. A value of `false` indicates that the thread proxy returned from the method in response to a notification event in the Resource Manager. On a user-mode schedulable (UMS) thread scheduler, this indicates that items have appeared on the scheduler's completion list, and the scheduler is required

to handle them.

### Remarks

Use this method to temporarily stop executing a virtual processor root when you cannot find any work in your scheduler. A call to the `Deactivate` method must originate from within the `Dispatch` method of the execution context that the virtual processor root was last activated with. In other words, the thread proxy invoking the `Deactivate` method must be the one that is currently executing on the virtual processor root. Calling the method on a virtual processor root you are not executing on could result in undefined behavior.

A deactivated virtual processor root may be woken up with a call to the `Activate` method, with the same argument that was passed in to the `Deactivate` method. The scheduler is responsible for ensuring that calls to the `Activate` and `Deactivate` methods are paired, but they are not required to be received in a specific order. The Resource Manager can handle receiving a call to the `Activate` method before it receives a call to the `Deactivate` method it was meant for.

If a virtual processor root awakens and the return value from the `Deactivate` method is the value **false**, the scheduler should query the UMS completion list via the `IUMSCompletionList::GetUnblockNotifications` method, act on that information, and then subsequently call the `Deactivate` method again. This should be repeated until such time as the `Deactivate` method returns the value `true`.

`invalid_argument` is thrown if the argument `pContext` has the value `NULL`.

`invalid_operation` is thrown if the virtual processor root has never been activated, or the argument `pContext` does not represent the execution context that was most recently dispatched by this virtual processor root.

The act of deactivating a virtual processor root decreases the subscription level of the underlying hardware thread by one. For more information on subscription levels, see [IExecutionResource::CurrentSubscriptionLevel](#).

## IVirtualProcessorRoot::EnsureAllTasksVisible Method

Causes data stored in the memory hierarchy of individual processors to become visible to all processors on the system. It ensures that a full memory fence has been executed on all processors before the method returns.

```
virtual void EnsureAllTasksVisible(_Inout_ IExecutionContext* pContext) = 0;
```

### Parameters

*pContext*

The context which is currently being dispatched by this virtual processor root.

### Remarks

You may find this method useful when you want to synchronize deactivation of a virtual processor root with the addition of new work into the scheduler. For performance reasons, you may decide to add work items to your scheduler without executing a memory barrier, which means work items added by a thread executing on one processor are not immediately visible to all other processors. By using this method in conjunction with the `Deactivate` method you can ensure that your scheduler does not deactivate all its virtual processor roots while work items exist in your scheduler's collections.

A call to the `EnsureAllTasksVisibleThe` method must originate from within the `Dispatch` method of the execution context that the virtual processor root was last activated with. In other words, the thread proxy invoking the `EnsureAllTasksVisible` method must be the one that is currently executing on the virtual processor root. Calling the method on a virtual processor root you are not executing on could result in undefined behavior.

`invalid_argument` is thrown if the argument `pContext` has the value `NULL`.

`invalid_operation` is thrown if the virtual processor root has never been activated, or the argument `pContext`

does not represent the execution context that was most recently dispatched by this virtual processor root.

## IVirtualProcessorRoot::GetId Method

Returns a unique identifier for the virtual processor root.

```
virtual unsigned int GetId() const = 0;
```

### Return Value

An integer identifier.

## See also

[concurrency Namespace](#)

# join Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

A `join` messaging block is a single-target, multi-source, ordered `propagator_block` which combines together messages of type `T` from each of its sources.

## Syntax

```
template<class T,  
        join_type _Jtype = non_greedy>  
class join : public propagator_block<single_link_registry<ITarget<std::vector<T>>>,  
        multi_link_registry<ISource<T>>>>;
```

### Parameters

*T*

The payload type of the messages joined and propagated by the block.

*\_Jtype*

The kind of `join` block this is, either `greedy` or `non_greedy`

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">join</a>	Overloaded. Constructs a <code>join</code> messaging block.
<a href="#">~join Destructor</a>	Destroys the <code>join</code> block.

### Protected Methods

NAME	DESCRIPTION
<a href="#">accept_message</a>	Accepts a message that was offered by this <code>join</code> messaging block, transferring ownership to the caller.
<a href="#">consume_message</a>	Consumes a message previously offered by the <code>join</code> messaging block and reserved by the target, transferring ownership to the caller.
<a href="#">link_target_notification</a>	A callback that notifies that a new target has been linked to this <code>join</code> messaging block.
<a href="#">propagate_message</a>	Asynchronously passes a message from an <code>ISource</code> block to this <code>join</code> messaging block. It is invoked by the <code>propagate</code> method, when called by a source block.

NAME	DESCRIPTION
<a href="#">propagate_to_any_targets</a>	Constructs an output message containing an input message from each source when they have all propagated a message. Sends this output message out to each of its targets.
<a href="#">release_message</a>	Releases a previous message reservation. (Overrides <a href="#">source_block::release_message</a> .)
<a href="#">reserve_message</a>	Reserves a message previously offered by this <code>join</code> messaging block. (Overrides <a href="#">source_block::reserve_message</a> .)
<a href="#">resume_propagation</a>	Resumes propagation after a reservation has been released. (Overrides <a href="#">source_block::resume_propagation</a> .)

## Remarks

For more information, see [Asynchronous Message Blocks](#).

## Inheritance Hierarchy

[ISource](#)

[ITarget](#)

[source\\_block](#)

[propagator\\_block](#)

`join`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## accept\_message

Accepts a message that was offered by this `join` messaging block, transferring ownership to the caller.

```
virtual message<_OutputType>* accept_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the offered `message` object.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

## consume\_message

Consumes a message previously offered by the `join` messaging block and reserved by the target, transferring ownership to the caller.

```
virtual message<_OutputType>* consume_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being consumed.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

Similar to `accept`, but is always preceded by a call to `reserve`.

## join

Constructs a `join` messaging block.

```
join(
    size_t _NumInputs);

join(
    size_t _NumInputs,
    filter_method const& _Filter);

join(
    Scheduler& _PScheduler,
    size_t _NumInputs);

join(
    Scheduler& _PScheduler,
    size_t _NumInputs,
    filter_method const& _Filter);

join(
    ScheduleGroup& _PScheduleGroup,
    size_t _NumInputs);

join(
    ScheduleGroup& _PScheduleGroup,
    size_t _NumInputs,
    filter_method const& _Filter);
```

### Parameters

*\_NumInputs*

The number of inputs this `join` block will be allowed.

*\_Filter*

A filter function which determines whether offered messages should be accepted.

*\_PScheduler*

The `Scheduler` object within which the propagation task for the `join` messaging block is scheduled.

*\_PScheduleGroup*

The `ScheduleGroup` object within which the propagation task for the `join` messaging block is scheduled. The `Scheduler` object used is implied by the schedule group.

### Remarks

The runtime uses the default scheduler if you do not specify the `_PScheduler` or `_PScheduleGroup` parameters.

The type `filter_method` is a functor with signature `bool (T const &)` which is invoked by this `join` messaging block to determine whether or not it should accept an offered message.

## ~join

Destroys the `join` block.

```
~join();
```

## link\_target\_notification

A callback that notifies that a new target has been linked to this `join` messaging block.

```
virtual void link_target_notification(_Inout_ ITarget<std::vector<T>> *);
```

## propagate\_message

Asynchronously passes a message from an `ISource` block to this `join` messaging block. It is invoked by the `propagate` method, when called by a source block.

```
message_status propagate_message(  
    _Inout_ message<T>* _PMessage,  
    _Inout_ ISource<T>* _PSource);
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A `message_status` indication of what the target decided to do with the message.

## propagate\_to\_any\_targets

Constructs an output message containing an input message from each source when they have all propagated a message. Sends this output message out to each of its targets.

```
void propagate_to_any_targets(_Inout_opt_ message<_OutputType> *);
```

## release\_message

Releases a previous message reservation.

```
virtual void release_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being released.

## reserve\_message

Reserves a message previously offered by this `join` messaging block.

```
virtual bool reserve_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the offered `message` object.

### Return Value

**true** if the message was successfully reserved, **false** otherwise.

### Remarks

After `reserve` is called, if it returns **true**, either `consume` or `release` must be called to either take or release ownership of the message.

## resume\_propagation

Resumes propagation after a reservation has been released.

```
virtual void resume_propagation();
```

## See also

[concurrency Namespace](#)

[choice Class](#)

[multitype\\_join Class](#)

# location Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

An abstraction of a physical location on hardware.

## Syntax

```
class location;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">location</a>	Overloaded. Constructs a <code>location</code> object.
<a href="#">~location Destructor</a>	Destroys a <code>location</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">current</a>	Returns a <code>location</code> object representing the most specific place the calling thread is executing.
<a href="#">from_numa_node</a>	Returns a <code>location</code> object which represents a given NUMA node.

### Public Operators

NAME	DESCRIPTION
<a href="#">operator!=</a>	Determines whether two <code>location</code> objects represent different location.
<a href="#">operator=</a>	Assigns the contents of a different <code>location</code> object to this one.
<a href="#">operator==</a>	Determines whether two <code>location</code> objects represent the same location.

## Inheritance Hierarchy

```
location
```

## Requirements

**Header:** `concr.h`

**Namespace:** concurrency

## ~location

Destroys a `location` object.

```
~location();
```

## current

Returns a `location` object representing the most specific place the calling thread is executing.

```
static location __cdecl current();
```

### Return Value

A location representing the most specific place the calling thread is executing.

## from\_numa\_node

Returns a `location` object which represents a given NUMA node.

```
static location __cdecl from_numa_node(unsigned short _NumaNodeNumber);
```

### Parameters

*\_NumaNodeNumber*

The NUMA node number to construct a location for.

### Return Value

A location representing the NUMA node specified by the `_NumaNodeNumber` parameter.

## location

Constructs a `location` object.

```
location();

location(
    const location& _Src);

location(
    T _LocationType,
    unsigned int _Id,
    unsigned int _BindingId = 0,
    _Inout_opt_ void* _PBinding = NULL);
```

### Parameters

*\_Src*

*\_LocationType*

*\_Id*

*\_BindingId*

*\_PBinding*

(Optional) Binding pointer.

### Remarks

A default constructed location represents the system as a whole.

## operator!=

Determines whether two `location` objects represent different location.

```
bool operator!= (const location& _Rhs) const;
```

### Parameters

*\_Rhs*

Operand `location`.

### Return Value

**true** if the two locations are different, **false** otherwise.

## operator=

Assigns the contents of a different `location` object to this one.

```
location& operator= (const location& _Rhs);
```

### Parameters

*\_Rhs*

The source `location` object.

### Return Value

## operator==

Determines whether two `location` objects represent the same location.

```
bool operator== (const location& _Rhs) const;
```

### Parameters

*\_Rhs*

Operand `location`.

### Return Value

**true** if the two locations are identical, and **false** otherwise.

## See also

[concurrency Namespace](#)

# message Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The basic message envelope containing the data payload being passed between messaging blocks.

## Syntax

```
template<class T>
class message : public ::Concurrency::details::_Runtime_object;
```

### Parameters

*T*

The data type of the payload within the message.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>type</code>	A type alias for <code>T</code> .

### Public Constructors

NAME	DESCRIPTION
<code>message</code>	Overloaded. Constructs a <code>message</code> object.
<code>~message</code> Destructor	Destroys the <code>message</code> object.

### Public Methods

NAME	DESCRIPTION
<code>add_ref</code>	Adds to the reference count for the <code>message</code> object. Used for message blocks that need reference counting to determine message lifetimes.
<code>msg_id</code>	Returns the ID of the <code>message</code> object.
<code>remove_ref</code>	Subtracts from the reference count for the <code>message</code> object. Used for message blocks that need reference counting to determine message lifetimes.

### Public Data Members

NAME	DESCRIPTION
<code>payload</code>	The payload of the <code>message</code> object.

# Remarks

For more information, see [Asynchronous Message Blocks](#).

# Inheritance Hierarchy

message

# Requirements

**Header:** agents.h

**Namespace:** concurrency

# add\_ref

Adds to the reference count for the `message` object. Used for message blocks that need reference counting to determine message lifetimes.

```
long add_ref();
```

## Return Value

The new value of the reference count.

# message

Constructs a `message` object.

```
message(  
    T const& _P);  
  
message(  
    T const& _P,  
    runtime_object_identity _Id);  
  
message(  
    message const& _Msg);  
  
message(  
    _In_ message const* _Msg);
```

## Parameters

*\_P*

The payload of this message.

*\_Id*

The unique ID of this message.

*\_Msg*

A reference or pointer to a `message` object.

## Remarks

The constructor that takes a pointer to a `message` object as an argument throws an `invalid_argument` exception if the parameter `_Msg` is `NULL`.

## ~message

Destroys the `message` object.

```
virtual ~message();
```

## msg\_id

Returns the ID of the `message` object.

```
runtime_object_identity msg_id() const;
```

### Return Value

The `runtime_object_identity` of the `message` object.

## payload

The payload of the `message` object.

```
T const payload;
```

## remove\_ref

Subtracts from the reference count for the `message` object. Used for message blocks that need reference counting to determine message lifetimes.

```
long remove_ref();
```

### Return Value

The new value of the reference count.

## See also

[concurrency Namespace](#)

# message\_not\_found Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when a messaging block is unable to find a requested message.

## Syntax

```
class message_not_found : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">message_not_found</a>	Overloaded. Constructs a <code>message_not_found</code> object.

## Inheritance Hierarchy

`exception`

`message_not_found`

## Requirements

**Header:** conctr.h

**Namespace:** concurrency

## message\_not\_found

Constructs a `message_not_found` object.

```
explicit _CRTIMP message_not_found(_In_z_ const char* _Message) throw();  
  
message_not_found() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

[Asynchronous Message Blocks](#)

# message\_processor Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `message_processor` class is the abstract base class for processing of `message` objects. There is no guarantee on the ordering of the messages.

## Syntax

```
template<class T>
class message_processor;
```

### Parameters

*T*

The data type of the payload within messages handled by this `message_processor` object.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>type</code>	A type alias for <code>T</code> .

### Public Methods

NAME	DESCRIPTION
<code>async_send</code>	When overridden in a derived class, places messages into the block asynchronously.
<code>sync_send</code>	When overridden in a derived class, places messages into the block synchronously.
<code>wait</code>	When overridden in a derived class, waits for all asynchronous operations to complete.

### Protected Methods

NAME	DESCRIPTION
<code>process_incoming_message</code>	When overridden in a derived class, performs the forward processing of messages into the block. Called once every time a new message is added and the queue is found to be empty.

## Inheritance Hierarchy

`message_processor`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## async\_send

When overridden in a derived class, places messages into the block asynchronously.

```
virtual void async_send(_Inout_opt_ message<T>* _Msg) = 0;
```

### Parameters

*\_Msg*

A `message` object to send asynchronously.

### Remarks

Processor implementations should override this method.

## process\_incoming\_message

When overridden in a derived class, performs the forward processing of messages into the block. Called once every time a new message is added and the queue is found to be empty.

```
virtual void process_incoming_message() = 0;
```

### Remarks

Message block implementations should override this method.

## sync\_send

When overridden in a derived class, places messages into the block synchronously.

```
virtual void sync_send(_Inout_opt_ message<T>* _Msg) = 0;
```

### Parameters

*\_Msg*

A `message` object to send synchronously.

### Remarks

Processor implementations should override this method.

## wait

When overridden in a derived class, waits for all asynchronous operations to complete.

```
virtual void wait() = 0;
```

### Remarks

Processor implementations should override this method.

## See also

concurrency Namespace

ordered\_message\_processor Class

# missing\_wait Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when there are tasks still scheduled to a `task_group` or `structured_task_group` object at the time that object's destructor executes. This exception will never be thrown if the destructor is reached because of a stack unwinding as the result of an exception.

## Syntax

```
class missing_wait : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">missing_wait</a>	Overloaded. Constructs a <code>missing_wait</code> object.

## Remarks

Absent exception flow, you are responsible for calling either the `wait` or `run_and_wait` method of a `task_group` or `structured_task_group` object before allowing that object to destruct. The runtime throws this exception as an indication that you forgot to call the `wait` or `run_and_wait` method.

## Inheritance Hierarchy

`exception`

`missing_wait`

## Requirements

**Header:** `concrct.h`

**Namespace:** `concurrency`

## missing\_wait

Constructs a `missing_wait` object.

```
explicit _CRTIMP missing_wait(_In_z_ const char* _Message) throw();  
  
missing_wait() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

[task\\_group Class](#)

[wait](#)

[run\\_and\\_wait](#)

[structured\\_task\\_group Class](#)

# multi\_link\_registry Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `multi_link_registry` object is a `network_link_registry` that manages multiple source blocks or multiple target blocks.

## Syntax

```
template<class _Block>
class multi_link_registry : public network_link_registry<_Block>;
```

### Parameters

`_Block`

The block data type being stored in the `multi_link_registry` object.

## Members

### Public Constructors

NAME	DESCRIPTION
<code>multi_link_registry</code>	Constructs a <code>multi_link_registry</code> object.
<code>~multi_link_registry</code> Destructor	Destroys the <code>multi_link_registry</code> object.

### Public Methods

NAME	DESCRIPTION
<code>add</code>	Adds a link to the <code>multi_link_registry</code> object. (Overrides <code>network_link_registry::add</code> .)
<code>begin</code>	Returns an iterator to the first element in the <code>multi_link_registry</code> object. (Overrides <code>network_link_registry::begin</code> .)
<code>contains</code>	Searches the <code>multi_link_registry</code> object for a specified block. (Overrides <code>network_link_registry::contains</code> .)
<code>count</code>	Counts the number of items in the <code>multi_link_registry</code> object. (Overrides <code>network_link_registry::count</code> .)
<code>remove</code>	Removes a link from the <code>multi_link_registry</code> object. (Overrides <code>network_link_registry::remove</code> .)
<code>set_bound</code>	Sets an upper bound on the number of links that the <code>multi_link_registry</code> object can hold.

## Inheritance Hierarchy

multi\_link\_registry

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## add

Adds a link to the `multi_link_registry` object.

```
virtual void add(_EType _Link);
```

### Parameters

*\_Link*

A pointer to a block to be added.

### Remarks

The method throws an `invalid_link_target` exception if the link is already present in the registry, or if a bound has already been set with the `set_bound` function and a link has since been removed.

## begin

Returns an iterator to the first element in the `multi_link_registry` object.

```
virtual iterator begin();
```

### Return Value

An iterator addressing the first element in the `multi_link_registry` object.

### Remarks

The end state is indicated by a `NULL` link.

## contains

Searches the `multi_link_registry` object for a specified block.

```
virtual bool contains(_EType _Link);
```

### Parameters

*\_Link*

A pointer to a block that is to be searched for in the `multi_link_registry` object.

### Return Value

**true** if the specified block was found, **false** otherwise.

## count

Counts the number of items in the `multi_link_registry` object.

```
virtual size_t count();
```

### Return Value

The number of items in the `multi_link_registry` object.

## multi\_link\_registry

Constructs a `multi_link_registry` object.

```
multi_link_registry();
```

## ~multi\_link\_registry

Destroys the `multi_link_registry` object.

```
virtual ~multi_link_registry();
```

### Remarks

The method throws an `invalid_operation` exception if called before all links are removed.

## remove

Removes a link from the `multi_link_registry` object.

```
virtual bool remove(_EType _Link);
```

### Parameters

*\_Link*

A pointer to a block to be removed, if found.

### Return Value

**true** if the link was found and removed, **false** otherwise.

## set\_bound

Sets an upper bound on the number of links that the `multi_link_registry` object can hold.

```
void set_bound(size_t _MaxLinks);
```

### Parameters

*\_MaxLinks*

The maximum number of links that the `multi_link_registry` object can hold.

### Remarks

After a bound is set, unlinking an entry will cause the `multi_link_registry` object to enter an immutable state where further calls to `add` will throw an `invalid_link_target` exception.

## See also

concurrency Namespace  
single\_link\_registry Class

# multitype\_join Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

A `multitype_join` messaging block is a multi-source, single-target messaging block that combines together messages of different types from each of its sources and offers a tuple of the combined messages to its targets.

## Syntax

```
template<
    typename T,
    join_type _Jtype = non_greedy
>
class multitype_join: public ISource<typename _Unwrap<T>::type>;
```

### Parameters

*T*

The `tuple` payload type of the messages joined and propagated by the block.

*\_Jtype*

The kind of `join` block this is, either `greedy` or `non_greedy`

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>type</code>	A type alias for <code>T</code> .

### Public Constructors

NAME	DESCRIPTION
<code>multitype_join</code>	Overloaded. Constructs a <code>multitype_join</code> messaging block.
<code>~multitype_join</code> Destructor	Destroys the <code>multitype_join</code> messaging block.

### Public Methods

NAME	DESCRIPTION
<code>accept</code>	Accepts a message that was offered by this <code>multitype_join</code> block, transferring ownership to the caller.
<code>acquire_ref</code>	Acquires a reference count on this <code>multitype_join</code> messaging block, to prevent deletion.
<code>consume</code>	Consumes a message previously offered by the <code>multitype_join</code> messaging block and successfully reserved by the target, transferring ownership to the caller.

NAME	DESCRIPTION
<a href="#">link_target</a>	Links a target block to this <code>multitype_join</code> messaging block.
<a href="#">release</a>	Releases a previous successful message reservation.
<a href="#">release_ref</a>	Releases a reference count on this <code>multiple_join</code> messaging block.
<a href="#">reserve</a>	Reserves a message previously offered by this <code>multitype_join</code> messaging block.
<a href="#">unlink_target</a>	Unlinks a target block from this <code>multitype_join</code> messaging block.
<a href="#">unlink_targets</a>	Unlinks all targets from this <code>multitype_join</code> messaging block. (Overrides <code>ISource::unlink_targets</code> .)

## Remarks

For more information, see [Asynchronous Message Blocks](#).

## Inheritance Hierarchy

[ISource](#)

`multitype_join`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## accept

Accepts a message that was offered by this `multitype_join` block, transferring ownership to the caller.

```
virtual message<_Destination_type>* accept(
    runtime_object_identity _MsgId,
    _Inout_ ITarget<_Destination_type>* _PTarget);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the offered `message` object.

*\_PTarget*

A pointer to the target block that is calling the `accept` method.

### Return Value

A pointer to the message that the caller now has ownership of.

## acquire\_ref

Acquires a reference count on this `multitype_join` messaging block, to prevent deletion.

```
virtual void acquire_ref(_Inout_ ITarget<Destination_type>* _PTarget);
```

#### Parameters

*\_PTarget*

A pointer to the target block that is calling this method.

#### Remarks

This method is called by an `ITarget` object that is being linked to this source during the `link_target` method.

## consume

Consumes a message previously offered by the `multitype_join` messaging block and successfully reserved by the target, transferring ownership to the caller.

```
virtual message<Destination_type>* consume(  
    runtime_object_identity _MsgId,  
    _Inout_ ITarget<Destination_type>* _PTarget);
```

#### Parameters

*\_MsgId*

The `runtime_object_identity` of the reserved `message` object.

*\_PTarget*

A pointer to the target block that is calling the `consume` method.

#### Return Value

A pointer to the `message` object that the caller now has ownership of.

#### Remarks

The `consume` method is similar to `accept`, but must always be preceded by a call to `reserve` that returned **true**.

## link\_target

Links a target block to this `multitype_join` messaging block.

```
virtual void link_target(_Inout_ ITarget<Destination_type>* _PTarget);
```

#### Parameters

*\_PTarget*

A pointer to an `ITarget` block to link to this `multitype_join` messaging block.

## multitype\_join

Constructs a `multitype_join` messaging block.

```

explicit multitype_join(
    T _Tuple);

multitype_join(
    Scheduler& _PScheduler,
    T _Tuple);

multitype_join(
    ScheduleGroup& _PScheduleGroup,
    T _Tuple);

multitype_join(
    multitype_join&& _Join);

```

## Parameters

### *\_Tuple*

A `tuple` of sources for this `multitype_join` messaging block.

### *\_PScheduler*

The `Scheduler` object within which the propagation task for the `multitype_join` messaging block is scheduled.

### *\_PScheduleGroup*

The `ScheduleGroup` object within which the propagation task for the `multitype_join` messaging block is scheduled. The `Scheduler` object used is implied by the schedule group.

### *\_Join*

A `multitype_join` messaging block to copy from. Note that the original object is orphaned, making this a move constructor.

## Remarks

The runtime uses the default scheduler if you do not specify the `_PScheduler` or `_PScheduleGroup` parameters.

Move construction is not performed under a lock, which means that it is up to the user to make sure that there are no light-weight tasks in flight at the time of moving. Otherwise, numerous races can occur, leading to exceptions or inconsistent state.

## ~multitype\_join

Destroys the `multitype_join` messaging block.

```
~multitype_join();
```

## release

Releases a previous successful message reservation.

```

virtual void release(
    runtime_object_identity _MsgId,
    _Inout_ ITarget<Destination_type>* _PTarget);

```

## Parameters

### *\_MsgId*

The `runtime_object_identity` of the `message` object being released.

### *\_PTarget*

A pointer to the target block that is calling the `release` method.

## release\_ref

Releases a reference count on this `multiple_join` messaging block.

```
virtual void release_ref(_Inout_ ITarget<Destination_type>* _PTarget);
```

### Parameters

*\_PTarget*

A pointer to the target block that is calling this method.

### Remarks

This method is called by an `ITarget` object that is being unlinked from this source. The source block is allowed to release any resources reserved for the target block.

## reserve

Reserves a message previously offered by this `multitype_join` messaging block.

```
virtual bool reserve(  
    runtime_object_identity _MsgId,  
    _Inout_ ITarget<Destination_type>* _PTarget);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being reserved.

*\_PTarget*

A pointer to the target block that is calling the `reserve` method.

### Return Value

`true` if the message was successfully reserved, `false` otherwise. Reservations can fail for many reasons, including: the message was already reserved or accepted by another target, the source could deny reservations, and so forth.

### Remarks

After you call `reserve`, if it succeeds, you must call either `consume` or `release` in order to take or give up possession of the message, respectively.

## unlink\_target

Unlinks a target block from this `multitype_join` messaging block.

```
virtual void unlink_target(_Inout_ ITarget<Destination_type>* _PTarget);
```

### Parameters

*\_PTarget*

A pointer to an `ITarget` block to unlink from this `multitype_join` messaging block.

## unlink\_targets

Unlinks all targets from this `multitype_join` messaging block.

```
virtual void unlink_targets();
```

## See also

[concurrency Namespace](#)

[choice Class](#)

[join Class](#)

# nested\_scheduler\_missing\_detach Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when the Concurrency Runtime detects that you neglected to call the `CurrentScheduler::Detach` method on a context that attached to a second scheduler using the `Attach` method of the `Scheduler` object.

## Syntax

```
class nested_scheduler_missing_detach : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">nested_scheduler_missing_detach</a>	Overloaded. Constructs a <code>nested_scheduler_missing_detach</code> object.

## Remarks

This exception is thrown only when you nest one scheduler inside another by calling the `Attach` method of a `Scheduler` object on a context that is already owned by or attached to another scheduler. The Concurrency Runtime throws this exception opportunistically when it can detect the scenario as an aid to locating the problem. Not every instance of neglecting to call the `CurrentScheduler::Detach` method is guaranteed to throw this exception.

## Inheritance Hierarchy

`exception`

`nested_scheduler_missing_detach`

## Requirements

**Header:** `concrth`

**Namespace:** `concurrency`

## nested\_scheduler\_missing\_detach

Constructs a `nested_scheduler_missing_detach` object.

```
explicit _CRTIMP nested_scheduler_missing_detach(_In_z_ const char* _Message) throw();  
nested_scheduler_missing_detach() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

[Scheduler Class](#)

# network\_link\_registry Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `network_link_registry` abstract base class manages the links between source and target blocks.

## Syntax

```
template<class _Block>
class network_link_registry;
```

### Parameters

`_Block`

The block data type being stored in the `network_link_registry`.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>const_pointer</code>	A type that provides a pointer to a <code>const</code> element in a <code>network_link_registry</code> object.
<code>const_reference</code>	A type that provides a reference to a <code>const</code> element stored in a <code>network_link_registry</code> object for reading and performing const operations.
<code>iterator</code>	A type that provides an iterator that can read or modify any element in a <code>network_link_registry</code> object.
<code>type</code>	A type that represents the block type stored in the <code>network_link_registry</code> object.

### Public Methods

NAME	DESCRIPTION
<code>add</code>	When overridden in a derived class, adds a link to the <code>network_link_registry</code> object.
<code>begin</code>	When overridden in a derived class, returns an iterator to the first element in the <code>network_link_registry</code> object.
<code>contains</code>	When overridden in a derived class, searches the <code>network_link_registry</code> object for a specified block.
<code>count</code>	When overridden in a derived class, returns the number of items in the <code>network_link_registry</code> object.

NAME	DESCRIPTION
<a href="#">remove</a>	When overridden in a derived class, removes a specified block from the <code>network_link_registry</code> object.

## Remarks

The `network link registry` is not safe for concurrent access.

## Inheritance Hierarchy

`network_link_registry`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## add

When overridden in a derived class, adds a link to the `network_link_registry` object.

```
virtual void add(_EType _Link) = 0;
```

### Parameters

*\_Link*

A pointer to a block to be added.

## begin

When overridden in a derived class, returns an iterator to the first element in the `network_link_registry` object.

```
virtual iterator begin() = 0;
```

### Return Value

An iterator addressing the first element in the `network_link_registry` object.

### Remarks

The end state of the iterator is indicated by a `NULL` link.

## contains

When overridden in a derived class, searches the `network_link_registry` object for a specified block.

```
virtual bool contains(_EType _Link) = 0;
```

### Parameters

*\_Link*

A pointer to a block that is being searched for in the `network_link_registry` object.

### Return Value

**true** if the block was found, **false** otherwise.

## count

When overridden in a derived class, returns the number of items in the `network_link_registry` object.

```
virtual size_t count() = 0;
```

### Return Value

The number of items in the `network_link_registry` object.

## remove

When overridden in a derived class, removes a specified block from the `network_link_registry` object.

```
virtual bool remove(_EType _Link) = 0;
```

### Parameters

*\_Link*

A pointer to a block to be removed, if found.

### Return Value

**true** if the link was found and removed, **false** otherwise.

## See also

[concurrency Namespace](#)

[single\\_link\\_registry Class](#)

[multi\\_link\\_registry Class](#)

# operation\_timed\_out Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when an operation has timed out.

## Syntax

```
class operation_timed_out : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">operation_timed_out</a>	Overloaded. Constructs an <a href="#">operation_timed_out</a> object.

## Inheritance Hierarchy

[exception](#)

[operation\\_timed\\_out](#)

## Requirements

**Header:** conctr.h

**Namespace:** concurrency

## operation\_timed\_out

Constructs an [operation\\_timed\\_out](#) object.

```
explicit _CRTIMP operation_timed_out(_In_z_ const char* _Message) throw();  
operation_timed_out() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

# ordered\_message\_processor Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

An `ordered_message_processor` is a `message_processor` that allows message blocks to process messages in the order they were received.

## Syntax

```
template<class T>
class ordered_message_processor : public message_processor<T>;
```

### Parameters

*T*

The payload type of messages handled by the processor.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>type</code>	A type alias for <code>T</code> .

### Public Constructors

NAME	DESCRIPTION
<code>ordered_message_processor</code>	Constructs an <code>ordered_message_processor</code> object.
<code>~ordered_message_processor</code> Destructor	Destroys the <code>ordered_message_processor</code> object.

### Public Methods

NAME	DESCRIPTION
<code>async_send</code>	Asynchronously queues up messages and starts a processing task, if this has not been done already. (Overrides <code>message_processor::async_send</code> .)
<code>initialize</code>	Initializes the <code>ordered_message_processor</code> object with the appropriate callback function, scheduler and schedule group.
<code>initialize_batched_processing</code>	Initialize batched message processing
<code>sync_send</code>	Synchronously queues up messages and starts a processing task, if this has not been done already. (Overrides <code>message_processor::sync_send</code> .)

NAME	DESCRIPTION
<a href="#">wait</a>	A processor-specific spin wait used in destructors of message blocks to make sure that all asynchronous processing tasks have time to finish before destroying the block. (Overrides <a href="#">message_processor::wait</a> .)

### Protected Methods

NAME	DESCRIPTION
<a href="#">process_incoming_message</a>	The processing function that is called asynchronously. It dequeues messages and begins processing them. (Overrides <a href="#">message_processor::process_incoming_message</a> .)

## Inheritance Hierarchy

[message\\_processor](#)

`ordered_message_processor`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## async\_send

Asynchronously queues up messages and starts a processing task, if this has not been done already.

```
virtual void async_send(_Inout_opt_ message<T>* _Msg);
```

### Parameters

*\_Msg*

A pointer to a message.

## initialize

Initializes the `ordered_message_processor` object with the appropriate callback function, scheduler and schedule group.

```
void initialize(
    _Inout_opt_ Scheduler* _PScheduler,
    _Inout_opt_ ScheduleGroup* _PScheduleGroup,
    _Handler_method const& _Handler);
```

### Parameters

*\_PScheduler*

A pointer to the scheduler to be used for scheduling light-weight tasks.

*\_PScheduleGroup*

A pointer to the schedule group to be used for scheduling light-weight tasks.

*\_Handler*

The handler functor invoked during callback.

## initialize\_batched\_processing

Initialize batched message processing

```
virtual void initialize_batched_processing(  
    _Handler_method const& _Processor,  
    _Propagator_method const& _Propagator);
```

### Parameters

*\_Processor*

The processor functor invoked during callback.

*\_Propagator*

The propagator functor invoked during callback.

## ordered\_message\_processor

Constructs an `ordered_message_processor` object.

```
ordered_message_processor();
```

### Remarks

This `ordered_message_processor` will not schedule asynchronous or synchronous handlers until the `initialize` function is called.

## ~ordered\_message\_processor

Destroys the `ordered_message_processor` object.

```
virtual ~ordered_message_processor();
```

### Remarks

Waits for all outstanding asynchronous operations before destroying the processor.

## process\_incoming\_message

The processing function that is called asynchronously. It dequeues messages and begins processing them.

```
virtual void process_incoming_message();
```

## sync\_send

Synchronously queues up messages and starts a processing task, if this has not been done already.

```
virtual void sync_send(_Inout_opt_message<T>* _Msg);
```

### Parameters

*\_Msg*

A pointer to a message.

## wait

A processor-specific spin wait used in destructors of message blocks to make sure that all asynchronous processing tasks have time to finish before destroying the block.

```
virtual void wait();
```

## See also

[concurrency Namespace](#)

# overwrite\_buffer Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

An `overwrite_buffer` messaging block is a multi-target, multi-source, ordered `propagator_block` capable of storing a single message at a time. New messages overwrite previously held ones.

## Syntax

```
template<class T>
class overwrite_buffer : public propagator_block<multi_link_registry<ITarget<T>>,
multi_link_registry<ISource<T>>>;
```

### Parameters

*T*

The payload type of the messages stored and propagated by the buffer.

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">overwrite_buffer</a>	Overloaded. Constructs an <code>overwrite_buffer</code> messaging block.
<a href="#">~overwrite_buffer</a> Destructor	Destroys the <code>overwrite_buffer</code> messaging block.

### Public Methods

NAME	DESCRIPTION
<a href="#">has_value</a>	Checks whether this <code>overwrite_buffer</code> messaging block has a value yet.
<a href="#">value</a>	Gets a reference to the current payload of the message being stored in the <code>overwrite_buffer</code> messaging block.

### Protected Methods

NAME	DESCRIPTION
<a href="#">accept_message</a>	Accepts a message that was offered by this <code>overwrite_buffer</code> messaging block, returning a copy of the message to the caller.
<a href="#">consume_message</a>	Consumes a message previously offered by the <code>overwrite_buffer</code> messaging block and reserved by the target, returning a copy of the message to the caller.

NAME	DESCRIPTION
<a href="#">link_target_notification</a>	A callback that notifies that a new target has been linked to this <code>overwrite_buffer</code> messaging block.
<a href="#">propagate_message</a>	Asynchronously passes a message from an <code>ISource</code> block to this <code>overwrite_buffer</code> messaging block. It is invoked by the <code>propagate</code> method, when called by a source block.
<a href="#">propagate_to_any_targets</a>	Places the <code>message_PMessage</code> in this <code>overwrite_buffer</code> messaging block and offers it to all of the linked targets.
<a href="#">release_message</a>	Releases a previous message reservation. (Overrides <code>source_block::release_message</code> .)
<a href="#">reserve_message</a>	Reserves a message previously offered by this <code>overwrite_buffer</code> messaging block. (Overrides <code>source_block::reserve_message</code> .)
<a href="#">resume_propagation</a>	Resumes propagation after a reservation has been released. (Overrides <code>source_block::resume_propagation</code> .)
<a href="#">send_message</a>	Synchronously passes a message from an <code>ISource</code> block to this <code>overwrite_buffer</code> messaging block. It is invoked by the <code>send</code> method, when called by a source block.
<a href="#">supports_anonymous_source</a>	Overrides the <code>supports_anonymous_source</code> method to indicate that this block can accept messages offered to it by a source that is not linked. (Overrides <code>ITarget::supports_anonymous_source</code> .)

## Remarks

An `overwrite_buffer` messaging block propagates out copies of its stored message to each of its targets.

For more information, see [Asynchronous Message Blocks](#).

## Inheritance Hierarchy

[ISource](#)

[ITarget](#)

[source\\_block](#)

[propagator\\_block](#)

`overwrite_buffer`

## Requirements

**Header:** `agents.h`

**Namespace:** `concurrency`

`accept_message`

Accepts a message that was offered by this `overwrite_buffer` messaging block, returning a copy of the message to the caller.

```
virtual message<T>* accept_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the offered `message` object.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

The `overwrite_buffer` messaging block returns copies of the message to its targets, rather than transferring ownership of the currently held message.

## consume\_message

Consumes a message previously offered by the `overwrite_buffer` messaging block and reserved by the target, returning a copy of the message to the caller.

```
virtual message<T>* consume_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being consumed.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

Similar to `accept`, but is always preceded by a call to `reserve`.

## has\_value

Checks whether this `overwrite_buffer` messaging block has a value yet.

```
bool has_value() const;
```

### Return Value

**true** if the block has received a value, **false** otherwise.

## link\_target\_notification

A callback that notifies that a new target has been linked to this `overwrite_buffer` messaging block.

```
virtual void link_target_notification(_Inout_ ITarget<T>* _PTarget);
```

### Parameters

*\_PTarget*

A pointer to the newly linked target.

## ~overwrite\_buffer

Destroys the `overwrite_buffer` messaging block.

```
~overwrite_buffer();
```

## overwrite\_buffer

Constructs an `overwrite_buffer` messaging block.

```
overwrite_buffer();

overwrite_buffer(
    filter_method const& _Filter);

overwrite_buffer(
    Scheduler& _PScheduler);

overwrite_buffer(
    Scheduler& _PScheduler,
    filter_method const& _Filter);

overwrite_buffer(
    ScheduleGroup& _PScheduleGroup);

overwrite_buffer(
    ScheduleGroup& _PScheduleGroup,
    filter_method const& _Filter);
```

### Parameters

#### *\_Filter*

A filter function which determines whether offered messages should be accepted.

#### *\_PScheduler*

The `Scheduler` object within which the propagation task for the `overwrite_buffer` messaging block is scheduled.

#### *\_PScheduleGroup*

The `ScheduleGroup` object within which the propagation task for the `overwrite_buffer` messaging block is scheduled. The `Scheduler` object used is implied by the schedule group.

### Remarks

The runtime uses the default scheduler if you do not specify the `_PScheduler` or `_PScheduleGroup` parameters.

The type `filter_method` is a functor with signature `bool (T const &)` which is invoked by this `overwrite_buffer` messaging block to determine whether or not it should accept an offered message.

## propagate\_message

Asynchronously passes a message from an `ISource` block to this `overwrite_buffer` messaging block. It is invoked by the `propagate` method, when called by a source block.

```
virtual message_status propagate_message(
    _Inout_ message<T>* _PMessage,
    _Inout_ ISource<T>* _PSource);
```

## Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

## Return Value

A `message_status` indication of what the target decided to do with the message.

# propagate\_to\_any\_targets

Places the `message _PMessage` in this `overwrite_buffer` messaging block and offers it to all of the linked targets.

```
virtual void propagate_to_any_targets(_Inout_ message<T>* _PMessage);
```

## Parameters

*\_PMessage*

A pointer to a `message` object that this `overwrite_buffer` has taken ownership of.

## Remarks

This method overwrites the current message in the `overwrite_buffer` with the newly accepted message

`_PMessage`.

# send\_message

Synchronously passes a message from an `ISource` block to this `overwrite_buffer` messaging block. It is invoked by the `send` method, when called by a source block.

```
virtual message_status send_message(  
    _Inout_ message<T>* _PMessage,  
    _Inout_ ISource<T>* _PSource);
```

## Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

## Return Value

A `message_status` indication of what the target decided to do with the message.

# supports\_anonymous\_source

Overrides the `supports_anonymous_source` method to indicate that this block can accept messages offered to it by a source that is not linked.

```
virtual bool supports_anonymous_source();
```

## Return Value

`true` because the block does not postpone offered messages.

## release\_message

Releases a previous message reservation.

```
virtual void release_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being released.

## reserve\_message

Reserves a message previously offered by this `overwrite_buffer` messaging block.

```
virtual bool reserve_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being reserved.

### Return Value

**true** if the message was successfully reserved, **false** otherwise.

### Remarks

After `reserve` is called, if it returns **true**, either `consume` or `release` must be called to either take or release ownership of the message.

## resume\_propagation

Resumes propagation after a reservation has been released.

```
virtual void resume_propagation();
```

## value

Gets a reference to the current payload of the message being stored in the `overwrite_buffer` messaging block.

```
T value();
```

### Return Value

The payload of the currently stored message.

### Remarks

The value stored in the `overwrite_buffer` could change immediately after this method returns. This method will wait until a message arrives if no message is currently stored in the `overwrite_buffer`.

## See also

[concurrency Namespace](#)

[unbounded\\_buffer Class](#)

single\_assignment Class

# progress\_reporter Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The progress reporter class allows reporting progress notifications of a specific type. Each progress\_reporter object is bound to a particular asynchronous action or operation.

## Syntax

```
template<typename _ProgressType>
class progress_reporter;
```

### Parameters

*\_ProgressType*

The payload type of each progress notification reported through the progress reporter.

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">progress_reporter</a>	

### Public Methods

NAME	DESCRIPTION
<a href="#">report</a>	Sends a progress report to the asynchronous action or operation to which this progress reporter is bound.

## Remarks

This type is only available to Windows Runtime apps.

## Inheritance Hierarchy

```
progress_reporter
```

## Requirements

**Header:** ppltasks.h

**Namespace:** concurrency

## progress\_reporter

```
progress_reporter();
```

# report

Sends a progress report to the asynchronous action or operation to which this progress reporter is bound.

```
void report(const _ProgressType& val) const;
```

## Parameters

*val*

The payload to report through a progress notification.

## See also

[concurrency Namespace](#)

# propagator\_block Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

The `propagator_block` class is an abstract base class for message blocks that are both a source and target. It combines the functionality of both the `source_block` and `target_block` classes.

## Syntax

```
template<class _TargetLinkRegistry, class _SourceLinkRegistry, class _MessageProcessorType =
ordered_message_processor<typename _TargetLinkRegistry::type::type>>
class propagator_block : public source_block<_TargetLinkRegistry,
_MessageProcessorType>,
public ITarget<typename _SourceLinkRegistry::type::source_type>;
```

### Parameters

*\_TargetLinkRegistry*

The link registry to be used for holding the target links.

*\_SourceLinkRegistry*

The link registry to be used for holding the source links.

*\_MessageProcessorType*

The processor type for message processing.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>source_iterator</code>	The type of the iterator for the <code>source_link_manager</code> for this <code>propagator_block</code> .

### Public Constructors

NAME	DESCRIPTION
<code>propagator_block</code>	Constructs a <code>propagator_block</code> object.
<code>~propagator_block</code> Destructor	Destroys a <code>propagator_block</code> object.

### Public Methods

NAME	DESCRIPTION
<code>propagate</code>	Asynchronously passes a message from a source block to this target block.
<code>send</code>	Synchronously initiates a message to this block. Called by an <code>ISource</code> block. When this function completes, the message will already have propagated into the block.

## Protected Methods

NAME	DESCRIPTION
<a href="#">decline_incoming_messages</a>	Indicates to the block that new messages should be declined.
<a href="#">initialize_source_and_target</a>	Initializes the base object. Specifically, the <code>message_processor</code> object needs to be initialized.
<a href="#">link_source</a>	Links a specified source block to this <code>propagator_block</code> object.
<a href="#">process_input_messages</a>	Process input messages. This is only useful for propagator blocks, which derive from <code>source_block</code> (Overrides <code>source_block::process_input_messages</code> .)
<a href="#">propagate_message</a>	When overridden in a derived class, this method asynchronously passes a message from an <code>ISource</code> block to this <code>propagator_block</code> object. It is invoked by the <code>propagate</code> method, when called by a source block.
<a href="#">register_filter</a>	Registers a filter method that will be invoked on every received message.
<a href="#">remove_network_links</a>	Removes all the source and target network links from this <code>propagator_block</code> object.
<a href="#">send_message</a>	When overridden in a derived class, this method synchronously passes a message from an <code>ISource</code> block to this <code>propagator_block</code> object. It is invoked by the <code>send</code> method, when called by a source block.
<a href="#">unlink_source</a>	Unlinks a specified source block from this <code>propagator_block</code> object.
<a href="#">unlink_sources</a>	Unlinks all source blocks from this <code>propagator_block</code> object. (Overrides <code>ITarget::unlink_sources</code> .)

## Remarks

To avoid multiple inheritance, the `propagator_block` class inherits from the `source_block` class and `ITarget` abstract class. Most of the functionality in the `target_block` class is replicated here.

## Inheritance Hierarchy

[ISource](#)

[ITarget](#)

[source\\_block](#)

`propagator_block`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## decline\_incoming\_messages

Indicates to the block that new messages should be declined.

```
void decline_incoming_messages();
```

### Remarks

This method is called by the destructor to ensure that new messages are declined while destruction is in progress.

## initialize\_source\_and\_target

Initializes the base object. Specifically, the `message_processor` object needs to be initialized.

```
void initialize_source_and_target(  
    _Inout_opt_ Scheduler* _PScheduler = NULL,  
    _Inout_opt_ ScheduleGroup* _PScheduleGroup = NULL);
```

### Parameters

*\_PScheduler*

The scheduler to be used for scheduling tasks.

*\_PScheduleGroup*

The schedule group to be used for scheduling tasks.

## link\_source

Links a specified source block to this `propagator_block` object.

```
virtual void link_source(_Inout_ ISource<Source_type>* _PSource);
```

### Parameters

*\_PSource*

A pointer to the `ISource` block that is to be linked.

## process\_input\_messages

Process input messages. This is only useful for propagator blocks, which derive from `source_block`

```
virtual void process_input_messages(_Inout_ message<Target_type>* _PMessage);
```

### Parameters

*\_PMessage*

A pointer to the message that is to be processed.

## propagate

Asynchronously passes a message from a source block to this target block.

```
virtual message_status propagate(  
    _Inout_opt_ message<_Source_type>* _PMessage,  
    _Inout_opt_ ISource<_Source_type>* _PSource);
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A `message_status` indication of what the target decided to do with the message.

### Remarks

The `propagate` method is invoked on a target block by a linked source block. It queues up an asynchronous task to handle the message, if one is not already queued or executing.

The method throws an `invalid_argument` exception if either the `_PMessage` or `_PSource` parameter is `NULL`.

## propagate\_message

When overridden in a derived class, this method asynchronously passes a message from an `ISource` block to this `propagator_block` object. It is invoked by the `propagate` method, when called by a source block.

```
virtual message_status propagate_message(  
    _Inout_ message<_Source_type>* _PMessage,  
    _Inout_ ISource<_Source_type>* _PSource) = 0;
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A `message_status` indication of what the target decided to do with the message.

## propagator\_block

Constructs a `propagator_block` object.

```
propagator_block();
```

## ~propagator\_block

Destroys a `propagator_block` object.

```
virtual ~propagator_block();
```

## register\_filter

Registers a filter method that will be invoked on every received message.

```
void register_filter(filter_method const& _Filter);
```

### Parameters

*\_Filter*

The filter method.

## remove\_network\_links

Removes all the source and target network links from this `propagator_block` object.

```
void remove_network_links();
```

## send

Synchronously initiates a message to this block. Called by an `ISource` block. When this function completes, the message will already have propagated into the block.

```
virtual message_status send(  
    _Inout_ message<_Source_type>* _PMessage,  
    _Inout_ ISource<_Source_type>* _PSource);
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A `message_status` indication of what the target decided to do with the message.

### Remarks

This method throws an `invalid_argument` exception if either the `_PMessage` or `_PSource` parameter is `NULL`.

## send\_message

When overridden in a derived class, this method synchronously passes a message from an `ISource` block to this `propagator_block` object. It is invoked by the `send` method, when called by a source block.

```
virtual message_status send_message(  
    _Inout_ message<_Source_type> *,  
    _Inout_ ISource<_Source_type> *);
```

### Return Value

A `message_status` indication of what the target decided to do with the message.

### Remarks

By default, this block returns `declined` unless overridden by a derived class.

## unlink\_source

Unlinks a specified source block from this `propagator_block` object.

```
virtual void unlink_source(_Inout_ ISource<Source_type>* _PSource);
```

### Parameters

*\_PSource*

A pointer to the `ISource` block that is to be unlinked.

## unlink\_sources

Unlinks all source blocks from this `propagator_block` object.

```
virtual void unlink_sources();
```

## See also

[concurrency Namespace](#)

[source\\_block Class](#)

[ITarget Class](#)

# reader\_writer\_lock Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

A writer-preference queue-based reader-writer lock with local only spinning. The lock grants first in - first out (FIFO) access to writers and starves readers under a continuous load of writers.

## Syntax

```
class reader_writer_lock;
```

## Members

### Public Classes

NAME	DESCRIPTION
<a href="#">reader_writer_lock::scoped_lock Class</a>	An exception safe RAII wrapper that can be used to acquire <code>reader_writer_lock</code> lock objects as a writer.
<a href="#">reader_writer_lock::scoped_lock_read Class</a>	An exception safe RAII wrapper that can be used to acquire <code>reader_writer_lock</code> lock objects as a reader.

### Public Constructors

NAME	DESCRIPTION
<a href="#">reader_writer_lock</a>	Constructs a new <code>reader_writer_lock</code> object.
<a href="#">~reader_writer_lock Destructor</a>	Destroys the <code>reader_writer_lock</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">lock</a>	Acquires the reader-writer lock as a writer.
<a href="#">lock_read</a>	Acquires the reader-writer lock as a reader. If there are writers, active readers have to wait until they are done. The reader simply registers an interest in the lock and waits for writers to release it.
<a href="#">try_lock</a>	Attempts to acquire the reader-writer lock as a writer without blocking.
<a href="#">try_lock_read</a>	Attempts to acquire the reader-writer lock as a reader without blocking.
<a href="#">unlock</a>	Unlocks the reader-writer lock based on who locked it, reader or writer.

## Remarks

For more information, see [Synchronization Data Structures](#).

## Inheritance Hierarchy

```
reader_writer_lock
```

## Requirements

**Header:** concur.h

**Namespace:** concurrency

## lock

Acquires the reader-writer lock as a writer.

```
void lock();
```

### Remarks

It is often safer to utilize the [scoped\\_lock](#) construct to acquire and release a `reader_writer_lock` object as a writer in an exception safe way.

After a writer attempts to acquire the lock, any future readers will block until the writers have successfully acquired and released the lock. This lock is biased towards writers and can starve readers under a continuous load of writers.

Writers are chained so that a writer exiting the lock releases the next writer in line.

If the lock is already held by the calling context, an [improper\\_lock](#) exception will be thrown.

## lock\_read

Acquires the reader-writer lock as a reader. If there are writers, active readers have to wait until they are done. The reader simply registers an interest in the lock and waits for writers to release it.

```
void lock_read();
```

### Remarks

It is often safer to utilize the [scoped\\_lock\\_read](#) construct to acquire and release a `reader_writer_lock` object as a reader in an exception safe way.

If there are writers waiting on the lock, the reader will wait until all writers in line have acquired and released the lock. This lock is biased towards writers and can starve readers under a continuous load of writers.

## reader\_writer\_lock

Constructs a new `reader_writer_lock` object.

```
reader_writer_lock();
```

## ~reader\_writer\_lock

Destroys the `reader_writer_lock` object.

```
~reader_writer_lock();
```

### Remarks

It is expected that the lock is no longer held when the destructor runs. Allowing the reader writer lock to destruct with the lock still held results in undefined behavior.

## reader\_writer\_lock::scoped\_lock Class

An exception safe RAII wrapper that can be used to acquire `reader_writer_lock` lock objects as a writer.

```
class scoped_lock;
```

## scoped\_lock::scoped\_lock

Constructs a `scoped_lock` object and acquires the `reader_writer_lock` object passed in the `_Reader_writer_lock` parameter as a writer. If the lock is held by another thread, this call will block.

```
explicit _CRTIMP scoped_lock(reader_writer_lock& _Reader_writer_lock);
```

### Parameters

*\_Reader\_writer\_lock*

The `reader_writer_lock` object to acquire as a writer.

## scoped\_lock::~~scoped\_lock

Destroys a `reader_writer_lock` object and releases the lock supplied in its constructor.

```
~scoped_lock();
```

## reader\_writer\_lock::scoped\_lock\_read Class

An exception safe RAII wrapper that can be used to acquire `reader_writer_lock` lock objects as a reader.

```
class scoped_lock_read;
```

## try\_lock

Attempts to acquire the reader-writer lock as a writer without blocking.

## scoped\_lock\_read::scoped\_lock\_read

Constructs a `scoped_lock_read` object and acquires the `reader_writer_lock` object passed in the `_Reader_writer_lock` parameter as a reader. If the lock is held by another thread as a writer or there are pending writers, this call will block.

```
explicit _CRTIMP scoped_lock_read(reader_writer_lock& _Reader_writer_lock);
```

#### Parameters

*\_Reader\_writer\_lock*

The `reader_writer_lock` object to acquire as a reader.

Destroys a `scoped_lock_read` object and releases the lock supplied in its constructor.

```
~scoped_lock_read();
```

## try\_lock

```
bool try_lock();
```

#### Return Value

If the lock was acquired, the value **true**; otherwise, the value **false**.

## try\_lock\_read

Attempts to acquire the reader-writer lock as a reader without blocking.

```
bool try_lock_read();
```

#### Return Value

If the lock was acquired, the value **true**; otherwise, the value **false**.

## unlock

Unlocks the reader-writer lock based on who locked it, reader or writer.

```
void unlock();
```

#### Remarks

If there are writers waiting on the lock, the release of the lock will always go to the next writer in FIFO order. This lock is biased towards writers and can starve readers under a continuous load of writers.

## See also

[concurrency Namespace](#)

[critical\\_section Class](#)

# ScheduleGroup Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents an abstraction for a schedule group. Schedule groups organize a set of related work that benefits from being scheduled close together either temporally, by executing another task in the same group before moving to another group, or spatially, by executing multiple items within the same group on the same NUMA node or physical socket.

## Syntax

```
class ScheduleGroup;
```

## Members

### Protected Constructors

NAME	DESCRIPTION
<a href="#">~ScheduleGroup Destructor</a>	

### Public Methods

NAME	DESCRIPTION
<a href="#">Id</a>	Returns an identifier for the schedule group that is unique within the scheduler to which the group belongs.
<a href="#">Reference</a>	Increments the schedule group reference count.
<a href="#">Release</a>	Decrements the scheduler group reference count.
<a href="#">ScheduleTask</a>	Schedules a light-weight task within the schedule group.

## Inheritance Hierarchy

`ScheduleGroup`

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## Id

Returns an identifier for the schedule group that is unique within the scheduler to which the group belongs.

```
virtual unsigned int Id() const = 0;
```

## Return Value

An identifier for the schedule group that is unique within the scheduler to which the group belongs.

# operator delete

A `ScheduleGroup` object is destroyed internally by the runtime when all external references to it are released. It cannot be explicitly deleted.

```
void operator delete(
    void* _PObject);

void operator delete(
    void* _PObject,
    int,
    const char *,
    int);
```

## Parameters

*\_PObject*

A pointer to the object to be deleted.

# Reference

Increments the schedule group reference count.

```
virtual unsigned int Reference() = 0;
```

## Return Value

The newly incremented reference count.

## Remarks

This is typically used to manage the lifetime of the schedule group for composition. When the reference count of a schedule group falls to zero, the schedule group is deleted by the runtime. A schedule group created using either the `CurrentScheduler::CreateScheduleGroup` method, or the `Scheduler::CreateScheduleGroup` method starts out with a reference count of one.

# Release

Decrements the scheduler group reference count.

```
virtual unsigned int Release() = 0;
```

## Return Value

The newly decremented reference count.

## Remarks

This is typically used to manage the lifetime of the schedule group for composition. When the reference count of a schedule group falls to zero, the schedule group is deleted by the runtime. After you have called the `Release` method the specific number of times to remove the creation reference count and any additional references placed using the `Reference` method, you cannot utilize the schedule group further. Doing so will result in undefined behavior.

A schedule group is associated with a particular scheduler instance. You must ensure that all references to the

schedule group are released before all references to the scheduler are released, because the latter could result in the scheduler being destroyed. Doing otherwise results in undefined behavior.

## ~ScheduleGroup

```
virtual ~ScheduleGroup();
```

## ScheduleTask

Schedules a light-weight task within the schedule group.

```
virtual void ScheduleTask(  
    TaskProc _Proc,  
    _Inout_opt_ void* _Data) = 0;
```

### Parameters

*\_Proc*

A pointer to the function to execute to perform the body of the light-weight task.

*\_Data*

A void pointer to the data that will be passed as a parameter to the body of the task.

### Remarks

Calling the `ScheduleTask` method implicitly places a reference count on the schedule group which is removed by the runtime at an appropriate time after the task executes.

## See also

[concurrency Namespace](#)

[CurrentScheduler Class](#)

[Scheduler Class](#)

[Task Scheduler](#)

# Scheduler Class

3/4/2019 • 8 minutes to read • [Edit Online](#)

Represents an abstraction for a Concurrency Runtime scheduler.

## Syntax

```
class Scheduler;
```

## Members

### Protected Constructors

NAME	DESCRIPTION
<a href="#">Scheduler</a>	An object of the <code>Scheduler</code> class can only be created using factory methods, or implicitly.
<a href="#">~Scheduler Destructor</a>	An object of the <code>Scheduler</code> class is implicitly destroyed when all external references to it cease to exist.

### Public Methods

NAME	DESCRIPTION
<a href="#">Attach</a>	Attaches the scheduler to the calling context. After this method returns, the calling context is managed by the scheduler and the scheduler becomes the current scheduler.
<a href="#">Create</a>	Creates a new scheduler whose behavior is described by the <code>_Policy</code> parameter, places an initial reference on the scheduler, and returns a pointer to it.
<a href="#">CreateScheduleGroup</a>	Overloaded. Creates a new schedule group within the scheduler. The version that takes the parameter <code>_Placement</code> causes tasks within the newly created schedule group to be biased towards executing at the location specified by that parameter.
<a href="#">GetNumberOfVirtualProcessors</a>	Returns the current number of virtual processors for the scheduler.
<a href="#">GetPolicy</a>	Returns a copy of the policy that the scheduler was created with.
<a href="#">Id</a>	Returns a unique identifier for the scheduler.
<a href="#">IsAvailableLocation</a>	Determines whether a given location is available on the scheduler.

NAME	DESCRIPTION
<a href="#">Reference</a>	Increments the scheduler reference count.
<a href="#">RegisterShutdownEvent</a>	Causes the Windows event handle passed in the <code>_Event</code> parameter to be signaled when the scheduler shuts down and destroys itself. At the time the event is signaled, all work that had been scheduled to the scheduler is complete. Multiple shutdown events can be registered through this method.
<a href="#">Release</a>	Decrements the scheduler reference count.
<a href="#">ResetDefaultSchedulerPolicy</a>	Resets the default scheduler policy to the runtime default. The next time a default scheduler is created, it will use the runtime default policy settings.
<a href="#">ScheduleTask</a>	Overloaded. Schedules a light-weight task within the scheduler. The light-weight task will be placed in a schedule group determined by the runtime. The version that takes the parameter <code>_Placement</code> causes the task to be biased towards executing at the specified location.
<a href="#">SetDefaultSchedulerPolicy</a>	Allows a user defined policy to be used to create the default scheduler. This method can be called only when no default scheduler exists within the process. After a default policy has been set, it remains in effect until the next valid call to either the <code>SetDefaultSchedulerPolicy</code> or the <a href="#">ResetDefaultSchedulerPolicy</a> method.

## Remarks

The Concurrency Runtime scheduler uses execution contexts, which map to the operating system execution contexts, such as a thread, to execute the work queued to it by your application. At any time, the concurrency level of a scheduler is equal to the number of virtual processor granted to it by the Resource Manager. A virtual processor is an abstraction for a processing resource and maps to a hardware thread on the underlying system. Only a single scheduler context can execute on a virtual processor at a given time.

The Concurrency Runtime will create a default scheduler per process to execute parallel work. In addition you can create your own scheduler instances and manipulate it using this class.

## Inheritance Hierarchy

`Scheduler`

## Requirements

**Header:** `concr.t.h`

**Namespace:** `concurrency`

## Attach

Attaches the scheduler to the calling context. After this method returns, the calling context is managed by the scheduler and the scheduler becomes the current scheduler.

```
virtual void Attach() = 0;
```

### Remarks

Attaching a scheduler implicitly places a reference on the scheduler.

At some point in the future, you must call the [CurrentScheduler::Detach](#) method in order to allow the scheduler to shut down.

If this method is called from a context that is already attached to a different scheduler, the existing scheduler is remembered as the previous scheduler, and the newly created scheduler becomes the current scheduler. When you call the `CurrentScheduler::Detach` method at a later point, the previous scheduler is restored as the current scheduler.

This method will throw an [improper\\_scheduler\\_attach](#) exception if this scheduler is the current scheduler of the calling context.

## Create

Creates a new scheduler whose behavior is described by the `_Policy` parameter, places an initial reference on the scheduler, and returns a pointer to it.

```
static Scheduler* __cdecl Create(const SchedulerPolicy& _Policy);
```

### Parameters

*\_Policy*

The scheduler policy that describes behavior of the newly created scheduler.

### Return Value

A pointer to a newly created scheduler. This `Scheduler` object has an initial reference count placed on it.

### Remarks

After a scheduler is created with the `Create` method, you must call the `Release` method at some point in the future in order to remove the initial reference count and allow the scheduler to shut down.

A scheduler created with this method is not attached to the calling context. It can be attached to a context using the [Attach](#) method.

This method can throw a variety of exceptions, including [scheduler\\_resource\\_allocation\\_error](#) and [invalid\\_scheduler\\_policy\\_value](#).

## CreateScheduleGroup

Creates a new schedule group within the scheduler. The version that takes the parameter `_Placement` causes tasks within the newly created schedule group to be biased towards executing at the location specified by that parameter.

```
virtual ScheduleGroup* CreateScheduleGroup() = 0;  
  
virtual ScheduleGroup* CreateScheduleGroup(location& _Placement) = 0;
```

### Parameters

*\_Placement*

A reference to a location where the tasks within the schedule group will be biased towards executing at.

### Return Value

A pointer to the newly created schedule group. This `ScheduleGroup` object has an initial reference count placed on it.

### Remarks

You must invoke the [Release](#) method on a schedule group when you are done scheduling work to it. The scheduler will destroy the schedule group when all work queued to it has completed.

Note that if you explicitly created this scheduler, you must release all references to schedule groups within it, before you release your references on the scheduler.

## GetNumberOfVirtualProcessors

Returns the current number of virtual processors for the scheduler.

```
virtual unsigned int GetNumberOfVirtualProcessors() const = 0;
```

### Return Value

The current number of virtual processors for the scheduler.

## GetPolicy

Returns a copy of the policy that the scheduler was created with.

```
virtual SchedulerPolicy GetPolicy() const = 0;
```

### Return Value

A copy of the policy that the scheduler was created with.

## Id

Returns a unique identifier for the scheduler.

```
virtual unsigned int Id() const = 0;
```

### Return Value

A unique identifier for the scheduler.

## IsAvailableLocation

Determines whether a given location is available on the scheduler.

```
virtual bool IsAvailableLocation(const location& _Placement) const = 0;
```

### Parameters

*\_Placement*

A reference to the location to query the scheduler about.

### Return Value

An indication of whether or not the location specified by the `_Placement` argument is available on the scheduler.

## Remarks

Note that the return value is an instantaneous sampling of whether the given location is available. In the presence of multiple schedulers, dynamic resource management can add or take away resources from schedulers at any point. Should this happen, the given location can change availability.

## Reference

Increments the scheduler reference count.

```
virtual unsigned int Reference() = 0 ;
```

## Return Value

The newly incremented reference count.

## Remarks

This is typically used to manage the lifetime of the scheduler for composition. When the reference count of a scheduler falls to zero, the scheduler will shut down and destruct itself after all work on the scheduler has completed.

The method will throw an [improper\\_scheduler\\_reference](#) exception if the reference count prior to calling the `Reference` method was zero and the call is made from a context that is not owned by the scheduler.

## RegisterShutdownEvent

Causes the Windows event handle passed in the `_Event` parameter to be signaled when the scheduler shuts down and destroys itself. At the time the event is signaled, all work that had been scheduled to the scheduler is complete. Multiple shutdown events can be registered through this method.

```
virtual void RegisterShutdownEvent(HANDLE _Event) = 0;
```

## Parameters

`_Event`

A handle to a Windows event object which will be signaled by the runtime when the scheduler shuts down and destroys itself.

## Release

Decrements the scheduler reference count.

```
virtual unsigned int Release() = 0;
```

## Return Value

The newly decremented reference count.

## Remarks

This is typically used to manage the lifetime of the scheduler for composition. When the reference count of a scheduler falls to zero, the scheduler will shut down and destruct itself after all work on the scheduler has completed.

## ResetDefaultSchedulerPolicy

Resets the default scheduler policy to the runtime default. The next time a default scheduler is created, it will use the runtime default policy settings.

```
static void __cdecl ResetDefaultSchedulerPolicy();
```

### Remarks

This method can be called while a default scheduler exists within the process. It will not affect the policy of the existing default scheduler. However, if the default scheduler were to shutdown, and a new default were to be created at a later point, the new scheduler would use the runtime default policy settings.

## Scheduler

An object of the `Scheduler` class can only be created using factory methods, or implicitly.

```
Scheduler();
```

### Remarks

The process' default scheduler is created implicitly when you utilize many of the runtime functions which require a scheduler to be attached to the calling context. Methods within the `CurrentScheduler` class and features of the PPL and agents layers typically perform implicit attachment.

You can also create a scheduler explicitly through either the `CurrentScheduler::Create` method or the `Scheduler::Create` method.

## ~Scheduler

An object of the `Scheduler` class is implicitly destroyed when all external references to it cease to exist.

```
virtual ~Scheduler();
```

## ScheduleTask

Schedules a light-weight task within the scheduler. The light-weight task will be placed in a schedule group determined by the runtime. The version that takes the parameter `_Placement` causes the task to be biased towards executing at the specified location.

```
virtual void ScheduleTask(  
    TaskProc _Proc,  
    _Inout_opt_ void* _Data) = 0;  
  
virtual void ScheduleTask(  
    TaskProc _Proc,  
    _Inout_opt_ void* _Data,  
    location& _Placement) = 0;
```

### Parameters

*\_Proc*

A pointer to the function to execute to perform the body of the light-weight task.

*\_Data*

A void pointer to the data that will be passed as a parameter to the body of the task.

*\_Placement*

A reference to a location where the light-weight task will be biased towards executing at.

## SetDefaultSchedulerPolicy

Allows a user defined policy to be used to create the default scheduler. This method can be called only when no default scheduler exists within the process. After a default policy has been set, it remains in effect until the next valid call to either the `SetDefaultSchedulerPolicy` or the [ResetDefaultSchedulerPolicy](#) method.

```
static void __cdecl SetDefaultSchedulerPolicy(const SchedulerPolicy& _Policy);
```

### Parameters

*\_Policy*

The policy to be set as the default scheduler policy.

### Remarks

If the `SetDefaultSchedulerPolicy` method is called when a default scheduler already exists within the process, the runtime will throw a [default\\_scheduler\\_exists](#) exception.

## See also

[concurrency Namespace](#)

[Scheduler Class](#)

[PolicyElementKey](#)

[Task Scheduler](#)

# scheduler\_interface Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

Scheduler Interface

## Syntax

```
struct __declspec(novtable) scheduler_interface;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">scheduler_interface::schedule</a>	

## Inheritance Hierarchy

`scheduler_interface`

## Requirements

**Header:** pplinterface.h

**Namespace:** concurrency

## scheduler\_interface::schedule Method

```
virtual void schedule(  
    TaskProc_t,  
    void*) = 0;
```

## See also

[concurrency Namespace](#)

# scheduler\_not\_attached Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when an operation is performed which requires a scheduler to be attached to the current context and one is not.

## Syntax

```
class scheduler_not_attached : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">scheduler_not_attached</a>	Overloaded. Constructs a <code>scheduler_not_attached</code> object.

## Inheritance Hierarchy

`exception`

`scheduler_not_attached`

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## scheduler\_not\_attached

Constructs a `scheduler_not_attached` object.

```
explicit _CRTIMP scheduler_not_attached(_In_z_ const char* _Message) throw();  
scheduler_not_attached() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)  
[Scheduler Class](#)

# scheduler\_ptr Structure

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents a pointer to a scheduler. This class exists to allow the specification of a shared lifetime by using `shared_ptr` or just a plain reference by using raw pointer.

## Syntax

```
struct scheduler_ptr;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">scheduler_ptr::scheduler_ptr</a>	Overloaded. Creates a scheduler pointer from <code>shared_ptr</code> to scheduler

### Public Methods

NAME	DESCRIPTION
<a href="#">scheduler_ptr::get</a>	Returns the raw pointer to the scheduler

### Public Operators

NAME	DESCRIPTION
<a href="#">scheduler_ptr::operator bool</a>	Test whether the scheduler pointer is non-null
<a href="#">scheduler_ptr::operator-&gt;</a>	Behave like a pointer

## Inheritance Hierarchy

```
scheduler_ptr
```

## Requirements

**Header:** `pplinterface.h`

**Namespace:** `concurrency`

## scheduler\_ptr::get Method

Returns the raw pointer to the scheduler.

```
scheduler_interface* get() const;
```

## Return Value

### scheduler\_ptr::operator bool

Tests whether the scheduler pointer is non-null.

```
operator bool() const;
```

### scheduler\_ptr::operator->

Behaves like a pointer.

```
scheduler_interface* operator->() const;
```

## Return Value

### scheduler\_ptr::scheduler\_ptr Constructor

Creates a scheduler pointer from shared\_ptr to scheduler.

```
explicit scheduler_ptr(std::shared_ptr<scheduler_interface> scheduler);  
explicit scheduler_ptr(_In_opt_ scheduler_interface* pScheduler);
```

## Parameters

*scheduler*

The scheduler to convert.

*pScheduler*

The scheduler pointer to convert.

## See also

[concurrency Namespace](#)

# scheduler\_resource\_allocation\_error Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown because of a failure to acquire a critical resource in the Concurrency Runtime.

## Syntax

```
class scheduler_resource_allocation_error : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">scheduler_resource_allocation_error</a>	Overloaded. Constructs a <code>scheduler_resource_allocation_error</code> object.

### Public Methods

NAME	DESCRIPTION
<a href="#">get_error_code</a>	Returns the error code that caused the exception.

## Remarks

This exception is typically thrown when a call to the operating system from within the Concurrency Runtime fails. The error code which would normally be returned from a call to the Win32 method `GetLastError` is converted to a value of type `HRESULT` and can be retrieved using the `get_error_code` method.

## Inheritance Hierarchy

`exception`

`scheduler_resource_allocation_error`

## Requirements

**Header:** `concrth`

**Namespace:** `concurrency`

## get\_error\_code

Returns the error code that caused the exception.

```
HRESULT get_error_code() const throw();
```

## Return Value

The `HRESULT` value of the error that caused the exception.

# scheduler\_resource\_allocation\_error

Constructs a `scheduler_resource_allocation_error` object.

```
scheduler_resource_allocation_error(  
    _In_z_ const char* _Message,  
    HRESULT _Hresult) throw();  
  
explicit _CRTIMP scheduler_resource_allocation_error(  
    HRESULT _Hresult) throw();
```

## Parameters

*\_Message*

A descriptive message of the error.

*\_Hresult*

The `HRESULT` value of the error that caused the exception.

## See also

[concurrency Namespace](#)

# scheduler\_worker\_creation\_error Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown because of a failure to create a worker execution context in the Concurrency Runtime.

## Syntax

```
class scheduler_worker_creation_error : public scheduler_resource_allocation_error;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">scheduler_worker_creation_error</a>	Overloaded. Constructs a <code>scheduler_worker_creation_error</code> object.

## Remarks

This exception is typically thrown when a call to the operating system to create execution contexts from within the Concurrency Runtime fails. Execution contexts are threads that execute tasks in the Concurrency Runtime. The error code which would normally be returned from a call to the Win32 method `GetLastError` is converted to a value of type `HRESULT` and can be retrieved using the base class method `get_error_code`.

## Inheritance Hierarchy

`exception`

`scheduler_resource_allocation_error`

`scheduler_worker_creation_error`

## Requirements

**Header:** `concr.h`

**Namespace:** `concurrency`

## scheduler\_worker\_creation\_error

Constructs a `scheduler_worker_creation_error` object.

```
scheduler_worker_creation_error(  
    _In_z_ const char* _Message,  
    HRESULT _Hresult) throw();  
  
explicit _CRTIMP scheduler_worker_creation_error(  
    HRESULT _Hresult) throw();
```

## Parameters

*\_Message*

A descriptive message of the error.

*\_Hresult*

The `HRESULT` value of the error that caused the exception.

## See also

[concurrency Namespace](#)

# SchedulerPolicy Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

The `SchedulerPolicy` class contains a set of key/value pairs, one for each policy element, that control the behavior of a scheduler instance.

## Syntax

```
class SchedulerPolicy;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">SchedulerPolicy</a>	Overloaded. Constructs a new scheduler policy and populates it with values for <a href="#">policy keys</a> supported by Concurrency Runtime schedulers and the Resource Manager.
<a href="#">~SchedulerPolicy</a> Destructor	Destroys a scheduler policy.

### Public Methods

NAME	DESCRIPTION
<a href="#">GetPolicyValue</a>	Retrieves the value of the policy key supplied as the <code>key</code> parameter.
<a href="#">SetConcurrencyLimits</a>	Simultaneously sets the <code>MinConcurrency</code> and <code>MaxConcurrency</code> policies on the <code>SchedulerPolicy</code> object.
<a href="#">SetPolicyValue</a>	Sets the value of the policy key supplied as the <code>key</code> parameter and returns the old value.

### Public Operators

NAME	DESCRIPTION
<a href="#">operator=</a>	Assigns the scheduler policy from another scheduler policy.

## Remarks

For more information about the policies which can be controlled using the `SchedulerPolicy` class, see [PolicyElementKey](#).

## Inheritance Hierarchy

`SchedulerPolicy`

# Requirements

**Header:** `concrth`, `concrtrm.h`

**Namespace:** `concurrency`

## GetPolicyValue

Retrieves the value of the policy key supplied as the `key` parameter.

```
unsigned int GetPolicyValue(PolicyElementKey key) const;
```

### Parameters

*key*

The policy key to retrieve a value for.

### Return Value

If the key specified by the `key` parameter is supported, the policy value for the key cast to an `unsigned int`.

### Remarks

The method will throw `invalid_scheduler_policy_key` for an invalid policy key.

## operator=

Assigns the scheduler policy from another scheduler policy.

```
SchedulerPolicy& operator= (const SchedulerPolicy& _RhsPolicy);
```

### Parameters

*\_RhsPolicy*

The policy to assign to this policy.

### Return Value

A reference to the scheduler policy.

### Remarks

Often, the most convenient way to define a new scheduler policy is to copy an existing policy and modify it using the `SetPolicyValue` or `SetConcurrencyLimits` methods.

## SchedulerPolicy

Constructs a new scheduler policy and populates it with values for [policy keys](#) supported by Concurrency Runtime schedulers and the Resource Manager.

```
SchedulerPolicy();

SchedulerPolicy(
    size_t _PolicyKeyCount,
    ...);

SchedulerPolicy(
    const SchedulerPolicy& _SrcPolicy);
```

### Parameters

## *\_PolicyKeyCount*

The number of key/value pairs that follow the `_PolicyKeyCount` parameter.

## *\_SrcPolicy*

The source policy to copy.

### Remarks

The first constructor creates a new scheduler policy where all policies will be initialized to their default values.

The second constructor creates a new scheduler policy that uses a named-parameter style of initialization. Values after the `_PolicyKeyCount` parameter are supplied as key/value pairs. Any policy key which is not specified in this constructor will have its default value. This constructor could throw the exceptions [invalid\\_scheduler\\_policy\\_key](#), [invalid\\_scheduler\\_policy\\_value](#) or [invalid\\_scheduler\\_policy\\_thread\\_specification](#).

The third constructor is a copy constructor. Often, the most convenient way to define a new scheduler policy is to copy an existing policy and modify it using the `SetPolicyValue` or `SetConcurrencyLimits` methods.

## ~SchedulerPolicy

Destroys a scheduler policy.

```
~SchedulerPolicy();
```

## SetConcurrencyLimits

Simultaneously sets the `MinConcurrency` and `MaxConcurrency` policies on the `SchedulerPolicy` object.

```
void SetConcurrencyLimits(  
    unsigned int _MinConcurrency,  
    unsigned int _MaxConcurrency = MaxExecutionResources);
```

### Parameters

#### *\_MinConcurrency*

The value for the `MinConcurrency` policy key.

#### *\_MaxConcurrency*

The value for the `MaxConcurrency` policy key.

### Remarks

The method will throw [invalid\\_scheduler\\_policy\\_thread\\_specification](#) if the value specified for the `MinConcurrency` policy is greater than that specified for the `MaxConcurrency` policy.

The method can also throw [invalid\\_scheduler\\_policy\\_value](#) for other invalid values.

## SetPolicyValue

Sets the value of the policy key supplied as the `key` parameter and returns the old value.

```
unsigned int SetPolicyValue(  
    PolicyElementKey key,  
    unsigned int value);
```

### Parameters

*key*

The policy key to set a value for.

*value*

The value to set the policy key to.

### Return Value

If the key specified by the `key` parameter is supported, the old policy value for the key cast to an `unsigned int`.

### Remarks

The method will throw `invalid_scheduler_policy_key` for an invalid policy key or any policy key whose value cannot be set by the `SetPolicyValue` method.

The method will throw `invalid_scheduler_policy_value` for a value that is not supported for the key specified by the `key` parameter.

Note that this method is not allowed to set the `MinConcurrency` or `MaxConcurrency` policies. To set these values, use the `SetConcurrencyLimits` method.

## See also

[concurrency Namespace](#)

[PolicyElementKey](#)

[CurrentScheduler Class](#)

[Scheduler Class](#)

[Task Scheduler](#)

# simple\_partitioner Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `simple_partitioner` class represents a static partitioning of the range iterated over by `parallel_for`. The partitioner divides the range into chunks such that each chunk has at least the number of iterations specified by the chunk size.

## Syntax

```
class simple_partitioner;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<code>simple_partitioner</code>	Constructs a <code>simple_partitioner</code> object.
<code>~simple_partitioner</code> Destructor	Destroys a <code>simple_partitioner</code> object.

## Inheritance Hierarchy

```
simple_partitioner
```

## Requirements

**Header:** `ppl.h`

**Namespace:** `concurrency`

### `~simple_partitioner`

Destroys a `simple_partitioner` object.

```
~simple_partitioner();
```

### `simple_partitioner`

Constructs a `simple_partitioner` object.

```
explicit simple_partitioner(_Size_type _Chunk_size);
```

### Parameters

`_Chunk_size`

The minimum partition size.

## See also

[concurrency Namespace](#)

# single\_assignment Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

A `single_assignment` messaging block is a multi-target, multi-source, ordered `propagator_block` capable of storing a single, write-once `message`.

## Syntax

```
template<class T>
class single_assignment : public propagator_block<multi_link_registry<ITarget<T>>,
multi_link_registry<ISource<T>>>>;
```

### Parameters

*T*

The payload type of the message stored and propagated by the buffer.

## Members

### Public Constructors

NAME	DESCRIPTION
<code>single_assignment</code>	Overloaded. Constructs a <code>single_assignment</code> messaging block.
<code>~single_assignment</code> Destructor	Destroys the <code>single_assignment</code> messaging block.

### Public Methods

NAME	DESCRIPTION
<code>has_value</code>	Checks whether this <code>single_assignment</code> messaging block has been initialized with a value yet.
<code>value</code>	Gets a reference to the current payload of the message being stored in the <code>single_assignment</code> messaging block.

### Protected Methods

NAME	DESCRIPTION
<code>accept_message</code>	Accepts a message that was offered by this <code>single_assignment</code> messaging block, returning a copy of the message to the caller.
<code>consume_message</code>	Consumes a message previously offered by the <code>single_assignment</code> and reserved by the target, returning a copy of the message to the caller.

NAME	DESCRIPTION
<a href="#">link_target_notification</a>	A callback that notifies that a new target has been linked to this <code>single_assignment</code> messaging block.
<a href="#">propagate_message</a>	Asynchronously passes a message from an <code>ISource</code> block to this <code>single_assignment</code> messaging block. It is invoked by the <code>propagate</code> method, when called by a source block.
<a href="#">propagate_to_any_targets</a>	Places the <code>message _PMessage</code> in this <code>single_assignment</code> messaging block and offers it to all of the linked targets.
<a href="#">release_message</a>	Releases a previous message reservation. (Overrides <code>source_block::release_message</code> .)
<a href="#">reserve_message</a>	Reserves a message previously offered by this <code>single_assignment</code> messaging block. (Overrides <code>source_block::reserve_message</code> .)
<a href="#">resume_propagation</a>	Resumes propagation after a reservation has been released. (Overrides <code>source_block::resume_propagation</code> .)
<a href="#">send_message</a>	Synchronously passes a message from an <code>ISource</code> block to this <code>single_assignment</code> messaging block. It is invoked by the <code>send</code> method, when called by a source block.

## Remarks

A `single_assignment` messaging block propagates out copies of its message to each target.

For more information, see [Asynchronous Message Blocks](#).

## Inheritance Hierarchy

[ISource](#)

[ITarget](#)

[source\\_block](#)

[propagator\\_block](#)

`single_assignment`

## Requirements

**Header:** `agents.h`

**Namespace:** `concurrency`

## `accept_message`

Accepts a message that was offered by this `single_assignment` messaging block, returning a copy of the message to the caller.

```
virtual message<T>* accept_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the offered `message` object.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

The `single_assignment` messaging block returns copies of the message to its targets, rather than transferring ownership of the currently held message.

## consume\_message

Consumes a message previously offered by the `single_assignment` and reserved by the target, returning a copy of the message to the caller.

```
virtual message<T>* consume_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being consumed.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

Similar to `accept`, but is always preceded by a call to `reserve`.

## has\_value

Checks whether this `single_assignment` messaging block has been initialized with a value yet.

```
bool has_value() const;
```

### Return Value

**true** if the block has received a value, **false** otherwise.

## link\_target\_notification

A callback that notifies that a new target has been linked to this `single_assignment` messaging block.

```
virtual void link_target_notification(_Inout_ ITarget<T>* _PTarget);
```

### Parameters

*\_PTarget*

A pointer to the newly linked target.

## propagate\_message

Asynchronously passes a message from an `ISource` block to this `single_assignment` messaging block. It is invoked by the `propagate` method, when called by a source block.

```
virtual message_status propagate_message(  
    _Inout_ message<T>* _PMessage,  
    _Inout_ ISource<T>* _PSource);
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A `message_status` indication of what the target decided to do with the message.

## propagate\_to\_any\_targets

Places the `message` `_PMessage` in this `single_assignment` messaging block and offers it to all of the linked targets.

```
virtual void propagate_to_any_targets(_Inout_opt_ message<T>* _PMessage);
```

### Parameters

*\_PMessage*

A pointer to a `message` that this `single_assignment` messaging block has taken ownership of.

## release\_message

Releases a previous message reservation.

```
virtual void release_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being released.

## reserve\_message

Reserves a message previously offered by this `single_assignment` messaging block.

```
virtual bool reserve_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being reserved.

### Return Value

**true** if the message was successfully reserved, **false** otherwise.

### Remarks

After `reserve` is called, if it returns **true**, either `consume` or `release` must be called to either take or release ownership of the message.

## resume\_propagation

Resumes propagation after a reservation has been released.

```
virtual void resume_propagation();
```

## send\_message

Synchronously passes a message from an `ISource` block to this `single_assignment` messaging block. It is invoked by the `send` method, when called by a source block.

```
virtual message_status send_message(  
    _Inout_ message<T>* _PMessage,  
    _Inout_ ISource<T>* _PSource);
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A `message_status` indication of what the target decided to do with the message.

## single\_assignment

Constructs a `single_assignment` messaging block.

```
single_assignment();  
  
single_assignment(  
    filter_method const& _Filter);  
  
single_assignment(  
    Scheduler& _PScheduler);  
  
single_assignment(  
    Scheduler& _PScheduler,  
    filter_method const& _Filter);  
  
single_assignment(  
    ScheduleGroup& _PScheduleGroup);  
  
single_assignment(  
    ScheduleGroup& _PScheduleGroup,  
    filter_method const& _Filter);
```

### Parameters

*\_Filter*

A filter function which determines whether offered messages should be accepted.

#### `_PScheduler`

The `Scheduler` object within which the propagation task for the `single_assignment` messaging block is scheduled.

#### `_PScheduleGroup`

The `ScheduleGroup` object within which the propagation task for the `single_assignment` messaging block is scheduled. The `Scheduler` object used is implied by the schedule group.

#### Remarks

The runtime uses the default scheduler if you do not specify the `_PScheduler` or `_PScheduleGroup` parameters.

The type `filter_method` is a functor with signature `bool (T const &)` which is invoked by this `single_assignment` messaging block to determine whether or not it should accept an offered message.

## `~single_assignment`

Destroys the `single_assignment` messaging block.

```
~single_assignment();
```

## value

Gets a reference to the current payload of the message being stored in the `single_assignment` messaging block.

```
T const& value();
```

#### Return Value

The payload of the stored message.

#### Remarks

This method will wait until a message arrives if no message is currently stored in the `single_assignment` messaging block.

## See also

[concurrency Namespace](#)

[overwrite\\_buffer Class](#)

[unbounded\\_buffer Class](#)

# single\_link\_registry Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `single_link_registry` object is a `network_link_registry` that manages only a single source or target block.

## Syntax

```
template<class _Block>
class single_link_registry : public network_link_registry<_Block>;
```

### Parameters

`_Block`

The block data type being stored in the `single_link_registry` object.

## Members

### Public Constructors

NAME	DESCRIPTION
<code>single_link_registry</code>	Constructs a <code>single_link_registry</code> object.
<code>~single_link_registry</code> Destructor	Destroys the <code>single_link_registry</code> object.

### Public Methods

NAME	DESCRIPTION
<code>add</code>	Adds a link to the <code>single_link_registry</code> object. (Overrides <code>network_link_registry::add</code> .)
<code>begin</code>	Returns an iterator to the first element in the <code>single_link_registry</code> object. (Overrides <code>network_link_registry::begin</code> .)
<code>contains</code>	Searches the <code>single_link_registry</code> object for a specified block. (Overrides <code>network_link_registry::contains</code> .)
<code>count</code>	Counts the number of items in the <code>single_link_registry</code> object. (Overrides <code>network_link_registry::count</code> .)
<code>remove</code>	Removes a link from the <code>single_link_registry</code> object. (Overrides <code>network_link_registry::remove</code> .)

## Inheritance Hierarchy

`network_link_registry`

`single_link_registry`

# Requirements

**Header:** agents.h

**Namespace:** concurrency

## add

Adds a link to the `single_link_registry` object.

```
virtual void add(_EType _Link);
```

### Parameters

*\_Link*

A pointer to a block to be added.

### Remarks

The method throws an `invalid_link_target` exception if there is already a link in this registry.

## begin

Returns an iterator to the first element in the `single_link_registry` object.

```
virtual iterator begin();
```

### Return Value

An iterator addressing the first element in the `single_link_registry` object.

### Remarks

The end state is indicated by a `NULL` link.

## contains

Searches the `single_link_registry` object for a specified block.

```
virtual bool contains(_EType _Link);
```

### Parameters

*\_Link*

A pointer to a block that is to be searched for in the `single_link_registry` object.

### Return Value

**true** if the link was found, **false** otherwise.

## count

Counts the number of items in the `single_link_registry` object.

```
virtual size_t count();
```

### Return Value

The number of items in the `single_link_registry` object.

## remove

Removes a link from the `single_link_registry` object.

```
virtual bool remove(_EType _Link);
```

### Parameters

*\_Link*

A pointer to a block to be removed, if found.

### Return Value

**true** if the link was found and removed, **false** otherwise.

## single\_link\_registry

Constructs a `single_link_registry` object.

```
single_link_registry();
```

## ~single\_link\_registry

Destroys the `single_link_registry` object.

```
virtual ~single_link_registry();
```

### Remarks

The method throws an [invalid\\_operation](#) exception if it is called before the link is removed.

## See also

[concurrency Namespace](#)

[multi\\_link\\_registry Class](#)

# source\_block Class

3/4/2019 • 8 minutes to read • [Edit Online](#)

The `source_block` class is an abstract base class for source-only blocks. The class provides basic link management functionality as well as common error checks.

## Syntax

```
template<class _TargetLinkRegistry, class _MessageProcessorType = ordered_message_processor<typename
_TargetLinkRegistry::type::type>>
class source_block : public ISource<typename _TargetLinkRegistry::type::type>;
```

### Parameters

*\_TargetLinkRegistry*

Link registry to be used for holding the target links.

*\_MessageProcessorType*

Processor type for message processing.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>target_iterator</code>	The iterator to walk the connected targets.

### Public Constructors

NAME	DESCRIPTION
<code>source_block</code>	Constructs a <code>source_block</code> object.
<code>~source_block</code> Destructor	Destroys the <code>source_block</code> object.

### Public Methods

NAME	DESCRIPTION
<code>accept</code>	Accepts a message that was offered by this <code>source_block</code> object, transferring ownership to the caller.
<code>acquire_ref</code>	Acquires a reference count on this <code>source_block</code> object, to prevent deletion.
<code>consume</code>	Consumes a message previously offered by this <code>source_block</code> object and successfully reserved by the target, transferring ownership to the caller.
<code>link_target</code>	Links a target block to this <code>source_block</code> object.

NAME	DESCRIPTION
<a href="#">release</a>	Releases a previous successful message reservation.
<a href="#">release_ref</a>	Releases a reference count on this <code>source_block</code> object.
<a href="#">reserve</a>	Reserves a message previously offered by this <code>source_block</code> object.
<a href="#">unlink_target</a>	Unlinks a target block from this <code>source_block</code> object.
<a href="#">unlink_targets</a>	Unlinks all target blocks from this <code>source_block</code> object. (Overrides <code>ISource::unlink_targets</code> .)

## Protected Methods

NAME	DESCRIPTION
<a href="#">accept_message</a>	When overridden in a derived class, accepts an offered message by the source. Message blocks should override this method to validate the <code>_MsgId</code> and return a message.
<a href="#">async_send</a>	Asynchronously queues up messages and starts a propagation task, if this has not been done already
<a href="#">consume_message</a>	When overridden in a derived class, consumes a message that was previously reserved.
<a href="#">enable_batched_processing</a>	Enables batched processing for this block.
<a href="#">initialize_source</a>	Initializes the <code>message_propagator</code> within this <code>source_block</code> .
<a href="#">link_target_notification</a>	A callback that notifies that a new target has been linked to this <code>source_block</code> object.
<a href="#">process_input_messages</a>	Process input messages. This is only useful for propagator blocks, which derive from <code>source_block</code>
<a href="#">propagate_output_messages</a>	Propagate messages to targets.
<a href="#">propagate_to_any_targets</a>	When overridden in a derived class, propagates the given message to any or all of the linked targets. This is the main propagation routine for message blocks.
<a href="#">release_message</a>	When overridden in a derived class, releases a previous message reservation.
<a href="#">remove_targets</a>	Removes all target links for this source block. This should be called from the destructor.
<a href="#">reserve_message</a>	When overridden in a derived class, reserves a message previously offered by this <code>source_block</code> object.

NAME	DESCRIPTION
<a href="#">resume_propagation</a>	When overridden in a derived class, resumes propagation after a reservation has been released.
<a href="#">sync_send</a>	Synchronously queues up messages and starts a propagation task, if this has not been done already.
<a href="#">unlink_target_notification</a>	A callback that notifies that a target has been unlinked from this <code>source_block</code> object.
<a href="#">wait_for_outstanding_async_sends</a>	Waits for all asynchronous propagations to complete. This propagator-specific spin wait is used in destructors of message blocks to make sure that all asynchronous propagations have time to finish before destroying the block.

## Remarks

Message blocks should derive from this block to take advantage of link management and synchronization provided by this class.

## Inheritance Hierarchy

[ISource](#)

`source_block`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## accept

Accepts a message that was offered by this `source_block` object, transferring ownership to the caller.

```
virtual message<_Target_type>* accept(
    runtime_object_identity _MsgId,
    _Inout_ ITarget<_Target_type>* _PTarget);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the offered `message` object.

*\_PTarget*

A pointer to the target block that is calling the `accept` method.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

The method throws an `invalid_argument` exception if the parameter `_PTarget` is `NULL`.

The `accept` method is called by a target while a message is being offered by this `ISource` block. The message

pointer returned may be different from the one passed into the `propagate` method of the `ITarget` block, if this source decides to make a copy of the message.

## accept\_message

When overridden in a derived class, accepts an offered message by the source. Message blocks should override this method to validate the `_MsgId` and return a message.

```
virtual message<_Target_type>* accept_message(runtime_object_identity _MsgId) = 0;
```

### Parameters

`_MsgId`

The runtime object identity of the `message` object.

### Return Value

A pointer to the message that the caller now has ownership of.

### Remarks

To transfer ownership, the original message pointer should be returned. To maintain ownership, a copy of message payload needs to be made and returned.

## acquire\_ref

Acquires a reference count on this `source_block` object, to prevent deletion.

```
virtual void acquire_ref(_Inout_ ITarget<_Target_type> *);
```

### Remarks

This method is called by an `ITarget` object that is being linked to this source during the `link_target` method.

## async\_send

Asynchronously queues up messages and starts a propagation task, if this has not been done already

```
virtual void async_send(_Inout_opt_ message<_Target_type>* _Msg);
```

### Parameters

`_Msg`

A pointer to a `message` object to asynchronously send.

## consume

Consumes a message previously offered by this `source_block` object and successfully reserved by the target, transferring ownership to the caller.

```
virtual message<_Target_type>* consume(  
    runtime_object_identity _MsgId,  
    _Inout_ ITarget<_Target_type>* _PTarget);
```

### Parameters

`_MsgId`

The `runtime_object_identity` of the reserved `message` object.

*\_PTarget*

A pointer to the target block that is calling the `consume` method.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

The method throws an `invalid_argument` exception if the parameter `_PTarget` is `NULL`.

The method throws a `bad_target` exception if the parameter `_PTarget` does not represent the target that called `reserve`.

The `consume` method is similar to `accept`, but must always be preceded by a call to `reserve` that returned **true**.

## consume\_message

When overridden in a derived class, consumes a message that was previously reserved.

```
virtual message<_Target_type>* consume_message(runtime_object_identity _MsgId) = 0;
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being consumed.

### Return Value

A pointer to the message that the caller now has ownership of.

### Remarks

Similar to `accept`, but is always preceded by a call to `reserve`.

## enable\_batched\_processing

Enables batched processing for this block.

```
void enable_batched_processing();
```

## initialize\_source

Initializes the `message_propagator` within this `source_block`.

```
void initialize_source(  
    _Inout_opt_ Scheduler* _PScheduler = NULL,  
    _Inout_opt_ ScheduleGroup* _PScheduleGroup = NULL);
```

### Parameters

*\_PScheduler*

The scheduler to be used for scheduling tasks.

*\_PScheduleGroup*

The schedule group to be used for scheduling tasks.

## link\_target

Links a target block to this `source_block` object.

```
virtual void link_target(_Inout_ ITarget<Target_type>* _PTarget);
```

### Parameters

*\_PTarget*

A pointer to an `ITarget` block to link to this `source_block` object.

### Remarks

The method throws an `invalid_argument` exception if the parameter `_PTarget` is `NULL`.

## link\_target\_notification

A callback that notifies that a new target has been linked to this `source_block` object.

```
virtual void link_target_notification(_Inout_ ITarget<Target_type> *);
```

## process\_input\_messages

Process input messages. This is only useful for propagator blocks, which derive from `source_block`

```
virtual void process_input_messages(_Inout_ message<Target_type>* _PMessage);
```

### Parameters

*\_PMessage*

A pointer to the message that is to be processed.

## propagate\_output\_messages

Propagate messages to targets.

```
virtual void propagate_output_messages();
```

## propagate\_to\_any\_targets

When overridden in a derived class, propagates the given message to any or all of the linked targets. This is the main propagation routine for message blocks.

```
virtual void propagate_to_any_targets(_Inout_opt_ message<Target_type>* _PMessage);
```

### Parameters

*\_PMessage*

A pointer to the message that is to be propagated.

## release

Releases a previous successful message reservation.

```
virtual void release(
    runtime_object_identity _MsgId,
    _Inout_ ITarget<Target_type>* _PTarget);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the reserved `message` object.

*\_PTarget*

A pointer to the target block that is calling the `release` method.

### Remarks

The method throws an `invalid_argument` exception if the parameter `_PTarget` is `NULL`.

The method throws a `bad_target` exception if the parameter `_PTarget` does not represent the target that called `reserve`.

## release\_message

When overridden in a derived class, releases a previous message reservation.

```
virtual void release_message(runtime_object_identity _MsgId) = 0;
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being released.

## release\_ref

Releases a reference count on this `source_block` object.

```
virtual void release_ref(_Inout_ ITarget<Target_type>* _PTarget);
```

### Parameters

*\_PTarget*

A pointer to the target block that is calling this method.

### Remarks

This method is called by an `ITarget` object that is being unlinked from this source. The source block is allowed to release any resources reserved for the target block.

## remove\_targets

Removes all target links for this source block. This should be called from the destructor.

```
void remove_targets();
```

## reserve

Reserves a message previously offered by this `source_block` object.

```
virtual bool reserve(  
    runtime_object_identity _MsgId,  
    _Inout_ ITarget<_Target_type>* _PTarget);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the offered `message` object.

*\_PTarget*

A pointer to the target block that is calling the `reserve` method.

### Return Value

**true** if the message was successfully reserved, **false** otherwise. Reservations can fail for many reasons, including: the message was already reserved or accepted by another target, the source could deny reservations, and so forth.

### Remarks

The method throws an `invalid_argument` exception if the parameter `_PTarget` is `NULL`.

After you call `reserve`, if it succeeds, you must call either `consume` or `release` in order to take or give up possession of the message, respectively.

## reserve\_message

When overridden in a derived class, reserves a message previously offered by this `source_block` object.

```
virtual bool reserve_message(runtime_object_identity _MsgId) = 0;
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being reserved.

### Return Value

**true** if the message was successfully reserved, **false** otherwise.

### Remarks

After `reserve` is called, if it returns **true**, either `consume` or `release` must be called to either take or release ownership of the message.

## resume\_propagation

When overridden in a derived class, resumes propagation after a reservation has been released.

```
virtual void resume_propagation() = 0;
```

## source\_block

Constructs a `source_block` object.

```
source_block();
```

## ~source\_block

Destroys the `source_block` object.

```
virtual ~source_block();
```

## sync\_send

Synchronously queues up messages and starts a propagation task, if this has not been done already.

```
virtual void sync_send(_Inout_opt_ message<_Target_type>* _Msg);
```

### Parameters

*\_Msg*

A pointer to a `message` object to synchronously send.

## unlink\_target

Unlinks a target block from this `source_block` object.

```
virtual void unlink_target(_Inout_ ITarget<_Target_type>* _PTarget);
```

### Parameters

*\_PTarget*

A pointer to an `ITarget` block to unlink from this `source_block` object.

### Remarks

The method throws an `invalid_argument` exception if the parameter `_PTarget` is `NULL`.

## unlink\_target\_notification

A callback that notifies that a target has been unlinked from this `source_block` object.

```
virtual void unlink_target_notification(_Inout_ ITarget<_Target_type>* _PTarget);
```

### Parameters

*\_PTarget*

The `ITarget` block that was unlinked.

## unlink\_targets

Unlinks all target blocks from this `source_block` object.

```
virtual void unlink_targets();
```

## wait\_for\_outstanding\_async\_sends

Waits for all asynchronous propagations to complete. This propagator-specific spin wait is used in destructors of message blocks to make sure that all asynchronous propagations have time to finish before destroying the block.

```
void wait_for_outstanding_async_sends();
```

## See also

[concurrency Namespace](#)

[ISource Class](#)

# source\_link\_manager Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `source_link_manager` object manages messaging block network links to `ISource` blocks.

## Syntax

```
template<class _LinkRegistry>
class source_link_manager;
```

### Parameters

`_LinkRegistry`

The network link registry.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>const_pointer</code>	A type that provides a pointer to a <code>const</code> element in a <code>source_link_manager</code> object.
<code>const_reference</code>	A type that provides a reference to a <code>const</code> element stored in a <code>source_link_manager</code> object for reading and performing const operations.
<code>iterator</code>	A type that provides an iterator that can read or modify any element in the <code>source_link_manager</code> object.
<code>type</code>	The type of link registry being managed by the <code>source_link_manager</code> object.

### Public Constructors

NAME	DESCRIPTION
<code>source_link_manager</code>	Constructs a <code>source_link_manager</code> object.
<code>~source_link_manager</code> Destructor	Destroys the <code>source_link_manager</code> object.

### Public Methods

NAME	DESCRIPTION
<code>add</code>	Adds a source link to the <code>source_link_manager</code> object.
<code>begin</code>	Returns an iterator to the first element in the <code>source_link_manager</code> object.

NAME	DESCRIPTION
<a href="#">contains</a>	Searches the <code>network_link_registry</code> within this <code>source_link_manager</code> object for a specified block.
<a href="#">count</a>	Counts the number of linked blocks in the <code>source_link_manager</code> object.
<a href="#">reference</a>	Acquires a reference on the <code>source_link_manager</code> object.
<a href="#">register_target_block</a>	Registers the target block that holds this <code>source_link_manager</code> object.
<a href="#">release</a>	Releases the reference on the <code>source_link_manager</code> object.
<a href="#">remove</a>	Removes a link from the <code>source_link_manager</code> object.
<a href="#">set_bound</a>	Sets the maximum number of source links that can be added to this <code>source_link_manager</code> object.

## Remarks

Currently, the source blocks are reference counted. This is a wrapper on a `network_link_registry` object that allows concurrent access to the links and provides the ability to reference the links through callbacks. Message blocks (`target_block`s or `propagator_block`s) should use this class for their source links.

## Inheritance Hierarchy

`source_link_manager`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## add

Adds a source link to the `source_link_manager` object.

```
void add(_EType _Link);
```

### Parameters

*\_Link*

A pointer to a block to be added.

## begin

Returns an iterator to the first element in the `source_link_manager` object.

```
iterator begin();
```

### Return Value

An iterator addressing the first element in the `source_link_manager` object.

### Remarks

The end state of the iterator is indicated by a `NULL` link.

## contains

Searches the `network_link_registry` within this `source_link_manager` object for a specified block.

```
bool contains(_EType _Link);
```

### Parameters

*\_Link*

A pointer to a block that is to be searched for in the `source_link_manager` object.

### Return Value

**true** if the specified block was found, **false** otherwise.

## count

Counts the number of linked blocks in the `source_link_manager` object.

```
size_t count();
```

### Return Value

The number of linked blocks in the `source_link_manager` object.

## reference

Acquires a reference on the `source_link_manager` object.

```
void reference();
```

## register\_target\_block

Registers the target block that holds this `source_link_manager` object.

```
void register_target_block(_Inout_ ITarget<typename _Block::source_type>* _PTarget);
```

### Parameters

*\_PTarget*

The target block holding this `source_link_manager` object.

## release

Releases the reference on the `source_link_manager` object.

```
void release();
```

## remove

Removes a link from the `source_link_manager` object.

```
bool remove(_EType _Link);
```

### Parameters

*\_Link*

A pointer to a block to be removed, if found.

### Return Value

**true** if the link was found and removed, **false** otherwise.

## set\_bound

Sets the maximum number of source links that can be added to this `source_link_manager` object.

```
void set_bound(size_t _MaxLinks);
```

### Parameters

*\_MaxLinks*

The maximum number of links.

## source\_link\_manager

Constructs a `source_link_manager` object.

```
source_link_manager();
```

## ~source\_link\_manager

Destroys the `source_link_manager` object.

```
~source_link_manager();
```

## See also

[concurrency Namespace](#)

[single\\_link\\_registry Class](#)

[multi\\_link\\_registry Class](#)

# static\_partitioner Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `static_partitioner` class represents a static partitioning of the range iterated over by `parallel_for`. The partitioner divides the range into as many chunks as there are workers available to the underlying scheduler.

## Syntax

```
class static_partitioner;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">static_partitioner</a>	Constructs a <code>static_partitioner</code> object.
<a href="#">~static_partitioner Destructor</a>	Destroys a <code>static_partitioner</code> object.

## Inheritance Hierarchy

```
static_partitioner
```

## Requirements

**Header:** ppl.h

**Namespace:** concurrency

### ~static\_partitioner

Destroys a `static_partitioner` object.

```
~static_partitioner();
```

### static\_partitioner

Constructs a `static_partitioner` object.

```
static_partitioner();
```

## See also

[concurrency Namespace](#)

# structured\_task\_group Class

3/4/2019 • 8 minutes to read • [Edit Online](#)

The `structured_task_group` class represents a highly structured collection of parallel work. You can queue individual parallel tasks to a `structured_task_group` using `task_handle` objects, and wait for them to complete, or cancel the task group before they have finished executing, which will abort any tasks that have not begun execution.

## Syntax

```
class structured_task_group;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">structured_task_group</a>	Overloaded. Constructs a new <code>structured_task_group</code> object.
<a href="#">~structured_task_group</a> Destructor	Destroys a <code>structured_task_group</code> object. You are expected to call either the <code>wait</code> or <code>run_and_wait</code> method on the object prior to the destructor executing, unless the destructor is executing as a result of stack unwinding due to an exception.

### Public Methods

NAME	DESCRIPTION
<a href="#">cancel</a>	Makes a best effort attempt to cancel the sub-tree of work rooted at this task group. Every task scheduled on the task group will get canceled transitively if possible.
<a href="#">is_canceling</a>	Informs the caller whether or not the task group is currently in the midst of a cancellation. This does not necessarily indicate that the <code>cancel</code> method was called on the <code>structured_task_group</code> object (although such certainly qualifies this method to return <b>true</b> ). It may be the case that the <code>structured_task_group</code> object is executing inline and a task group further up in the work tree was canceled. In cases such as these where the runtime can determine ahead of time that cancellation will flow through this <code>structured_task_group</code> object, <b>true</b> will be returned as well.

NAME	DESCRIPTION
<code>run</code>	Overloaded. Schedules a task on the <code>structured_task_group</code> object. The caller manages the lifetime of the <code>task_handle</code> object passed in the <code>_Task_handle</code> parameter. The version that takes the parameter <code>_Placement</code> causes the task to be biased towards executing at the location specified by that parameter.
<code>run_and_wait</code>	Overloaded. Schedules a task to be run inline on the calling context with the assistance of the <code>structured_task_group</code> object for full cancellation support. If a <code>task_handle</code> object is passed as a parameter to <code>run_and_wait</code> , the caller is responsible for managing the lifetime of the <code>task_handle</code> object. The function then waits until all work on the <code>structured_task_group</code> object has either completed or been canceled.
<code>wait</code>	Waits until all work on the <code>structured_task_group</code> has completed or is canceled.

## Remarks

There are a number of severe restrictions placed on usage of a `structured_task_group` object in order to gain performance:

- A single `structured_task_group` object cannot be used by multiple threads. All operations on a `structured_task_group` object must be performed by the thread that created the object. The two exceptions to this rule are the member functions `cancel` and `is_canceling`. The object may not be in the capture list of a lambda expression and be used within a task, unless the task is using one of the cancellation operations.
- All tasks scheduled to a `structured_task_group` object are scheduled through the use of `task_handle` objects which you must explicitly manage the lifetime of.
- Multiple groups may only be used in absolutely nested order. If two `structured_task_group` objects are declared, the second one being declared (the inner one) must destruct before any method except `cancel` or `is_canceling` is called on the first one (the outer one). This condition holds true in both the case of simply declaring multiple `structured_task_group` objects within the same or functionally nested scopes as well as the case of a task that was queued to the `structured_task_group` via the `run` or `run_and_wait` methods.
- Unlike the general `task_group` class, all states in the `structured_task_group` class are final. After you have queued tasks to the group and waited for them to complete, you may not use the same group again.

For more information, see [Task Parallelism](#).

## Inheritance Hierarchy

`structured_task_group`

## Requirements

**Header:** `ppl.h`

**Namespace:** concurrency

## cancel

Makes a best effort attempt to cancel the sub-tree of work rooted at this task group. Every task scheduled on the task group will get canceled transitively if possible.

```
void cancel();
```

### Remarks

For more information, see [Cancellation](#).

## is\_canceling

Informs the caller whether or not the task group is currently in the midst of a cancellation. This does not necessarily indicate that the `cancel` method was called on the `structured_task_group` object (although such certainly qualifies this method to return **true**). It may be the case that the `structured_task_group` object is executing inline and a task group further up in the work tree was canceled. In cases such as these where the runtime can determine ahead of time that cancellation will flow through this `structured_task_group` object, **true** will be returned as well.

```
bool is_canceling();
```

### Return Value

An indication of whether the `structured_task_group` object is in the midst of a cancellation (or is guaranteed to be shortly).

### Remarks

For more information, see [Cancellation](#).

## run

Schedules a task on the `structured_task_group` object. The caller manages the lifetime of the `task_handle` object passed in the `_Task_handle` parameter. The version that takes the parameter `_Placement` causes the task to be biased towards executing at the location specified by that parameter.

```
template<class _Function>
void run(
    task_handle<_Function>& _Task_handle);

template<class _Function>
void run(
    task_handle<_Function>& _Task_handle,
    location& _Placement);
```

### Parameters

*\_Function*

The type of the function object that will be invoked to execute the body of the task handle.

*\_Task\_handle*

A handle to the work being scheduled. Note that the caller has responsibility for the lifetime of this object. The runtime will continue to expect it to live until either the `wait` or `run_and_wait` method has been called on this

`structured_task_group` object.

### *\_Placement*

A reference to the location where the task represented by the `_Task_handle` parameter should execute.

### **Remarks**

The runtime creates a copy of the work function that you pass to this method. Any state changes that occur in a function object that you pass to this method will not appear in your copy of that function object.

If the `structured_task_group` destructs as the result of stack unwinding from an exception, you do not need to guarantee that a call has been made to either the `wait` or `run_and_wait` method. In this case, the destructor will appropriately cancel and wait for the task represented by the `_Task_handle` parameter to complete.

Throws an [invalid\\_multiple\\_scheduling](#) exception if the task handle given by the `_Task_handle` parameter has already been scheduled onto a task group object via the `run` method and there has been no intervening call to either the `wait` or `run_and_wait` method on that task group.

## run\_and\_wait

Schedules a task to be run inline on the calling context with the assistance of the `structured_task_group` object for full cancellation support. If a `task_handle` object is passed as a parameter to `run_and_wait`, the caller is responsible for managing the lifetime of the `task_handle` object. The function then waits until all work on the `structured_task_group` object has either completed or been canceled.

```
template<class _Function>
task_group_status run_and_wait(task_handle<_Function>& _Task_handle);

template<class _Function>
task_group_status run_and_wait(const _Function& _Func);
```

### **Parameters**

#### *\_Function*

The type of the function object that will be invoked to execute the task.

#### *\_Task\_handle*

A handle to the task which will be run inline on the calling context. Note that the caller has responsibility for the lifetime of this object. The runtime will continue to expect it to live until the `run_and_wait` method finishes execution.

#### *\_Func*

A function which will be called to invoke the body of the work. This may be a lambda or other object which supports a version of the function call operator with the signature `void operator()()`.

### **Return Value**

An indication of whether the wait was satisfied or the task group was canceled, due to either an explicit cancel operation or an exception being thrown from one of its tasks. For more information, see [task\\_group\\_status](#)

### **Remarks**

Note that one or more of the tasks scheduled to this `structured_task_group` object may execute inline on the calling context.

If one or more of the tasks scheduled to this `structured_task_group` object throws an exception, the runtime will select one such exception of its choosing and propagate it out of the call to the `run_and_wait` method.

After this function returns, the `structured_task_group` object is considered in a final state and should not be

used. Note that utilization after the `run_and_wait` method returns will result in undefined behavior.

In the non-exceptional path of execution, you have a mandate to call either this method or the `wait` method before the destructor of the `structured_task_group` executes.

## structured\_task\_group

Constructs a new `structured_task_group` object.

```
structured_task_group();  
structured_task_group(cancellation_token _CancellationToken);
```

### Parameters

*\_CancellationToken*

A cancellation token to associate with this structured task group. The structured task group will be canceled when the token is canceled.

### Remarks

The constructor that takes a cancellation token creates a `structured_task_group` that will be canceled when the source associated with the token is canceled. Providing an explicit cancellation token also isolates this structured task group from participating in an implicit cancellation from a parent group with a different token or no token.

## ~structured\_task\_group

Destroys a `structured_task_group` object. You are expected to call either the `wait` or `run_and_wait` method on the object prior to the destructor executing, unless the destructor is executing as a result of stack unwinding due to an exception.

```
~structured_task_group();
```

### Remarks

If the destructor runs as the result of normal execution (for example, not stack unwinding due to an exception) and neither the `wait` nor `run_and_wait` methods have been called, the destructor may throw a [missing\\_wait](#) exception.

## wait

Waits until all work on the `structured_task_group` has completed or is canceled.

```
task_group_status wait();
```

### Return Value

An indication of whether the wait was satisfied or the task group was canceled, due to either an explicit cancel operation or an exception being thrown from one of its tasks. For more information, see [task\\_group\\_status](#)

### Remarks

Note that one or more of the tasks scheduled to this `structured_task_group` object may execute inline on the calling context.

If one or more of the tasks scheduled to this `structured_task_group` object throws an exception, the runtime

will select one such exception of its choosing and propagate it out of the call to the `wait` method.

After this function returns, the `structured_task_group` object is considered in a final state and should not be used. Note that utilization after the `wait` method returns will result in undefined behavior.

In the non-exceptional path of execution, you have a mandate to call either this method or the `run_and_wait` method before the destructor of the `structured_task_group` executes.

## See also

[concurrency Namespace](#)

[task\\_group Class](#)

[task\\_handle Class](#)

# target\_block Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

The `target_block` class is an abstract base class that provides basic link management functionality and error checking for target only blocks.

## Syntax

```
template<class _SourceLinkRegistry, class _MessageProcessorType = ordered_message_processor<typename
_SourceLinkRegistry::type::source_type>>
class target_block : public ITarget<typename _SourceLinkRegistry::type::source_type>;
```

### Parameters

*\_SourceLinkRegistry*

The link registry to be used for holding the source links.

*\_MessageProcessorType*

The processor type for message processing.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>source_iterator</code>	The type of the iterator for the <code>source_link_manager</code> for this <code>target_block</code> object.

### Public Constructors

NAME	DESCRIPTION
<code>target_block</code>	Constructs a <code>target_block</code> object.
<code>~target_block</code> Destructor	Destroys the <code>target_block</code> object.

### Public Methods

NAME	DESCRIPTION
<code>propagate</code>	Asynchronously passes a message from a source block to this target block.
<code>send</code>	Synchronously passes a message from a source block to this target block.

### Protected Methods

NAME	DESCRIPTION
<a href="#">async_send</a>	Asynchronously sends a message for processing.
<a href="#">decline_incoming_messages</a>	Indicates to the block that new messages should be declined.
<a href="#">enable_batched_processing</a>	Enables batched processing for this block.
<a href="#">initialize_target</a>	Initializes the base object. Specifically, the <code>message_processor</code> object needs to be initialized.
<a href="#">link_source</a>	Links a specified source block to this <code>target_block</code> object.
<a href="#">process_input_messages</a>	Processes messages that are received as inputs.
<a href="#">process_message</a>	When overridden in a derived class, processes a message that was accepted by this <code>target_block</code> object.
<a href="#">propagate_message</a>	When overridden in a derived class, this method asynchronously passes a message from an <code>ISource</code> block to this <code>target_block</code> object. It is invoked by the <code>propagate</code> method, when called by a source block.
<a href="#">register_filter</a>	Registers a filter method that will be invoked on every message received.
<a href="#">remove_sources</a>	Unlinks all sources after waiting for outstanding asynchronous send operations to complete.
<a href="#">send_message</a>	When overridden in a derived class, this method synchronously passes a message from an <code>ISource</code> block to this <code>target_block</code> object. It is invoked by the <code>send</code> method, when called by a source block.
<a href="#">sync_send</a>	Synchronously send a message for processing.
<a href="#">unlink_source</a>	Unlinks a specified source block from this <code>target_block</code> object.
<a href="#">unlink_sources</a>	Unlinks all source blocks from this <code>target_block</code> object. (Overrides <code>ITarget::unlink_sources</code> .)
<a href="#">wait_for_async_sends</a>	Waits for all asynchronous propagations to complete.

## Inheritance Hierarchy

[ITarget](#)

`target_block`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## async\_send

Asynchronously sends a message for processing.

```
void async_send(_Inout_opt_ message<_Source_type>* _PMessage);
```

### Parameters

*\_PMessage*

A pointer to the message being sent.

## decline\_incoming\_messages

Indicates to the block that new messages should be declined.

```
void decline_incoming_messages();
```

### Remarks

This method is called by the destructor to ensure that new messages are declined while destruction is in progress.

## enable\_batched\_processing

Enables batched processing for this block.

```
void enable_batched_processing();
```

## initialize\_target

Initializes the base object. Specifically, the `message_processor` object needs to be initialized.

```
void initialize_target(  
    _Inout_opt_ Scheduler* _PScheduler = NULL,  
    _Inout_opt_ ScheduleGroup* _PScheduleGroup = NULL);
```

### Parameters

*\_PScheduler*

The scheduler to be used for scheduling tasks.

*\_PScheduleGroup*

The schedule group to be used for scheduling tasks.

## link\_source

Links a specified source block to this `target_block` object.

```
virtual void link_source(_Inout_ ISource<_Source_type>* _PSource);
```

### Parameters

*\_PSource*

A pointer to the `ISource` block that is to be linked.

## Remarks

This function should not be called directly on a `target_block` object. Blocks should be connected together using the `link_target` method on `ISource` blocks, which will invoke the `link_source` method on the corresponding target.

## process\_input\_messages

Processes messages that are received as inputs.

```
virtual void process_input_messages(_Inout_ message<_Source_type>* _PMessage);
```

## Parameters

*\_PMessage*

A pointer to the message that is to be processed.

## process\_message

When overridden in a derived class, processes a message that was accepted by this `target_block` object.

```
virtual void process_message(message<_Source_type> *);
```

## propagate

Asynchronously passes a message from a source block to this target block.

```
virtual message_status propagate(  
    _Inout_opt_ message<_Source_type>* _PMessage,  
    _Inout_opt_ ISource<_Source_type>* _PSource);
```

## Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

## Return Value

A [message\\_status](#) indication of what the target decided to do with the message.

## Remarks

The method throws an [invalid\\_argument](#) exception if either the `_PMessage` or `_PSource` parameter is `NULL`.

## propagate\_message

When overridden in a derived class, this method asynchronously passes a message from an `ISource` block to this `target_block` object. It is invoked by the `propagate` method, when called by a source block.

```
virtual message_status propagate_message(  
    _Inout_ message<_Source_type>* _PMessage,  
    _Inout_ ISource<_Source_type>* _PSource) = 0;
```

## Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

## Return Value

A `message_status` indication of what the target decided to do with the message.

# register\_filter

Registers a filter method that will be invoked on every message received.

```
void register_filter(filter_method const& _Filter);
```

## Parameters

*\_Filter*

The filter method.

# remove\_sources

Unlinks all sources after waiting for outstanding asynchronous send operations to complete.

```
void remove_sources();
```

## Remarks

All target blocks should call this routine to remove the sources in their destructor.

# send

Synchronously passes a message from a source block to this target block.

```
virtual message_status send(  
    _Inout_ message<_Source_type>* _PMessage,  
    _Inout_ ISource<_Source_type>* _PSource);
```

## Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

## Return Value

A `message_status` indication of what the target decided to do with the message.

## Remarks

The method throws an `invalid_argument` exception if either the `_PMessage` or `_PSource` parameter is `NULL`.

Using the `send` method outside of message initiation and to propagate messages within a network is dangerous and can lead to deadlock.

When `send` returns, the message has either already been accepted, and transferred into the target block, or it has been declined by the target.

## send\_message

When overridden in a derived class, this method synchronously passes a message from an `ISource` block to this `target_block` object. It is invoked by the `send` method, when called by a source block.

```
virtual message_status send_message(  
    _Inout_ message<_Source_type> *,  
    _Inout_ ISource<_Source_type> *);
```

### Return Value

A [message\\_status](#) indication of what the target decided to do with the message.

### Remarks

By default, this block returns `declined` unless overridden by a derived class.

## sync\_send

Synchronously send a message for processing.

```
void sync_send(_Inout_opt_ message<_Source_type>* _PMessage);
```

### Parameters

*\_PMessage*

A pointer to the message being sent.

## target\_block

Constructs a `target_block` object.

```
target_block();
```

## ~target\_block

Destroys the `target_block` object.

```
virtual ~target_block();
```

## unlink\_source

Unlinks a specified source block from this `target_block` object.

```
virtual void unlink_source(_Inout_ ISource<_Source_type>* _PSource);
```

### Parameters

*\_PSource*

A pointer to the `ISource` block that is to be unlinked.

## unlink\_sources

Unlinks all source blocks from this `target_block` object.

```
virtual void unlink_sources();
```

## wait\_for\_async\_sends

Waits for all asynchronous propagations to complete.

```
void wait_for_async_sends();
```

### Remarks

This method is used by message block destructors to ensure all asynchronous operations have had time to finish before destroying the block.

## See also

[concurrency Namespace](#)

[ITarget Class](#)

# task Class (Concurrency Runtime)

3/20/2019 • 7 minutes to read • [Edit Online](#)

The Parallel Patterns Library (PPL) `task` class. A `task` object represents work that can be executed asynchronously, and concurrently with other tasks and parallel work produced by parallel algorithms in the Concurrency Runtime. It produces a result of type `_ResultType` on successful completion. Tasks of type `task<void>` produce no result. A task can be waited upon and canceled independently of other tasks. It can also be composed with other tasks using continuations( `then` ), and `join`( `when_all` ) and `choice`( `when_any` ) patterns.

## Syntax

```
template <typename T>
class task;

template <>
class task<void>;

template<typename _ReturnType>
class task;
```

### Parameters

*T*

The task object type.

*\_ReturnType*

The result type of this task.

## Members

### Public Typedefs

NAME	DESCRIPTION
<code>result_type</code>	The type of the result an object of this class produces.

### Public Constructors

NAME	DESCRIPTION
<code>task</code>	Overloaded. Constructs a <code>task</code> object.

### Public Methods

NAME	DESCRIPTION
<code>get</code>	Overloaded. Returns the result this task produced. If the task is not in a terminal state, a call to <code>get</code> will wait for the task to finish. This method does not return a value when called on a task with a <code>result_type</code> of <code>void</code> .

NAME	DESCRIPTION
<a href="#">is_apartment_aware</a>	Determines whether the task unwraps a Windows Runtime <code>IAsyncInfo</code> interface or is descended from such a task.
<a href="#">is_done</a>	Determines if the task is completed.
<a href="#">scheduler</a>	Returns the scheduler for this task
<a href="#">then</a>	Overloaded. Adds a continuation task to this task.
<a href="#">wait</a>	Waits for this task to reach a terminal state. It is possible for <code>wait</code> to execute the task inline, if all of the tasks dependencies are satisfied, and it has not already been picked up for execution by a background worker.

### Public Operators

NAME	DESCRIPTION
<a href="#">operator!=</a>	Overloaded. Determines whether two <code>task</code> objects represent different internal tasks.
<a href="#">operator=</a>	Overloaded. Replaces the contents of one <code>task</code> object with another.
<a href="#">operator==</a>	Overloaded. Determines whether two <code>task</code> objects represent the same internal task.

## Remarks

For more information, see [Task Parallelism](#).

## Inheritance Hierarchy

`task`

## Requirements

**Header:** `ppltasks.h`

**Namespace:** `concurrency`

## get

Returns the result this task produced. If the task is not in a terminal state, a call to `get` will wait for the task to finish. This method does not return a value when called on a task with a `result_type` of `void`.

```
_ReturnType get() const;
void get() const;
```

### Return Value

The result of the task.

## Remarks

If the task is canceled, a call to `get` will throw a `task_canceled` exception. If the task encountered a different exception or an exception was propagated to it from an antecedent task, a call to `get` will throw that exception.

### IMPORTANT

In a Universal Windows Platform (UWP) app, do not call `concurrency::task::wait` or `get` ( `wait` calls `get` ) in code that runs on the user-interface thread. Otherwise, the runtime throws `concurrency::invalid_operation` because these methods block the current thread and can cause the app to become unresponsive. However, you can call the `get` method to receive the result of the antecedent task in a task-based continuation because the result is immediately available.

## is\_apartment\_aware

Determines whether the task unwraps a Windows Runtime `IAsyncInfo` interface or is descended from such a task.

```
bool is_apartment_aware() const;
```

### Return Value

**true** if the task unwraps an `IAsyncInfo` interface or is descended from such a task, **false** otherwise.

## task::is\_done Method (Concurrency Runtime)

Determines if the task is completed.

```
bool is_done() const;
```

### Return Value

True if the task has completed, false otherwise.

### Remarks

The function returns true if the task is completed or canceled (with or without user exception).

## operator!=

Determines whether two `task` objects represent different internal tasks.

```
bool operator!= (const task<_ReturnType>& _Rhs) const;  
  
bool operator!= (const task<void>& _Rhs) const;
```

### Parameters

*\_Rhs*

The task to compare.

### Return Value

**true** if the objects refer to different underlying tasks, and **false** otherwise.

## operator=

Replaces the contents of one `task` object with another.

```
task& operator= (const task& _Other);  
  
task& operator= (task&& _Other);
```

### Parameters

*\_Other*

The source `task` object.

### Return Value

### Remarks

As `task` behaves like a smart pointer, after a copy assignment, this `task` object represents the same actual task as `_Other` does.

## operator==

Determines whether two `task` objects represent the same internal task.

```
bool operator== (const task<_ReturnType>& _Rhs) const;  
  
bool operator== (const task<void>& _Rhs) const;
```

### Parameters

*\_Rhs*

The task to compare.

### Return Value

**true** if the objects refer to the same underlying task, and **false** otherwise.

## task::scheduler Method (Concurrency Runtime)

Returns the scheduler for this task

```
scheduler_ptr scheduler() const;
```

### Return Value

A pointer to the scheduler

## task

Constructs a `task` object.

```

task();

template<typename T>
__declspec(
    noinline) explicit task(T _Param);

template<typename T>
__declspec(
    noinline) explicit task(T _Param, const task_options& _TaskOptions);

task(
    const task& _Other);

task(
    task&& _Other);

```

## Parameters

*T*

The type of the parameter from which the task is to be constructed.

*\_Param*

The parameter from which the task is to be constructed. This could be a lambda, a function object, a `task_completion_event<result_type>` object, or a `Windows::Foundation::IAsyncInfo` if you are using tasks in your Windows Runtime app. The lambda or function object should be a type equivalent to `std::function<X(void)>`, where X can be a variable of type `result_type`, `task<result_type>`, or a `Windows::Foundation::IAsyncInfo` in Windows Runtime apps.

*\_TaskOptions*

The task options include cancellation token, scheduler etc

*\_Other*

The source `task` object.

## Remarks

The default constructor for a `task` is only present in order to allow tasks to be used within containers. A default constructed task cannot be used until you assign a valid task to it. Methods such as `get`, `wait` or `then` will throw an `invalid_argument` exception when called on a default constructed task.

A task that is created from a `task_completion_event` will complete (and have its continuations scheduled) when the task completion event is set.

The version of the constructor that takes a cancellation token creates a task that can be canceled using the `cancellation_token_source` the token was obtained from. Tasks created without a cancellation token are not cancelable.

Tasks created from a `Windows::Foundation::IAsyncInfo` interface or a lambda that returns an `IAsyncInfo` interface reach their terminal state when the enclosed Windows Runtime asynchronous operation or action completes. Similarly, tasks created from a lambda that returns a `task<result_type>` reach their terminal state when the inner task reaches its terminal state, and not when the lambda returns.

`task` behaves like a smart pointer and is safe to pass around by value. It can be accessed by multiple threads without the need for locks.

The constructor overloads that take a `Windows::Foundation::IAsyncInfo` interface or a lambda returning such an interface, are only available to Windows Runtime apps.

For more information, see [Task Parallelism](#).

# then

Adds a continuation task to this task.

```
template<typename _Function>
__declspec(
    noinline) auto then(const _Function& _Func) const -> typename
details::_ContinuationTypeTraits<_Function,
    _ReturnType>::_TaskOfType;

template<typename _Function>
__declspec(
    noinline) auto then(const _Function& _Func,
    const task_options& _TaskOptions) const -> typename details::_ContinuationTypeTraits<_Function,
    _ReturnType>::_TaskOfType;

template<typename _Function>
__declspec(
    noinline) auto then(const _Function& _Func,
    cancellation_token _CancellationToken,
    task_continuation_context _ContinuationContext) const -> typename
details::_ContinuationTypeTraits<_Function,
    _ReturnType>::_TaskOfType;

template<typename _Function>
__declspec(
    noinline) auto then(const _Function& _Func,
    const task_options& _TaskOptions = task_options()) const -> typename
details::_ContinuationTypeTraits<_Function,
    void>::_TaskOfType;

template<typename _Function>
__declspec(
    noinline) auto then(const _Function& _Func,
    cancellation_token _CancellationToken,
    task_continuation_context _ContinuationContext) const -> typename
details::_ContinuationTypeTraits<_Function,
    void>::_TaskOfType;
```

## Parameters

### *\_Function*

The type of the function object that will be invoked by this task.

### *\_Func*

The continuation function to execute when this task completes. This continuation function must take as input a variable of either `result_type` or `task<result_type>`, where `result_type` is the type of the result this task produces.

### *\_TaskOptions*

The task options include cancellation token, scheduler and continuation context. By default the former 3 options are inherited from the antecedent task

### *\_CancellationToken*

The cancellation token to associate with the continuation task. A continuation task that is created without a cancellation token will inherit the token of its antecedent task.

### *\_ContinuationContext*

A variable that specifies where the continuation should execute. This variable is only useful when used in a UWP app. For more information, see [task\\_continuation\\_context](#)

## Return Value

The newly created continuation task. The result type of the returned task is determined by what `_Func` returns.

### Remarks

The overloads of `then` that take a lambda or functor that returns a `Windows::Foundation::IAsyncInfo` interface, are only available to Windows Runtime apps.

For more information on how to use task continuations to compose asynchronous work, see [Task Parallelism](#).

## wait

Waits for this task to reach a terminal state. It is possible for `wait` to execute the task inline, if all of the tasks dependencies are satisfied, and it has not already been picked up for execution by a background worker.

```
task_status wait() const;
```

### Return Value

A `task_status` value which could be either `completed` or `canceled`. If the task encountered an exception during execution, or an exception was propagated to it from an antecedent task, `wait` will throw that exception.

### Remarks

#### IMPORTANT

In a Universal Windows Platform (UWP) app, do not call `wait` in code that runs on the user-interface thread. Otherwise, the runtime throws `concurrency::invalid_operation` because this method blocks the current thread and can cause the app to become unresponsive. However, you can call the `concurrency::task::get` method to receive the result of the antecedent task in a task-based continuation.

## See also

[concurrency Namespace](#)

# task\_canceled Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown by the PPL tasks layer in order to force the current task to cancel. It is also thrown by the `get()` method on `task`, for a canceled task.

## Syntax

```
class task_canceled : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<code>task_canceled</code>	Overloaded. Constructs a <code>task_canceled</code> object.

## Inheritance Hierarchy

`exception`

`task_canceled`

## Requirements

**Header:** `concrth`

**Namespace:** `concurrency`

## task\_canceled

Constructs a `task_canceled` object.

```
explicit _CRTIMP task_canceled(_In_z_ const char* _Message) throw();  
task_canceled() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

# task\_completion\_event Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

The `task_completion_event` class allows you to delay the execution of a task until a condition is satisfied, or start a task in response to an external event.

## Syntax

```
template<typename _ResultType>
class task_completion_event;

template<>
class task_completion_event<void>;
```

### Parameters

*\_ResultType*

The result type of this `task_completion_event` class.

## Members

### Public Constructors

NAME	DESCRIPTION
<code>task_completion_event</code>	Constructs a <code>task_completion_event</code> object.

### Public Methods

NAME	DESCRIPTION
<code>set</code>	Overloaded. Sets the task completion event.
<code>set_exception</code>	Overloaded. Propagates an exception to all tasks associated with this event.

## Remarks

Use a task created from a task completion event when your scenario requires you to create a task that will complete, and thereby have its continuations scheduled for execution, at some point in the future. The `task_completion_event` must have the same type as the task you create, and calling the set method on the task completion event with a value of that type will cause the associated task to complete, and provide that value as a result to its continuations.

If the task completion event is never signaled, any tasks created from it will be canceled when it is destructed.

`task_completion_event` behaves like a smart pointer, and should be passed by value.

## Inheritance Hierarchy

`task_completion_event`

# Requirements

**Header:** `ppltasks.h`

**Namespace:** `concurrency`

## set

Sets the task completion event.

```
bool set(_ResultType _Result) const ;  
  
bool set() const ;
```

### Parameters

*\_Result*

The result to set this event with.

### Return Value

The method returns **true** if it was successful in setting the event. It returns **false** if the event is already set.

### Remarks

In the presence of multiple or concurrent calls to `set`, only the first call will succeed and its result (if any) will be stored in the task completion event. The remaining sets are ignored and the method will return false. When you set a task completion event, all the tasks created from that event will immediately complete, and its continuations, if any, will be scheduled. Task completion objects that have a `_ResultType` other than **void** will pass the value to their continuations.

## set\_exception

Propagates an exception to all tasks associated with this event.

```
template<typename _E>  
__declspec(noinline) bool set_exception(_E _Except) const;  
  
__declspec(noinline) bool set_exception(std::exception_ptr _ExceptionPtr) const ;
```

### Parameters

*\_E*

The exception type.

*\_Except*

The exception to set.

*\_ExceptionPtr*

The exception pointer to set.

### Return Value

## task\_completion\_event

Constructs a `task_completion_event` object.

```
task_completion_event();
```

# See also

[concurrency Namespace](#)

# task\_continuation\_context Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

The `task_continuation_context` class allows you to specify where you would like a continuation to be executed. It is only useful to use this class from a Windows Runtime app. For non-Windows Runtime apps, the task continuation's execution context is determined by the runtime, and not configurable.

## Syntax

```
class task_continuation_context : public details::_ContextCallback;
```

## Members

### Public Methods

NAME	DESCRIPTION
<a href="#">get_current_winrt_context</a>	Returns a task continuation context object that represents the current winrt thread context.
<a href="#">use_arbitrary</a>	Creates a task continuation context which allows the Runtime to choose the execution context for a continuation.
<a href="#">use_current</a>	Returns a task continuation context object that represents the current execution context.
<a href="#">use_default</a>	Creates the default task continuation context.
<a href="#">use_synchronous_execution</a>	Returns a task continuation context object that represents the synchronous execution context.

## Inheritance Hierarchy

`_ContextCallback`

`task_continuation_context`

## Requirements

**Header:** `ppltasks.h`

**Namespace:** `concurrency`

## `get_current_winrt_context`

Returns a task continuation context object that represents the current WinRT thread context.

## Syntax

```
static task_continuation_context get_current_winrt_context();
```

## Return Value

The current Windows Runtime thread context. Returns an empty `task_continuation_context` if called from a non-Windows Runtime context.

## Remarks

The `get_current_winrt_context` method captures the caller's Windows Runtime thread context. It returns an empty context to non-Windows Runtime callers.

The value returned by `get_current_winrt_context` can be used to indicate to the Runtime that the continuation should execute in the apartment model of the captured context (STA vs MTA), regardless of whether the antecedent task is apartment aware. An apartment aware task is a task that unwraps a Windows Runtime `IAsyncInfo` interface, or a task that is descended from such a task.

This method is similar to the `use_current` method, but it is also available to native C++ code without C++/CX extension support. It is intended for use by advanced users writing C++/CX-agnostic library code for both native and Windows Runtime callers. Unless you need this functionality, we recommend the `use_current` method, which is only available to C++/CX clients.

## use\_arbitrary

Creates a task continuation context which allows the Runtime to choose the execution context for a continuation.

```
static task_continuation_context use_arbitrary();
```

### Return Value

A task continuation context that represents an arbitrary location.

### Remarks

When this continuation context is used the continuation will execute in a context the runtime chooses even if the antecedent task is apartment aware.

`use_arbitrary` can be used to turn off the default behavior for a continuation on an apartment aware task created in an STA.

This method is only available to Windows Runtime apps.

## use\_current

Returns a task continuation context object that represents the current execution context.

```
static task_continuation_context use_current();
```

### Return Value

The current execution context.

### Remarks

This method captures the caller's Windows Runtime context so that continuations can be executed in the right apartment.

The value returned by `use_current` can be used to indicate to the Runtime that the continuation should execute in the captured context (STA vs MTA) regardless of whether or not the antecedent task is apartment aware. An apartment aware task is a task that unwraps a Windows Runtime `IAsyncInfo` interface, or a task that is descended from such a task.

This method is only available to Windows Runtime apps.

## use\_default

Creates the default task continuation context.

```
static task_continuation_context use_default();
```

### Return Value

The default continuation context.

### Remarks

The default context is used if you don't specify a continuation context when you call the `then` method. In Windows applications for Windows 7 and below, as well as desktop applications on Windows 8 and higher, the runtime determines where task continuations will execute. However, in a Windows Runtime app, the default continuation context for a continuation on an apartment aware task is the apartment where `then` is invoked.

An apartment aware task is a task that unwraps a Windows Runtime `IAsyncInfo` interface, or a task that is descended from such a task. Therefore, if you schedule a continuation on an apartment aware task in a Windows Runtime STA, the continuation will execute in that STA.

A continuation on a non-apartment aware task will execute in a context the Runtime chooses.

## task\_continuation\_context::use\_synchronous\_execution

Returns a task continuation context object that represents the synchronous execution context.

## Syntax

```
static task_continuation_context use_synchronous_execution();
```

## Return Value

The synchronous execution context.

## Remarks

The `use_synchronous_execution` method forces the continuation task to run synchronously on the context, causing its antecedent task's completion.

If the antecedent task has already completed when the continuation is attached, the continuation runs synchronously on the context that attaches the continuation.

## See also

[concurrency Namespace](#)

# task\_group Class

3/4/2019 • 8 minutes to read • [Edit Online](#)

The `task_group` class represents a collection of parallel work which can be waited on or canceled.

## Syntax

```
class task_group;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">task_group</a>	Overloaded. Constructs a new <code>task_group</code> object.
<a href="#">~task_group</a> Destructor	Destroys a <code>task_group</code> object. You are expected to call the either the <code>wait</code> or <code>run_and_wait</code> method on the object prior to the destructor executing, unless the destructor is executing as the result of stack unwinding due to an exception.

### Public Methods

NAME	DESCRIPTION
<a href="#">cancel</a>	Makes a best effort attempt to cancel the sub-tree of work rooted at this task group. Every task scheduled on the task group will get canceled transitively if possible.
<a href="#">is_canceling</a>	Informs the caller whether or not the task group is currently in the midst of a cancellation. This does not necessarily indicate that the <code>cancel</code> method was called on the <code>task_group</code> object (although such certainly qualifies this method to return <code>true</code> ). It may be the case that the <code>task_group</code> object is executing inline and a task group further up in the work tree was canceled. In cases such as these where the runtime can determine ahead of time that cancellation will flow through this <code>task_group</code> object, <code>true</code> will be returned as well.

NAME	DESCRIPTION
<code>run</code>	Overloaded. Schedules a task on the <code>task_group</code> object. If a <code>task_handle</code> object is passed as a parameter to <code>run</code> , the caller is responsible for managing the lifetime of the <code>task_handle</code> object. The version of the method that takes a reference to a function object as a parameter involves heap allocation inside the runtime which may be perform less well than using the version that takes a reference to a <code>task_handle</code> object. The version which takes the parameter <code>_Placement</code> causes the task to be biased towards executing at the location specified by that parameter.
<code>run_and_wait</code>	Overloaded. Schedules a task to be run inline on the calling context with the assistance of the <code>task_group</code> object for full cancellation support. The function then waits until all work on the <code>task_group</code> object has either completed or been canceled. If a <code>task_handle</code> object is passed as a parameter to <code>run_and_wait</code> , the caller is responsible for managing the lifetime of the <code>task_handle</code> object.
<code>wait</code>	Waits until all work on the <code>task_group</code> object has either completed or been canceled.

## Remarks

Unlike the heavily restricted `structured_task_group` class, the `task_group` class is much more general construct. It does not have any of the restrictions described by `structured_task_group`. `task_group` objects may safely be used across threads and utilized in free-form ways. The disadvantage of the `task_group` construct is that it may not perform as well as the `structured_task_group` construct for tasks which perform small amounts of work.

For more information, see [Task Parallelism](#).

## Inheritance Hierarchy

`task_group`

## Requirements

**Header:** `ppl.h`

**Namespace:** `concurrency`

## cancel

Makes a best effort attempt to cancel the sub-tree of work rooted at this task group. Every task scheduled on the task group will get canceled transitively if possible.

```
void cancel();
```

## Remarks

For more information, see [Cancellation](#).

## is\_canceling

Informs the caller whether or not the task group is currently in the midst of a cancellation. This does not necessarily indicate that the `cancel` method was called on the `task_group` object (although such certainly qualifies this method to return `true`). It may be the case that the `task_group` object is executing inline and a task group further up in the work tree was canceled. In cases such as these where the runtime can determine ahead of time that cancellation will flow through this `task_group` object, `true` will be returned as well.

```
bool is_canceling();
```

### Return Value

An indication of whether the `task_group` object is in the midst of a cancellation (or is guaranteed to be shortly).

### Remarks

For more information, see [Cancellation](#).

## run

Schedules a task on the `task_group` object. If a `task_handle` object is passed as a parameter to `run`, the caller is responsible for managing the lifetime of the `task_handle` object. The version of the method that takes a reference to a function object as a parameter involves heap allocation inside the runtime which may be perform less well than using the version that takes a reference to a `task_handle` object. The version which takes the parameter `_Placement` causes the task to be biased towards executing at the location specified by that parameter.

```
template<
    typename _Function
>
void run(
    const _Function& _Func
);

template<
    typename _Function
>
void run(
    const _Function& _Func,
    location& _Placement
);

template<
    typename _Function
>
void run(
    task_handle<_Function>& _Task_handle
);

template<
    typename _Function
>
void run(
    task_handle<_Function>& _Task_handle,
    location& _Placement
);
```

### Parameters

### *\_Function*

The type of the function object that will be invoked to execute the body of the task handle.

### *\_Func*

A function which will be called to invoke the body of the task. This may be a lambda expression or other object which supports a version of the function call operator with the signature `void operator()()`.

### *\_Placement*

A reference to the location where the task represented by the `_Func` parameter should execute.

### *\_Task\_handle*

A handle to the work being scheduled. Note that the caller has responsibility for the lifetime of this object. The runtime will continue to expect it to live until either the `wait` or `run_and_wait` method has been called on this `task_group` object.

### Remarks

The runtime schedules the provided work function to run at a later time, which can be after the calling function returns. This method uses a `task_handle` object to hold a copy of the provided work function. Therefore, any state changes that occur in a function object that you pass to this method will not appear in your copy of that function object. In addition, make sure that the lifetime of any objects that you pass by pointer or by reference to the work function remain valid until the work function returns.

If the `task_group` destructs as the result of stack unwinding from an exception, you do not need to guarantee that a call has been made to either the `wait` or `run_and_wait` method. In this case, the destructor will appropriately cancel and wait for the task represented by the `_Task_handle` parameter to complete.

The method throws an `invalid_multiple_scheduling` exception if the task handle given by the `_Task_handle` parameter has already been scheduled onto a task group object via the `run` method and there has been no intervening call to either the `wait` or `run_and_wait` method on that task group.

## run\_and\_wait

Schedules a task to be run inline on the calling context with the assistance of the `task_group` object for full cancellation support. The function then waits until all work on the `task_group` object has either completed or been canceled. If a `task_handle` object is passed as a parameter to `run_and_wait`, the caller is responsible for managing the lifetime of the `task_handle` object.

```
template<
    class _Function
>
task_group_status run_and_wait(
    task_handle<_Function>& _Task_handle
);

template<
    class _Function
>
task_group_status run_and_wait(
    const _Function& _Func
);
```

### Parameters

#### *\_Function*

The type of the function object that will be invoked to execute the body of the task.

#### *\_Task\_handle*

A handle to the task which will be run inline on the calling context. Note that the caller has responsibility for

the lifetime of this object. The runtime will continue to expect it to live until the `run_and_wait` method finishes execution.

#### *\_Func*

A function which will be called to invoke the body of the work. This may be a lambda expression or other object which supports a version of the function call operator with the signature `void operator()()`.

#### **Return Value**

An indication of whether the wait was satisfied or the task group was canceled, due to either an explicit cancel operation or an exception being thrown from one of its tasks. For more information, see [task\\_group\\_status](#).

#### **Remarks**

Note that one or more of the tasks scheduled to this `task_group` object may execute inline on the calling context.

If one or more of the tasks scheduled to this `task_group` object throws an exception, the runtime will select one such exception of its choosing and propagate it out of the call to the `run_and_wait` method.

Upon return from the `run_and_wait` method on a `task_group` object, the runtime resets the object to a clean state where it can be reused. This includes the case where the `task_group` object was canceled.

In the non-exceptional path of execution, you have a mandate to call either this method or the `wait` method before the destructor of the `task_group` executes.

## task\_group

Constructs a new `task_group` object.

```
task_group();

task_group(
    cancellation_token _CancellationToken
);
```

#### **Parameters**

##### *\_CancellationToken*

A cancellation token to associate with this task group. The task group will be canceled when the token is canceled.

#### **Remarks**

The constructor that takes a cancellation token creates a `task_group` that will be canceled when the source associated with the token is canceled. Providing an explicit cancellation token also isolates this task group from participating in an implicit cancellation from a parent group with a different token or no token.

## ~task\_group

Destroys a `task_group` object. You are expected to call either the `wait` or `run_and_wait` method on the object prior to the destructor executing, unless the destructor is executing as the result of stack unwinding due to an exception.

```
~task_group();
```

#### **Remarks**

If the destructor runs as the result of normal execution (for example, not stack unwinding due to an exception) and neither the `wait` nor `run_and_wait` methods have been called, the destructor may throw a [missing\\_wait](#) exception.

## wait

Waits until all work on the `task_group` object has either completed or been canceled.

```
task_group_status wait();
```

### Return Value

An indication of whether the wait was satisfied or the task group was canceled, due to either an explicit cancel operation or an exception being thrown from one of its tasks. For more information, see [task\\_group\\_status](#).

### Remarks

Note that one or more of the tasks scheduled to this `task_group` object may execute inline on the calling context.

If one or more of the tasks scheduled to this `task_group` object throws an exception, the runtime will select one such exception of its choosing and propagate it out of the call to the `wait` method.

Calling `wait` on a `task_group` object resets it to a clean state where it can be reused. This includes the case where the `task_group` object was canceled.

In the non-exceptional path of execution, you have a mandate to call either this method or the `run_and_wait` method before the destructor of the `task_group` executes.

## See also

[concurrency Namespace](#)

[structured\\_task\\_group Class](#)

[task\\_handle Class](#)

# task\_handle Class

5/21/2019 • 2 minutes to read • [Edit Online](#)

The `task_handle` class represents an individual parallel work item. It encapsulates the instructions and the data required to execute a piece of work.

## Syntax

```
template<
    typename _Function
>
class task_handle : public ::Concurrency::details::_UnrealizedChore;
```

### Parameters

*\_Function*

The type of the function object that will be invoked to execute the work represented by the `task_handle` object.

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">task_handle</a>	Constructs a new <code>task_handle</code> object. The work of the task is performed by invoking the function specified as a parameter to the constructor.
<a href="#">~task_handle</a> Destructor	Destroys the <code>task_handle</code> object.

### Public Operators

NAME	DESCRIPTION
<a href="#">operator()</a>	The function call operator that the runtime invokes to perform the work of the task handle.

## Remarks

`task_handle` objects can be used in conjunction with a `structured_task_group` or a more general `task_group` object, to decompose work into parallel tasks. For more information, see [Task Parallelism](#).

Note that the creator of a `task_handle` object is responsible for maintaining the lifetime of the created `task_handle` object until it is no longer required by the Concurrency Runtime. Typically, this means that the `task_handle` object must not destruct until either the `wait` or `run_and_wait` method of the `task_group` or `structured_task_group` to which it is queued has been called.

`task_handle` objects are typically used in conjunction with C++ lambdas. Because you do not know the true type of the lambda, the `make_task` function is typically used to create a `task_handle` object.

The runtime creates a copy of the work function that you pass to a `task_handle` object. Therefore, any state

changes that occur in a function object that you pass to a `task_handle` object will not appear in your copy of that function object.

## Inheritance Hierarchy

`task_handle`

## Requirements

**Header:** `ppl.h`

**Namespace:** `concurrency`

## operator()

The function call operator that the runtime invokes to perform the work of the task handle.

```
void operator>() const;
```

## task\_handle

Constructs a new `task_handle` object. The work of the task is performed by invoking the function specified as a parameter to the constructor.

```
task_handle(const _Function& _Func);
```

### Parameters

*\_Func*

The function that will be invoked to execute the work represented by the `task_handle` object. This may be a lambda function, a pointer to a function, or any object that supports a version of the function call operator with the signature `void operator>()`.

### Remarks

The runtime creates a copy of the work function that you pass to the constructor. Therefore, any state changes that occur in a function object that you pass to a `task_handle` object will not appear in your copy of that function object.

## ~task\_handle

Destroys the `task_handle` object.

```
~task_handle();
```

## See also

[concurrency Namespace](#)

[task\\_group Class](#)

[structured\\_task\\_group Class](#)

# task\_options Class (Concurrency Runtime)

3/4/2019 • 2 minutes to read • [Edit Online](#)

Represents the allowed options for creating a task

## Syntax

```
class task_options;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">task_options::task_options Constructor (Concurrency Runtime)</a>	Overloaded. Default list of task creation options

### Public Methods

NAME	DESCRIPTION
<a href="#">task_options::get_cancellation_token Method (Concurrency Runtime)</a>	Returns the cancellation token
<a href="#">task_options::get_continuation_context Method (Concurrency Runtime)</a>	Returns the continuation context
<a href="#">task_options::get_scheduler Method (Concurrency Runtime)</a>	Returns the scheduler
<a href="#">task_options::has_cancellation_token Method (Concurrency Runtime)</a>	Indicates whether a cancellation token was specified by the user
<a href="#">task_options::has_scheduler Method (Concurrency Runtime)</a>	Indicates whether a scheduler n was specified by the user
<a href="#">task_options::set_cancellation_token Method (Concurrency Runtime)</a>	Sets the given token in the options
<a href="#">task_options::set_continuation_context Method (Concurrency Runtime)</a>	Sets the given continuation context in the options

## Inheritance Hierarchy

task\_options

## Requirements

**Header:** ppltasks.h

**Namespace:** concurrency

## task\_options::get\_cancellation\_token Method (Concurrency Runtime)

Returns the cancellation token

```
cancellation_token get_cancellation_token() const;
```

### Return Value

## task\_options::get\_continuation\_context Method (Concurrency Runtime)

Returns the continuation context

```
task_continuation_context get_continuation_context() const;
```

### Return Value

## task\_options::get\_scheduler Method (Concurrency Runtime)

Returns the scheduler

```
scheduler_ptr get_scheduler() const;
```

### Return Value

## task\_options::has\_cancellation\_token Method (Concurrency Runtime)

Indicates whether a cancellation token was specified by the user

```
bool has_cancellation_token() const;
```

### Return Value

## task\_options::has\_scheduler Method (Concurrency Runtime)

Indicates whether a scheduler n was specified by the user

```
bool has_scheduler() const;
```

### Return Value

## task\_options::set\_cancellation\_token Method (Concurrency Runtime)

Sets the given token in the options

```
void set_cancellation_token(cancellation_token _Token);
```

### Parameters

`_Token`

## task\_options::set\_continuation\_context Method (Concurrency Runtime)

Sets the given continuation context in the options

```
void set_continuation_context(task_continuation_context _ContinuationContext);
```

### Parameters

`_ContinuationContext`

## task\_options::task\_options Constructor (Concurrency Runtime)

Default list of task creation options

```
task_options();

task_options(
    cancellation_token _Token);

task_options(
    task_continuation_context _ContinuationContext);

task_options(
    cancellation_token _Token,
    task_continuation_context _ContinuationContext);

template<typename _SchedType>
task_options(
    std::shared_ptr<_SchedType> _Scheduler);

task_options(
    scheduler_interface& _Scheduler);

task_options(
    scheduler_ptr _Scheduler);

task_options(
    const task_options& _TaskOptions);
```

### Parameters

`_SchedType`

`_Token`

`_ContinuationContext`

`_Scheduler`

`_TaskOptions`

## See also

[concurrency Namespace](#)

# timer Class

3/4/2019 • 3 minutes to read • [Edit Online](#)

A `timer` messaging block is a single-target `source_block` capable of sending a message to its target after a specified time period has elapsed or at specific intervals.

## Syntax

```
template<class T>
class timer : public Concurrency::details::_Timer, public source_block<single_link_registry<ITarget<T>>>;
```

### Parameters

`T`

The payload type of the output messages of this block.

## Members

### Public Constructors

NAME	DESCRIPTION
<code>timer</code>	Overloaded. Constructs a <code>timer</code> messaging block that will fire a given message after a specified interval.
<code>~timer</code> Destructor	Destroys a <code>timer</code> messaging block.

### Public Methods

NAME	DESCRIPTION
<code>pause</code>	Stops the <code>timer</code> messaging block. If it is a repeating <code>timer</code> messaging block, it can be restarted with a subsequent <code>start()</code> call. For non-repeating timers, this has the same effect as a <code>stop</code> call.
<code>start</code>	Starts the <code>timer</code> messaging block. The specified number of milliseconds after this is called, the specified value will be propagated downstream as a <code>message</code> .
<code>stop</code>	Stops the <code>timer</code> messaging block.

### Protected Methods

NAME	DESCRIPTION
<code>accept_message</code>	Accepts a message that was offered by this <code>timer</code> messaging block, transferring ownership to the caller.
<code>consume_message</code>	Consumes a message previously offered by the <code>timer</code> and reserved by the target, transferring ownership to the caller.

NAME	DESCRIPTION
<a href="#">link_target_notification</a>	A callback that notifies that a new target has been linked to this <code>timer</code> messaging block.
<a href="#">propagate_to_any_targets</a>	Tries to offer the message produced by the <code>timer</code> block to all of the linked targets.
<a href="#">release_message</a>	Releases a previous message reservation. (Overrides <code>source_block::release_message</code> .)
<a href="#">reserve_message</a>	Reserves a message previously offered by this <code>timer</code> messaging block. (Overrides <code>source_block::reserve_message</code> .)
<a href="#">resume_propagation</a>	Resumes propagation after a reservation has been released. (Overrides <code>source_block::resume_propagation</code> .)

## Remarks

For more information, see [Asynchronous Message Blocks](#).

## Inheritance Hierarchy

[ISource](#)

[source\\_block](#)

`timer`

## Requirements

**Header:** agents.h

**Namespace:** concurrency

## accept\_message

Accepts a message that was offered by this `timer` messaging block, transferring ownership to the caller.

```
virtual message<T>* accept_message(runtime_object_identity _MsgId);
```

### Parameters

`_MsgId`

The `runtime_object_identity` of the offered `message` object.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

## consume\_message

Consumes a message previously offered by the `timer` and reserved by the target, transferring ownership to the caller.

```
virtual message<T>* consume_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being consumed.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

Similar to `accept`, but is always preceded by a call to `reserve`.

## link\_target\_notification

A callback that notifies that a new target has been linked to this `timer` messaging block.

```
virtual void link_target_notification(_Inout_ ITarget<T>* _PTarget);
```

### Parameters

*\_PTarget*

A pointer to the newly linked target.

## pause

Stops the `timer` messaging block. If it is a repeating `timer` messaging block, it can be restarted with a subsequent `start()` call. For non-repeating timers, this has the same effect as a `stop` call.

```
void pause();
```

## propagate\_to\_any\_targets

Tries to offer the message produced by the `timer` block to all of the linked targets.

```
virtual void propagate_to_any_targets(_Inout_opt_ message<T> *);
```

## release\_message

Releases a previous message reservation.

```
virtual void release_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being released.

## reserve\_message

Reserves a message previously offered by this `timer` messaging block.

```
virtual bool reserve_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being reserved.

### Return Value

**true** if the message was successfully reserved, **false** otherwise.

### Remarks

After `reserve` is called, if it returns **true**, either `consume` or `release` must be called to either take or release ownership of the message.

## resume\_propagation

Resumes propagation after a reservation has been released.

```
virtual void resume_propagation();
```

## start

Starts the `timer` messaging block. The specified number of milliseconds after this is called, the specified value will be propagated downstream as a `message`.

```
void start();
```

## stop

Stops the `timer` messaging block.

```
void stop();
```

## timer

Constructs a `timer` messaging block that will fire a given message after a specified interval.

```

timer(
    unsigned int _Ms,
    T const& value,
    ITarget<T>* _PTarget = NULL,
    bool _Repeating = false);

timer(
    Scheduler& _Scheduler,
    unsigned int _Ms,
    T const& value,
    _Inout_opt_ ITarget<T>* _PTarget = NULL,
    bool _Repeating = false);

timer(
    ScheduleGroup& _ScheduleGroup,
    unsigned int _Ms,
    T const& value,
    _Inout_opt_ ITarget<T>* _PTarget = NULL,
    bool _Repeating = false);

```

## Parameters

### *\_Ms*

The number of milliseconds that must elapse after the call to start for the specified message to be propagated downstream.

### *value*

The value which will be propagated downstream when the timer elapses.

### *\_PTarget*

The target to which the timer will propagate its message.

### *\_Repeating*

If true, indicates that the timer will fire periodically every `_Ms` milliseconds.

### *\_Scheduler*

The `Scheduler` object within which the propagation task for the `timer` messaging block is scheduled is scheduled.

### *\_ScheduleGroup*

The `ScheduleGroup` object within which the propagation task for the `timer` messaging block is scheduled. The `Scheduler` object used is implied by the schedule group.

## Remarks

The runtime uses the default scheduler if you do not specify the `_Scheduler` or `_ScheduleGroup` parameters.

## ~timer

Destroys a `timer` messaging block.

```
~timer();
```

## See also

[concurrency Namespace](#)

# transformer Class

3/4/2019 • 4 minutes to read • [Edit Online](#)

A `transformer` messaging block is a single-target, multi-source, ordered `propagator_block` which can accept messages of one type and is capable of storing an unbounded number of messages of a different type.

## Syntax

```
template<class _Input, class _Output>
class transformer : public propagator_block<single_link_registry<ITarget<_Output>>,
    multi_link_registry<ISource<_Input>>>>;
```

### Parameters

`_Input`

The payload type of the messages accepted by the buffer.

`_Output`

The payload type of the messages stored and propagated out by the buffer.

## Members

### Public Constructors

NAME	DESCRIPTION
<code>transformer</code>	Overloaded. Constructs a <code>transformer</code> messaging block.
<code>~transformer</code> Destructor	Destroys the <code>transformer</code> messaging block.

### Protected Methods

NAME	DESCRIPTION
<code>accept_message</code>	Accepts a message that was offered by this <code>transformer</code> messaging block, transferring ownership to the caller.
<code>consume_message</code>	Consumes a message previously offered by the <code>transformer</code> and reserved by the target, transferring ownership to the caller.
<code>link_target_notification</code>	A callback that notifies that a new target has been linked to this <code>transformer</code> messaging block.
<code>propagate_message</code>	Asynchronously passes a message from an <code>ISource</code> block to this <code>transformer</code> messaging block. It is invoked by the <code>propagate</code> method, when called by a source block.
<code>propagate_to_any_targets</code>	Executes the transformer function on the input messages.

NAME	DESCRIPTION
<a href="#">release_message</a>	Releases a previous message reservation. (Overrides <a href="#">source_block::release_message</a> .)
<a href="#">reserve_message</a>	Reserves a message previously offered by this <code>transformer</code> messaging block. (Overrides <a href="#">source_block::reserve_message</a> .)
<a href="#">resume_propagation</a>	Resumes propagation after a reservation has been released. (Overrides <a href="#">source_block::resume_propagation</a> .)
<a href="#">send_message</a>	Synchronously passes a message from an <code>ISource</code> block to this <code>transformer</code> messaging block. It is invoked by the <code>send</code> method, when called by a source block.
<a href="#">supports_anonymous_source</a>	Overrides the <code>supports_anonymous_source</code> method to indicate that this block can accept messages offered to it by a source that is not linked. (Overrides <a href="#">ITarget::supports_anonymous_source</a> .)

## Remarks

For more information, see [Asynchronous Message Blocks](#).

## Inheritance Hierarchy

[ISource](#)

[ITarget](#)

[source\\_block](#)

[propagator\\_block](#)

`transformer`

## Requirements

**Header:** `agents.h`

**Namespace:** `concurrency`

## `accept_message`

Accepts a message that was offered by this `transformer` messaging block, transferring ownership to the caller.

```
virtual message<_Output>* accept_message(runtime_object_identity _MsgId);
```

### Parameters

`_MsgId`

The `runtime_object_identity` of the offered `message` object.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

## consume\_message

Consumes a message previously offered by the `transformer` and reserved by the target, transferring ownership to the caller.

```
virtual message<_Output>* consume_message(runtime_object_identity _MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being consumed.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

Similar to `accept`, but is always preceded by a call to `reserve`.

## link\_target\_notification

A callback that notifies that a new target has been linked to this `transformer` messaging block.

```
virtual void link_target_notification(_Inout_ ITarget<_Output> *);
```

## propagate\_message

Asynchronously passes a message from an `ISource` block to this `transformer` messaging block. It is invoked by the `propagate` method, when called by a source block.

```
virtual message_status propagate_message(  
    _Inout_ message<_Input>* _PMessage,  
    _Inout_ ISource<_Input>* _PSource);
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A `message_status` indication of what the target decided to do with the message.

## propagate\_to\_any\_targets

Executes the transformer function on the input messages.

```
virtual void propagate_to_any_targets(_Inout_opt_ message<_Output> *);
```

## release\_message

Releases a previous message reservation.

```
virtual void release_message(runtime_object_identity_MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being released.

## reserve\_message

Reserves a message previously offered by this `transformer` messaging block.

```
virtual bool reserve_message(runtime_object_identity_MsgId);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being reserved.

### Return Value

**true** if the message was successfully reserved, **false** otherwise.

### Remarks

After `reserve` is called, if it returns **true**, either `consume` or `release` must be called to either take or release ownership of the message.

## resume\_propagation

Resumes propagation after a reservation has been released.

```
virtual void resume_propagation();
```

## send\_message

Synchronously passes a message from an `ISource` block to this `transformer` messaging block. It is invoked by the `send` method, when called by a source block.

```
virtual message_status send_message(  
    _Inout_ message<_Input>* _PMessage,  
    _Inout_ ISource<_Input>* _PSource);
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A [message\\_status](#) indication of what the target decided to do with the message.

## supports\_anonymous\_source

Overrides the `supports_anonymous_source` method to indicate that this block can accept messages offered to it

by a source that is not linked.

```
virtual bool supports_anonymous_source();
```

### Return Value

**true** because the block does not postpone offered messages.

## transformer

Constructs a `transformer` messaging block.

```
transformer(  
    _Transform_method const& _Func,  
    _Inout_opt_ ITarget<Output>* _PTarget = NULL);  
  
transformer(  
    _Transform_method const& _Func,  
    _Inout_opt_ ITarget<Output>* _PTarget,  
    filter_method const& _Filter);  
  
transformer(  
    Scheduler& _PScheduler,  
    _Transform_method const& _Func,  
    _Inout_opt_ ITarget<Output>* _PTarget = NULL);  
  
transformer(  
    Scheduler& _PScheduler,  
    _Transform_method const& _Func,  
    _Inout_opt_ ITarget<Output>* _PTarget,  
    filter_method const& _Filter);  
  
transformer(  
    ScheduleGroup& _PScheduleGroup,  
    _Transform_method const& _Func,  
    _Inout_opt_ ITarget<Output>* _PTarget = NULL);  
  
transformer(  
    ScheduleGroup& _PScheduleGroup,  
    _Transform_method const& _Func,  
    _Inout_opt_ ITarget<Output>* _PTarget,  
    filter_method const& _Filter);
```

### Parameters

*\_Func*

A function that will be invoked for each accepted message.

*\_PTarget*

A pointer to a target block to link with the transformer.

*\_Filter*

A filter function which determines whether offered messages should be accepted.

*\_PScheduler*

The `Scheduler` object within which the propagation task for the `transformer` messaging block is scheduled.

*\_PScheduleGroup*

The `ScheduleGroup` object within which the propagation task for the `transformer` messaging block is scheduled.

The `Scheduler` object used is implied by the schedule group.

### Remarks

The runtime uses the default scheduler if you do not specify the `_PScheduler` or `_PScheduleGroup` parameters.

The type `_Transform_method` is a functor with signature `_Output (_Input const &)` which is invoked by this `transformer` messaging block to process a message.

The type `filter_method` is a functor with signature `bool (_Input const &)` which is invoked by this `transformer` messaging block to determine whether or not it should accept an offered message.

## ~transformer

Destroys the `transformer` messaging block.

```
~transformer();
```

## See also

[concurrency Namespace](#)

[call Class](#)

# unbounded\_buffer Class

3/4/2019 • 5 minutes to read • [Edit Online](#)

An `unbounded_buffer` messaging block is a multi-target, multi-source, ordered `propagator_block` capable of storing an unbounded number of messages.

## Syntax

```
template<
    class      _Type
>
class unbounded_buffer : public propagator_block<multi_link_registry<ITarget<      _Type>>,
    multi_link_registry<ISource<      _Type>>>>;
```

### Parameters

*\_Type*

The payload type of the messages stored and propagated by the buffer.

## Members

### Public Constructors

NAME	DESCRIPTION
<code>unbounded_buffer</code>	Overloaded. Constructs an <code>unbounded_buffer</code> messaging block.
<code>~unbounded_buffer</code> Destructor	Destroys the <code>unbounded_buffer</code> messaging block.

### Public Methods

NAME	DESCRIPTION
<code>dequeue</code>	Removes an item from the <code>unbounded_buffer</code> messaging block.
<code>enqueue</code>	Adds an item to the <code>unbounded_buffer</code> messaging block.

### Protected Methods

NAME	DESCRIPTION
<code>accept_message</code>	Accepts a message that was offered by this <code>unbounded_buffer</code> messaging block, transferring ownership to the caller.
<code>consume_message</code>	Consumes a message previously offered by the <code>unbounded_buffer</code> messaging block and reserved by the target, transferring ownership to the caller.

NAME	DESCRIPTION
<a href="#">link_target_notification</a>	A callback that notifies that a new target has been linked to this <code>unbounded_buffer</code> messaging block.
<a href="#">process_input_messages</a>	Places the <code>message</code> <code>_PMessage</code> in this <code>unbounded_buffer</code> messaging block and tries to offer it to all of the linked targets.
<a href="#">propagate_message</a>	Asynchronously passes a message from an <code>ISource</code> block to this <code>unbounded_buffer</code> messaging block. It is invoked by the <code>propagate</code> method, when called by a source block.
<a href="#">propagate_output_messages</a>	Places the <code>message</code> <code>_PMessage</code> in this <code>unbounded_buffer</code> messaging block and tries to offer it to all of the linked targets. (Overrides <a href="#">source_block::propagate_output_messages</a> .)
<a href="#">release_message</a>	Releases a previous message reservation. (Overrides <a href="#">source_block::release_message</a> .)
<a href="#">reserve_message</a>	Reserves a message previously offered by this <code>unbounded_buffer</code> messaging block. (Overrides <a href="#">source_block::reserve_message</a> .)
<a href="#">resume_propagation</a>	Resumes propagation after a reservation has been released. (Overrides <a href="#">source_block::resume_propagation</a> .)
<a href="#">send_message</a>	Synchronously passes a message from an <code>ISource</code> block to this <code>unbounded_buffer</code> messaging block. It is invoked by the <code>send</code> method, when called by a source block.
<a href="#">supports_anonymous_source</a>	Overrides the <code>supports_anonymous_source</code> method to indicate that this block can accept messages offered to it by a source that is not linked. (Overrides <a href="#">ITarget::supports_anonymous_source</a> .)

For more information, see [Asynchronous Message Blocks](#).

## Inheritance Hierarchy

[ISource](#)

[ITarget](#)

[source\\_block](#)

[propagator\\_block](#)

`unbounded_buffer`

## Requirements

**Header:** `agents.h`

**Namespace:** `concurrency`

## accept\_message

Accepts a message that was offered by this `unbounded_buffer` messaging block, transferring ownership to the caller.

```
virtual message<_Type> * accept_message(  
    runtime_object_identity      _MsgId  
);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the offered `message` object.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

## consume\_message

Consumes a message previously offered by the `unbounded_buffer` messaging block and reserved by the target, transferring ownership to the caller.

```
virtual message<_Type> * consume_message(  
    runtime_object_identity      _MsgId  
);
```

### Parameters

*\_MsgId*

The `runtime_object_identity` of the `message` object being consumed.

### Return Value

A pointer to the `message` object that the caller now has ownership of.

### Remarks

Similar to `accept`, but is always preceded by a call to `reserve`.

## dequeue

Removes an item from the `unbounded_buffer` messaging block.

```
_Type dequeue();
```

### Return Value

The payload of the message removed from the `unbounded_buffer`.

## enqueue

Adds an item to the `unbounded_buffer` messaging block.

```
bool enqueue(  
    _Type const&                _Item  
);
```

## Parameters

*\_Item*

The item to add.

## Return Value

**true** if the item was accepted, **false** otherwise.

# link\_target\_notification

A callback that notifies that a new target has been linked to this `unbounded_buffer` messaging block.

```
virtual void link_target_notification(  
    _Inout_ ITarget<_Type> *          _PTarget  
);
```

## Parameters

*\_PTarget*

A pointer to the newly linked target.

# propagate\_message

Asynchronously passes a message from an `ISource` block to this `unbounded_buffer` messaging block. It is invoked by the `propagate` method, when called by a source block.

```
virtual message_status propagate_message(  
    _Inout_ message<_Type> *          _PMessage,  
    _Inout_ ISource<_Type> *          _PSource  
);
```

## Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

## Return Value

A `message_status` indication of what the target decided to do with the message.

# propagate\_output\_messages

Places the `message` `_PMessage` in this `unbounded_buffer` messaging block and tries to offer it to all of the linked targets.

```
virtual void propagate_output_messages();
```

## Remarks

If another message is already ahead of this one in the `unbounded_buffer`, propagation to linked targets will not occur until any earlier messages have been accepted or consumed. The first linked target to successfully `accept` or `consume` the message takes ownership, and no other target can then get the message.

# process\_input\_messages

Places the `message` `_PMessage` in this `unbounded_buffer` messaging block and tries to offer it to all of the linked targets.

```
virtual void process_input_messages(  
    _Inout_ message<_Type> *          _PMessage  
);
```

### Parameters

`_PMessage`

A pointer to the message that is to be processed.

## release\_message

Releases a previous message reservation.

```
virtual void release_message(  
    runtime_object_identity            _MsgId  
);
```

### Parameters

`_MsgId`

The `runtime_object_identity` of the `message` object being released.

## reserve\_message

Reserves a message previously offered by this `unbounded_buffer` messaging block.

```
virtual bool reserve_message(  
    runtime_object_identity            _MsgId  
);
```

### Parameters

`_MsgId`

The `runtime_object_identity` of the `message` object being reserved.

### Return Value

**true** if the message was successfully reserved, **false** otherwise.

### Remarks

After `reserve` is called, if it returns **true**, either `consume` or `release` must be called to either take or release ownership of the message.

## resume\_propagation

Resumes propagation after a reservation has been released.

```
virtual void resume_propagation();
```

## send\_message

Synchronously passes a message from an `ISource` block to this `unbounded_buffer` messaging block. It is

invoked by the `send` method, when called by a source block.

```
virtual message_status send_message(  
    _Inout_ message<_Type> *          _PMessage,  
    _Inout_ ISource<_Type> *          _PSource  
);
```

### Parameters

*\_PMessage*

A pointer to the `message` object.

*\_PSource*

A pointer to the source block offering the message.

### Return Value

A `message_status` indication of what the target decided to do with the message.

## supports\_anonymous\_source

Overrides the `supports_anonymous_source` method to indicate that this block can accept messages offered to it by a source that is not linked.

```
virtual bool supports_anonymous_source();
```

### Return Value

**true** because the block does not postpone offered messages.

## unbounded\_buffer

Constructs an `unbounded_buffer` messaging block.

```
unbounded_buffer();  
  
unbounded_buffer(  
    filter_method const&          _Filter  
);  
  
unbounded_buffer(  
    Scheduler&                    _PScheduler  
);  
  
unbounded_buffer(  
    Scheduler&                    _PScheduler,  
    filter_method const&          _Filter  
);  
  
unbounded_buffer(  
    ScheduleGroup&                _PScheduleGroup  
);  
  
unbounded_buffer(  
    ScheduleGroup&                _PScheduleGroup,  
    filter_method const&          _Filter  
);
```

### Parameters

*\_Filter*

A filter function which determines whether offered messages should be accepted.

#### *\_PScheduler*

The `Scheduler` object within which the propagation task for the `unbounded_buffer` messaging block is scheduled.

#### *\_PScheduleGroup*

The `ScheduleGroup` object within which the propagation task for the `unbounded_buffer` messaging block is scheduled. The `Scheduler` object used is implied by the schedule group.

#### **Remarks**

The runtime uses the default scheduler if you do not specify the `_PScheduler` or `_PScheduleGroup` parameters.

The type `filter_method` is a functor with signature `bool (_Type const &)` which is invoked by this `unbounded_buffer` messaging block to determine whether or not it should accept an offered message.

## `~unbounded_buffer`

Destroys the `unbounded_buffer` messaging block.

```
~unbounded_buffer();
```

## See also

[concurrency Namespace](#)

[overwrite\\_buffer Class](#)

[single\\_assignment Class](#)

# unsupported\_os Class

3/4/2019 • 2 minutes to read • [Edit Online](#)

This class describes an exception thrown when an unsupported operating system is used.

## Syntax

```
class unsupported_os : public std::exception;
```

## Members

### Public Constructors

NAME	DESCRIPTION
<a href="#">unsupported_os</a>	Overloaded. Constructs an <a href="#">unsupported_os</a> object.

## Inheritance Hierarchy

[exception](#)

[unsupported\\_os](#)

## Requirements

**Header:** conacr.h

**Namespace:** concurrency

## unsupported\_os

Constructs an [unsupported\\_os](#) object.

```
explicit _CRTIMP unsupported_os(_In_z_ const char* _Message) throw();  
unsupported_os() throw();
```

### Parameters

*\_Message*

A descriptive message of the error.

## See also

[concurrency Namespace](#)

# std Namespace

3/4/2019 • 2 minutes to read • [Edit Online](#)

## Syntax

```
namespace std;
```

## Members

### Functions

NAME	DESCRIPTION
<a href="#">make_exception_ptr</a> Function	

## Requirements

**Header:** `ppltasks.h`

## See also

[Reference](#)

# make\_exception\_ptr Function

3/4/2019 • 2 minutes to read • [Edit Online](#)

## Syntax

```
template<class _E>  
exception_ptr make_exception_ptr(_E _Except);
```

### Parameters

*\_E*

Exception type.

*\_Except*

Exception value.

## Return Value

## Requirements

**Header:** `ppltasks.h`

**Namespace:** `std`

## See also

[std Namespace](#)

# stdx Namespace

3/4/2019 • 2 minutes to read • [Edit Online](#)

## Syntax

```
namespace stdx;
```

## Members

### Functions

NAME	DESCRIPTION
<a href="#">declval Function</a>	

## Requirements

**Header:** `ppltasks.h`

## See also

[Reference](#)

# declval Function

3/4/2019 • 2 minutes to read • [Edit Online](#)

## Syntax

```
template<class _T>  
_T&& declval();
```

### Parameters

`_T`

## Return Value

## Requirements

**Header:** `ppltasks.h`

**Namespace:** `stdx`

## See also

[stdx Namespace](#)

# OpenMP in Visual C++

4/22/2019 • 2 minutes to read • [Edit Online](#)

The OpenMP C and C++ application program interface lets you write applications that effectively use multiple processors. Visual C++ supports the OpenMP 2.0 standard.

## In This Section

### [Library Reference](#)

Provides links to constructs used in the OpenMP API.

### [C and C++ Application Program Interface](#)

Discusses the OpenMP C and C++ API, as documented in the version 2.0 specification from the OpenMP Architecture Review Board.

## Related Sections

### [/openmp \(Enable OpenMP 2.0 Support\)](#)

Causes the compiler to process `#pragma omp`.

### [Predefined Macros](#)

Names the predefined ANSI C and Microsoft C++ implementation macros. See the `_OPENMP` macro.

# SIMD Extension

4/22/2019 • 3 minutes to read • [Edit Online](#)

Visual C++ currently supports the OpenMP 2.0 standard, however Visual Studio 2019 also now offers SIMD functionality.

## NOTE

To use SIMD, compile with the `-openmp:experimental` switch that enables additional OpenMP features not available when using the `-openmp` switch.

The `-openmp:experimental` switch subsumes `-openmp`, meaning all OpenMP 2.0 features are included in its use.

For more information, see [SIMD Extension to C++ OpenMP in Visual Studio](#).

## OpenMP SIMD in Visual C++

OpenMP SIMD, introduced in the OpenMP 4.0 standard, targets making vector-friendly loops. By using the `simd` directive before a loop, the compiler can ignore vector dependencies, make the loop as vector-friendly as possible, and respect the users' intention to have multiple loop iterations executed simultaneously.

```
#pragma omp simd
for (i = 0; i < count; i++)
{
    a[i] = a[i-1] + 1;
    b[i] = *c + 1;
    bar(i);
}
```

Visual C++ provides similar non-OpenMP loop pragmas like `#pragma vector` and `#pragma ivdep`, however with OpenMP SIMD, the compiler can do more, like:

- Always allowed to ignore present vector dependencies.
- `/fp:fast` is enabled within the loop.
- Outer loops and loops with function calls are vector-friendly.
- Nested loops can be coalesced into one loop and made vector-friendly.
- Hybrid acceleration with `#pragma omp for simd` to enable coarse-grained multi-threading and fine-grained vectors.

For vector-friendly loops, the compiler remains silent unless you use a vector-support log switch:

```
cl -O2 -openmp:experimental -Qvec-report:2 mycode.cpp
```

```
mycode.cpp(84) : info C5002: Omp simd loop not vectorized due to reason '1200'
mycode.cpp(90) : info C5002: Omp simd loop not vectorized due to reason '1200'
mycode.cpp(96) : info C5001: Omp simd loop vectorized
```

For non-vector-friendly loops, the compiler issues each a message:

```
cl -O2 -openmp:experimental mycode.cpp
```

```
mycode.cpp(84) : info C5002: Omp simd loop not vectorized due to reason '1200'  
mycode.cpp(90) : info C5002: Omp simd loop not vectorized due to reason '1200'
```

## Clauses

The OpenMP SIMD directive can also take the following clauses to enhance vector-support:

DIRECTIVE	DESCRIPTION
<code>simdlen(length)</code>	Specify the number of vector lanes.
<code>safelen(length)</code>	Specify the vector dependency distance.
<code>linear(list[ : linear-step] )</code>	The linear mapping from loop induction variable to array subscription.
<code>aligned(list[ : alignment])</code>	The alignment of data.
<code>private(list)</code>	Specify data privatization.
<code>lastprivate(list)</code>	Specify data privatization with final value from the last iteration.
<code>reduction(reduction-identifier:list)</code>	Specify customized reduction operations.
<code>collapse(n)</code>	Coalescing loop nest.

### NOTE

Non-effective SIMD clauses are parsed and ignored by the compiler with a warning.

For example, use of the following code issues a warning:

```
#pragma omp simd simdlen(8)  
for (i = 0; i < count; i++)  
{  
    a[i] = a[i-1] + 1;  
    b[i] = *c + 1;  
    bar(i);  
}
```

```
warning C4849: OpenMP 'simdlen' clause ignored in 'simd' directive
```

## Example

The OpenMP SIMD directive provides users a way to dictate the compiler make loops vector-friendly. By annotating a loop with the OpenMP SIMD directive, users intend to have multiple loop iterations executed simultaneously.

For example, the following loop is annotated with the OpenMP SIMD directive. There's no perfect parallelism among loop iterations since there's a backward dependency from `a[i]` to `a[i-1]`, but because of the SIMD directive the compiler is still allowed to pack consecutive iterations of the first statement into one vector instruction and run

them in parallel.

```
#pragma omp simd
for (i = 0; i < count; i++)
{
    a[i] = a[i-1] + 1;
    b[i] = *c + 1;
    bar(i);
}
```

Therefore, the following transformed vector form of the loop is **legal** because the compiler keeps the sequential behavior of each original loop iteration. In other words, `a[i]` is executed after `a[i-1]`, `b[i]` is after `a[i]` and the call to `bar` happens last.

```
for (i = 0; i < count; i+=4)
{
    a[i:i+3] = a[i-1:i+2] + 1;
    b[i:i+3] = *c + 1;
    bar(i);
    bar(i+1);
    bar(i+2);
    bar(i+3);
}
```

It's **not legal** to move the memory reference `*c` out of the loop if it may alias with `a[i]` or `b[i]`. It's also not legal to reorder the statements inside one original iteration if it breaks the sequential dependency. For example, the following transformed loop isn't legal:

```
c = b;
t = *c;
for (i = 0; i < count; i+=4)
{
    a[i:i+3] = a[i-1:i+2] + 1;
    bar(i); // illegal to reorder if bar[i] depends on b[i]
    b[i:i+3] = t + 1; // illegal to move *c out of the loop
    bar(i+1);
    bar(i+2);
    bar(i+3);
}
```

## See also

[/openmp \(Enable OpenMP 2.0 Support\)](#)

# OpenMP C and C++ Application Program Interface

4/22/2019 • 2 minutes to read • [Edit Online](#)

Discusses the OpenMP C and C++ API, as documented in the version 2.0 specification from the OpenMP Architecture Review Board.

Version 2.0 March 2002

Copyright 1997-2002 OpenMP Architecture Review Board.

Permission to copy without fee all or part of this material is granted, provided the OpenMP Architecture Review Board copyright notice and the title of this document appear. Notice is given that copying is by permission of OpenMP Architecture Review Board.

## Contents

1. [Introduction](#)
2. [Directives](#)
3. [Run-time library functions](#)
4. [Environment variables](#)

### Appendices

- A. [Examples](#)
- B. [Stubs for run-time library functions](#)
- C. [OpenMP C and C++ grammar](#)
- D. [The schedule clause](#)
- E. [Implementation-defined behaviors in OpenMP C/C++](#)
- F. [New features and clarifications in version 2.0](#)

## See also

[OpenMP](#)

# 1. Introduction

1/28/2019 • 8 minutes to read • [Edit Online](#)

This document specifies a collection of compiler directives, library functions, and environment variables that you can use to specify shared-memory parallelism in C and C++ programs. The functionality described in this document is collectively known as the *OpenMP C/C++ Application Program Interface (API)*. The goal of this specification is to provide a model for parallel programming that allows a program to be portable across shared-memory architectures from different vendors. Compilers from many vendors support the OpenMP C/C++ API. More information about OpenMP, including the *OpenMP Fortran Application Program Interface*, can be found at the following web site:

<https://www.openmp.org>

The directives, library functions, and environment variables defined in this document allow you to create and manage parallel programs while allowing portability. The directives extend the C and C++ sequential programming model with single program multiple data (SPMD) constructs, work-sharing constructs, and synchronization constructs. They also support the sharing and privatization of data. Compilers that support the OpenMP C and C++ API include a command-line option to the compiler that activates and allows interpretation of all OpenMP compiler directives.

## 1.1 Scope

This specification covers only user-directed parallelization, wherein you explicitly define what actions the compiler and run-time system take to execute the program in parallel. OpenMP C and C++ implementations aren't required to check for dependencies, conflicts, deadlocks, race conditions, or other problems that result in incorrect program execution. You are responsible for ensuring that the application using the OpenMP C and C++ API constructs executes correctly. Compiler-generated automatic parallelization and directives to the compiler to assist such parallelization aren't covered in this document.

## 1.2 Definition of terms

The following terms are used in this document:

- barrier

A synchronization point that all threads in a team must reach. Each thread waits until all threads in the team arrive at this point. There are explicit barriers identified by directives and implicit barriers created by the implementation.

- construct

A construct is a statement. It consists of a directive, followed by a structured block. Some directives aren't part of a construct. (See *openmp-directive* in [appendix C](#)).

- directive

A C or C++ `#pragma` followed by the `omp` identifier, other text, and a new line. The directive specifies program behavior.

- dynamic extent

All statements in the *lexical extent*, plus any statement inside a function that's executed as a result of the execution of statements within the lexical extent. A dynamic extent is also referred to as a *region*.

- lexical extent

Statements lexically held within a *structured block*.

- master thread

The thread that creates a team when a *parallel region* is entered.

- parallel region

Statements that bind to an OpenMP parallel construct and may be executed by many threads.

- private

A private variable names a block of storage that's unique to the thread making the reference. There are several ways to specify that a variable is private: a definition within a parallel region, a `threadprivate` directive, a `private`, `firstprivate`, `lastprivate`, or `reduction` clause, or use of the variable as a `for` loop control variable in a `for` loop immediately following a `for` or `parallel for` directive.

- region

A dynamic extent.

- serial region

Statements executed only by the *master thread* outside of the dynamic extent of any *parallel region*.

- serialize

To execute a parallel construct with:

- a team of threads consisting of only a single thread (which is the master thread for that parallel construct),
- serial order of execution for the statements within the structured block (the same order as if the block were not part of a parallel construct), and
- no effect on the value returned by `omp_in_parallel()` (apart from the effects of any nested parallel constructs).

- shared

A shared variable names a single block of storage. All threads in a team that access this variable also access this single block of storage.

- structured block

A structured block is a statement (single or compound) that has a single entry and a single exit. If there's a jump into or out of a statement, that statement is a structured block. (This rule includes a call to `longjmp` (3C) or the use of `throw`, although a call to `exit` is permitted.) If its execution always begins at the opening `{` and always ends at the closing `}`, a compound statement is a structured block. An expression statement, selection statement, iteration statement, or `try` block is a structured block if the corresponding compound statement obtained by enclosing it in `{` and `}` would be a structured block. A jump statement, labeled statement, or declaration statement isn't a structured block.

- team

One or more threads cooperating in the execution of a construct.

- thread

An execution entity having a serial flow of control, a set of private variables, and access to shared variables.

- variable

An identifier, optionally qualified by namespace names, that names an object.

## 1.3 Execution model

OpenMP uses the fork-join model of parallel execution. Although this fork-join model can be useful for solving various problems, it's tailored for large array-based applications. OpenMP is intended to support programs that execute correctly both as parallel programs (many threads of execution and a full OpenMP support library). It's also for programs that execute correctly as sequential programs (directives ignored and a simple OpenMP stubs library). However, it's possible and permitted to develop a program that doesn't behave correctly when executed sequentially. Furthermore, different degrees of parallelism may result in different numeric results because of changes in the association of numeric operations. For example, a serial addition reduction may have a different pattern of addition associations than a parallel reduction. These different associations may change the results of floating-point addition.

A program written with the OpenMP C/C++ API begins execution as a single thread of execution called the *master thread*. The master thread executes in a serial region until the first parallel construct is encountered. In the OpenMP C/C++ API, the `parallel` directive constitutes a parallel construct. When a parallel construct is encountered, the master thread creates a team of threads, and the master becomes master of the team. Each thread in the team executes the statements in the dynamic extent of a parallel region, except for the work-sharing constructs. All threads in the team must encounter work-sharing constructs in the same order, and one or more of the threads executes the statements within the associated structured block. The barrier implied at the end of a work-sharing construct without a `nowait` clause is executed by all threads in the team.

If a thread modifies a shared object, it affects not only its own execution environment, but also those of the other threads in the program. The modification is guaranteed to be complete, from the point of view of another thread, at the next sequence point (as defined in the base language) only if the object is declared to be volatile. Otherwise, the modification is guaranteed to be complete after first the modifying thread. The other threads then (or concurrently) see a `flush` directive that specifies the object (either implicitly or explicitly). When the `flush` directives that are implied by other OpenMP directives don't guarantee the correct ordering of side effects, it's the programmer's responsibility to supply additional, explicit `flush` directives.

Upon completion of the parallel construct, the threads in the team synchronize at an implicit barrier, and only the master thread continues execution. Any number of parallel constructs can be specified in a single program. As a result, a program may fork and join many times during execution.

The OpenMP C/C++ API allows programmers to use directives in functions called from within parallel constructs. Directives that don't appear in the lexical extent of a parallel construct but may lie in the dynamic extent are called *orphaned* directives. With orphaned directives, programmers can execute major portions of their program in parallel, with only minimal changes to the sequential program. With this functionality, you can code parallel constructs at the top levels of the program call tree and use directives to control execution in any of the called functions.

Unsynchronized calls to C and C++ output functions that write to the same file may result in output in which data written by different threads appears in nondeterministic order. Similarly, unsynchronized calls to input functions that read from the same file may read data in nondeterministic order. Unsynchronized use of I/O, such that each thread accesses a different file, produces the same results as serial execution of the I/O functions.

## 1.4 Compliance

An implementation of the OpenMP C/C++ API is *OpenMP-compliant* if it recognizes and preserves the semantics of all the elements of this specification, as laid out in Chapters 1, 2, 3, 4, and Appendix C. Appendices A, B, D, E, and F are for information purposes only and aren't part of the specification. Implementations that include only a subset of the API aren't OpenMP-compliant.

The OpenMP C and C++ API is an extension to the base language that's supported by an implementation. If the base language doesn't support a language construct or extension that appears in this document, the OpenMP implementation isn't required to support it.

All standard C and C++ library functions and built-in functions (that is, functions of which the compiler has specific knowledge) must be thread-safe. Unsynchronized use of thread-safe functions by different threads inside a parallel region doesn't produce undefined behavior. However, the behavior might not be the same as in a serial region. (A random number generation function is an example.)

The OpenMP C/C++ API specifies that certain behavior is *implementation-defined*. A conforming OpenMP implementation is required to define and document its behavior in these cases. For a list of implementation-defined behaviors, see [appendix E](#).

## 1.5 Normative references

- ISO/IEC 9899:1999, *Information Technology - Programming Languages - C*. This OpenMP API specification refers to ISO/IEC 9899:1999 as C99.
- ISO/IEC 9899:1990, *Information Technology - Programming Languages - C*. This OpenMP API specification refers to ISO/IEC 9899:1990 as C90.
- ISO/IEC 14882:1998, *Information Technology - Programming Languages - C++*. This OpenMP API specification refers to ISO/IEC 14882:1998 as C++.

Where this OpenMP API specification refers to C, reference is made to the base language supported by the implementation.

## 1.6 Organization

- [Run-time library functions](#)
- [Environment variables](#)
- [Implementation-defined behaviors in OpenMP C/C++](#)
- [New features in OpenMP C/C++ version 2.0](#)

## 2. Directives

1/28/2019 • 35 minutes to read • [Edit Online](#)

Directives are based on `#pragma` directives defined in the C and C++ standards. Compilers that support the OpenMP C and C++ API will include a command-line option that activates and allows interpretation of all OpenMP compiler directives.

### 2.1 Directive format

The syntax of an OpenMP directive is formally specified by the grammar in [appendix C](#), and informally as follows:

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

Each directive starts with `#pragma omp`, to reduce the potential for conflict with other (non-OpenMP or vendor extensions to OpenMP) pragma directives with the same names. The rest of the directive follows the conventions of the C and C++ standards for compiler directives. In particular, white space can be used before and after the `#`, and sometimes white space must be used to separate the words in a directive. Preprocessing tokens following the `#pragma omp` are subject to macro replacement.

Directives are case-sensitive. The order in which clauses appear in directives isn't significant. Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause. If *variable-list* appears in a clause, it must specify only variables. Only one *directive-name* can be specified per directive. For example, the following directive isn't allowed:

```
/* ERROR - multiple directive names not allowed */  
#pragma omp parallel barrier
```

An OpenMP directive applies to at most one succeeding statement, which must be a structured block.

### 2.2 Conditional compilation

The `_OPENMP` macro name is defined by OpenMP-compliant implementations as the decimal constant *yyyymm*, which will be the year and month of the approved specification. This macro must not be the subject of a `#define` or a `#undef` preprocessing directive.

```
#ifdef _OPENMP  
iam = omp_get_thread_num() + index;  
#endif
```

If vendors define extensions to OpenMP, they may specify additional predefined macros.

### 2.3 parallel construct

The following directive defines a parallel region, which is a region of the program that's to be executed by many threads in parallel. This directive is the fundamental construct that starts parallel execution.

```
#pragma omp parallel [clause[ [,] clause] ...] new-line structured-block
```

The *clause* is one of the following:

- `if( scalar-expression )`
- `private( variable-list )`
- `firstprivate( variable-list )`
- `default(shared | none)`
- `shared( variable-list )`
- `copyin( variable-list )`
- `reduction( operator : variable-list )`
- `num_threads( integer-expression )`

When a thread gets to a parallel construct, a team of threads is created if one of the following cases is true:

- No `if` clause is present.
- The `if` expression evaluates to a nonzero value.

This thread becomes the master thread of the team, with a thread number of 0, and all threads in the team, including the master thread, execute the region in parallel. If the value of the `if` expression is zero, the region is serialized.

To determine the number of threads that are requested, the following rules will be considered in order. The first rule whose condition is met will be applied:

1. If the `num_threads` clause is present, then the value of the integer expression is the number of threads requested.
2. If the `omp_set_num_threads` library function has been called, then the value of the argument in the most recently executed call is the number of threads requested.
3. If the environment variable `OMP_NUM_THREADS` is defined, then the value of this environment variable is the number of threads requested.
4. If none of the methods above is used, then the number of threads requested is implementation-defined.

If the `num_threads` clause is present then it supersedes the number of threads requested by the `omp_set_num_threads` library function or the `OMP_NUM_THREADS` environment variable only for the parallel region it's applied to. Later parallel regions aren't affected by it.

The number of threads that execute the parallel region also depends upon whether dynamic adjustment of the number of threads is enabled. If dynamic adjustment is disabled, then the requested number of threads will execute the parallel region. If dynamic adjustment is enabled then the requested number of threads is the maximum number of threads that may execute the parallel region.

If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads requested for the parallel region is more than the number that the run-time system can supply, the behavior of the program is implementation-defined. An implementation may, for example, interrupt the execution of the program, or it may serialize the parallel region.

The `omp_set_dynamic` library function and the `OMP_DYNAMIC` environment variable can be used to enable and disable dynamic adjustment of the number of threads.

The number of physical processors actually hosting the threads at any given time is implementation-defined. Once created, the number of threads in the team stays constant for the duration of that parallel region. It can be changed either explicitly by the user or automatically by the run-time system from one parallel region to another.

The statements contained within the dynamic extent of the parallel region are executed by each thread, and each thread can execute a path of statements that's different from the other threads. Directives encountered outside the

lexical extent of a parallel region are referred to as orphaned directives.

There's an implied barrier at the end of a parallel region. Only the master thread of the team continues execution at the end of a parallel region.

If a thread in a team executing a parallel region encounters another parallel construct, it creates a new team, and it becomes the master of that new team. Nested parallel regions are serialized by default. As a result, by default, a nested parallel region is executed by a team composed of one thread. The default behavior may be changed by using either the runtime library function `omp_set_nested` or the environment variable `OMP_NESTED`. However, the number of threads in a team that execute a nested parallel region is implementation-defined.

Restrictions to the `parallel` directive are as follows:

- At most, one `if` clause can appear on the directive.
- It's unspecified whether any side effects inside the if expression or `num_threads` expression occur.
- A `throw` executed inside a parallel region must cause execution to resume within the dynamic extent of the same structured block, and it must be caught by the same thread that threw the exception.
- Only a single `num_threads` clause can appear on the directive. The `num_threads` expression is evaluated outside the context of the parallel region, and must evaluate to a positive integer value.
- The order of evaluation of the `if` and `num_threads` clauses is unspecified.

### Cross-references

- `private`, `firstprivate`, `default`, `shared`, `copyin`, and `reduction` clauses ([section 2.7.2](#))
- `OMP_NUM_THREADS` environment variable
- `omp_set_dynamic` library function
- `OMP_DYNAMIC` environment variable
- `omp_set_nested` function
- `OMP_NESTED` environment variable
- `omp_set_num_threads` library function

## 2.4 Work-sharing constructs

A work-sharing construct distributes the execution of the associated statement among the members of the team that encounter it. The work-sharing directives don't launch new threads, and there's no implied barrier on entry to a work-sharing construct.

The sequence of work-sharing constructs and `barrier` directives encountered must be the same for every thread in a team.

OpenMP defines the following work-sharing constructs, and these constructs are described in the sections that follow:

- `for` directive
- `sections` directive
- `single` directive

### 2.4.1 for construct

The `for` directive identifies an iterative work-sharing construct that specifies that the iterations of the associated loop will be executed in parallel. The iterations of the `for` loop are distributed across threads that already exist in the team executing the parallel construct to which it binds. The syntax of the `for` construct is as follows:

```
#pragma omp for [clause[[],] clause] ... ] new-line for-loop
```

The clause is one of the following:

- `private( variable-list )`
- `firstprivate( variable-list )`
- `lastprivate( variable-list )`
- `reduction( operator : variable-list )`
- `ordered`
- `schedule( kind [ , chunk_size ] )`
- `nowait`

The `for` directive places restrictions on the structure of the corresponding `for` loop. Specifically, the corresponding `for` loop must have canonical shape:

```
for ( init-expr ; var logical-op b ; incr-expr )
```

*init-expr*

One of the following:

- `var = lb`
- `integer-type var = lb`

*incr-expr*

One of the following:

- `++ var`
- `var ++`
- `-- var`
- `var --`
- `var += incr`
- `var -= incr`
- `var = var + incr`
- `var = incr + var`
- `var = var - incr`

*var*

A signed integer variable. If this variable would otherwise be shared, it's implicitly made private for the duration of the `for`. Do not modify this variable within the body of the `for` statement. Unless the variable is specified `lastprivate`, its value after the loop is indeterminate.

*logical-op*

One of the following:

- `<`
- `<=`
- `>`
- `>=`

*lb, b, and incr*

Loop invariant integer expressions. There's no synchronization during the evaluation of these expressions, so any evaluated side effects produce indeterminate results.

The canonical form allows the number of loop iterations to be computed on entry to the loop. This computation is made with values in the type of *var*, after integral promotions. In particular, if value of  $b - lb + incr$  can't be represented in that type, the result is indeterminate. Further, if *logical-op* is `<` or `<=`, then *incr-expr* must cause *var* to increase on each iteration of the loop. If *logical-op* is `>` or `>=`, then *incr-expr* must cause *var* to get smaller on each iteration of the loop.

The `schedule` clause specifies how iterations of the `for` loop are divided among threads of the team. The correctness of a program must not depend on which thread executes a particular iteration. The value of *chunk\_size*, if specified, must be a loop invariant integer expression with a positive value. There's no synchronization during the evaluation of this expression, so any evaluated side effects produce indeterminate results. The schedule *kind* can be one of the following values:

Table 2-1: `schedule` clause *kind* values

static	When <code>schedule(static, chunk_size )</code> is specified, iterations are divided into chunks of a size specified by <i>chunk_size</i> . The chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. When no <i>chunk_size</i> is specified, the iteration space is divided into chunks that are approximately equal in size, with one chunk assigned to each thread.
dynamic	When <code>schedule(dynamic, chunk_size )</code> is specified, the iterations are divided into a series of chunks, each containing <i>chunk_size</i> iterations. Each chunk is assigned to a thread that's waiting for an assignment. The thread executes the chunk of iterations and then waits for its next assignment, until no chunks remain to be assigned. The last chunk to be assigned may have a smaller number of iterations. When no <i>chunk_size</i> is specified, it defaults to 1.
guided	When <code>schedule(guided, chunk_size )</code> is specified, the iterations are assigned to threads in chunks with decreasing sizes. When a thread finishes its assigned chunk of iterations, it's dynamically assigned another chunk, until none is left. For a <i>chunk_size</i> of 1, the size of each chunk is approximately the number of unassigned iterations divided by the number of threads. These sizes decrease almost exponentially to 1. For a <i>chunk_size</i> with value <i>k</i> greater than 1, the sizes decrease almost exponentially to <i>k</i> , except that the last chunk may have fewer than <i>k</i> iterations. When no <i>chunk_size</i> is specified, it defaults to 1.
runtime	When <code>schedule(runtime)</code> is specified, the decision regarding scheduling is deferred until runtime. The schedule <i>kind</i> and size of the chunks can be chosen at run time by setting the environment variable <code>OMP_SCHEDULE</code> . If this environment variable isn't set, the resulting schedule is implementation-defined. When <code>schedule(runtime)</code> is specified, <i>chunk_size</i> must not be specified.

In the absence of an explicitly defined `schedule` clause, the default `schedule` is implementation-defined.

An OpenMP-compliant program shouldn't rely on a particular schedule for correct execution. A program shouldn't rely on a schedule *kind* conforming precisely to the description given above, because it's possible to have variations in the implementations of the same schedule *kind* across different compilers. The descriptions can be used to select the schedule that's appropriate for a particular situation.

The `ordered` clause must be present when `ordered` directives bind to the `for` construct.

There's an implicit barrier at the end of a `for` construct unless a `nowait` clause is specified.

Restrictions to the `for` directive are as follows:

- The `for` loop must be a structured block, and, in addition, its execution must not be terminated by a `break` statement.
- The values of the loop control expressions of the `for` loop associated with a `for` directive must be the same for all the threads in the team.
- The `for` loop iteration variable must have a signed integer type.
- Only a single `schedule` clause can appear on a `for` directive.
- Only a single `ordered` clause can appear on a `for` directive.
- Only a single `nowait` clause can appear on a `for` directive.
- It's unspecified if or how often any side effects within the `chunk_size`, `lb`, `b`, or `incr` expressions occur.
- The value of the `chunk_size` expression must be the same for all threads in the team.

#### Cross-references

- `private`, `firstprivate`, `lastprivate`, and `reduction` clauses ([section 2.7.2](#))
- `OMP_SCHEDULE` environment variable
- `ordered` construct
- `schedule` clause

### 2.4.2 sections construct

The `sections` directive identifies a noniterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. Each section is executed once by a thread in the team. The syntax of the `sections` directive is as follows:

```
#pragma omp sections [clause[[,] clause] ...] new-line
{
  [#pragma omp section new-line]
  structured-block
  [#pragma omp section new-linestructured-block ]
  ...
}
```

The clause is one of the following:

- `private( variable-list )`
- `firstprivate( variable-list )`
- `lastprivate( variable-list )`
- `reduction( operator : variable-list )`
- `nowait`

Each section is preceded by a `section` directive, although the `section` directive is optional for the first section.

The `section` directives must appear within the lexical extent of the `sections` directive. There's an implicit barrier at the end of a `sections` construct, unless a `nowait` is specified.

Restrictions to the `sections` directive are as follows:

- A `section` directive must not appear outside the lexical extent of the `sections` directive.

- Only a single `nowait` clause can appear on a `sections` directive.

#### Cross-references

- `private`, `firstprivate`, `lastprivate`, and `reduction` clauses ([section 2.7.2](#))

### 2.4.3 single construct

The `single` directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread). The syntax of the `single` directive is as follows:

```
#pragma omp single [clause[[,] clause] ...] new-linestructured-block
```

The clause is one of the following:

- `private( variable-list )`
- `firstprivate( variable-list )`
- `copyprivate( variable-list )`
- `nowait`

There's an implicit barrier after the `single` construct unless a `nowait` clause is specified.

Restrictions to the `single` directive are as follows:

- Only a single `nowait` clause can appear on a `single` directive.
- The `copyprivate` clause must not be used with the `nowait` clause.

#### Cross-references

- `private`, `firstprivate`, and `copyprivate` clauses ([section 2.7.2](#))

## 2.5 Combined parallel work-sharing constructs

Combined parallel work-sharing constructs are shortcuts for specifying a parallel region that has only one work-sharing construct. The semantics of these directives are the same as explicitly specifying a `parallel` directive followed by a single work-sharing construct.

The following sections describe the combined parallel work-sharing constructs:

- [parallel for](#) directive
- [parallel sections](#) directive

### 2.5.1 parallel for construct

The `parallel for` directive is a shortcut for a `parallel` region that contains only a single `for` directive. The syntax of the `parallel for` directive is as follows:

```
#pragma omp parallel for [clause[[,] clause] ...] new-linefor-loop
```

This directive allows all the clauses of the `parallel` directive and the `for` directive, except the `nowait` clause, with identical meanings and restrictions. The semantics are the same as explicitly specifying a `parallel` directive immediately followed by a `for` directive.

#### Cross-references

- [parallel](#) directive
- [for](#) directive
- [Data attribute clauses](#)

### 2.5.2 parallel sections construct

The `parallel sections` directive provides a shortcut form for specifying a `parallel` region that has only a single `sections` directive. The semantics are the same as explicitly specifying a `parallel` directive immediately followed by a `sections` directive. The syntax of the `parallel sections` directive is as follows:

```
#pragma omp parallel sections [clause[[,] clause] ...] new-line
{
  [#pragma omp section new-line]
  structured-block
  [#pragma omp section new-linestructured-block ]
  ...
}
```

The *clause* can be one of the clauses accepted by the `parallel` and `sections` directives, except the `nowait` clause.

#### Cross-references

- [parallel](#) directive
- [sections](#) directive

## 2.6 Master and synchronization directives

The following sections describe:

- [master](#) construct
- [critical](#) construct
- [barrier](#) directive
- [atomic](#) construct
- [flush](#) directive
- [ordered](#) construct

### 2.6.1 master construct

The `master` directive identifies a construct that specifies a structured block that's executed by the master thread of the team. The syntax of the `master` directive is as follows:

```
#pragma omp master new-linestructured-block
```

Other threads in the team don't execute the associated structured block. There's no implied barrier either on entry to or exit from the master construct.

### 2.6.2 critical construct

The `critical` directive identifies a construct that restricts execution of the associated structured block to a single thread at a time. The syntax of the `critical` directive is as follows:

```
#pragma omp critical [(name)] new-linestructured-block
```

An optional *name* may be used to identify the critical region. Identifiers used to identify a critical region have external linkage and are in a name space that is separate from the name spaces used by labels, tags, members, and ordinary identifiers.

A thread waits at the beginning of a critical region until no other thread is executing a critical region (anywhere in the program) with the same name. All unnamed `critical` directives map to the same unspecified name.

### 2.6.3 barrier directive

The `barrier` directive synchronizes all the threads in a team. When encountered, each thread in the team waits

until all of the others have reached this point. The syntax of the `barrier` directive is as follows:

```
#pragma omp barrier new-line
```

After all threads in the team have encountered the barrier, each thread in the team begins executing the statements after the barrier directive in parallel. Because the `barrier` directive doesn't have a C language statement as part of its syntax, there are some restrictions on its placement within a program. For more information about the formal grammar, see [appendix C](#). The example below illustrates these restrictions.

```
/* ERROR - The barrier directive cannot be the immediate
 *          substatement of an if statement
 */
if (x!=0)
    #pragma omp barrier
    ...

/* OK - The barrier directive is enclosed in a
 *      compound statement.
 */
if (x!=0) {
    #pragma omp barrier
}
```

#### 2.6.4 atomic construct

The `atomic` directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. The syntax of the `atomic` directive is as follows:

```
#pragma omp atomic new-lineexpression-stmt
```

The expression statement must have one of the following forms:

- $x \text{ binop } = \text{ expr}$
- $x \text{ ++}$
- $\text{++ } x$
- $x \text{ --}$
- $\text{-- } x$

In the preceding expressions:

- $x$  is an lvalue expression with scalar type.
- $\text{expr}$  is an expression with scalar type, and it doesn't reference the object designated by  $x$ .
- $\text{binop}$  isn't an overloaded operator and is one of `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<`, or `>>`.

Although it's implementation-defined whether an implementation replaces all `atomic` directives with `critical` directives that have the same unique *name*, the `atomic` directive permits better optimization. Often hardware instructions are available that can perform the atomic update with the least overhead.

Only the load and store of the object designated by  $x$  are atomic; the evaluation of  $\text{expr}$  isn't atomic. To avoid race conditions, all updates of the location in parallel should be protected with the `atomic` directive, except those that are known to be free of race conditions.

Restrictions to the `atomic` directive are as follows:

- All atomic references to the storage location  $x$  throughout the program are required to have a compatible type.

## Examples

```
extern float a[], *p = a, b;
/* Protect against races among multiple updates. */
#pragma omp atomic
a[index[i]] += b;
/* Protect against races with updates through a. */
#pragma omp atomic
p[i] -= 1.0f;

extern union {int n; float x;} u;
/* ERROR - References through incompatible types. */
#pragma omp atomic
u.n++;
#pragma omp atomic
u.x -= 1.0f;
```

### 2.6.5 flush directive

The `flush` directive, whether explicit or implied, specifies a "cross-thread" sequence point at which the implementation is required to ensure that all threads in a team have a consistent view of certain objects (specified below) in memory. This means that previous evaluations of expressions that reference those objects are complete and subsequent evaluations haven't yet begun. For example, compilers must restore the values of the objects from registers to memory, and hardware may need to flush write buffers to memory and reload the values of the objects from memory.

The syntax of the `flush` directive is as follows:

```
#pragma omp flush [(variable-list)] new-line
```

If the objects that require synchronization can all be designated by variables, then those variables can be specified in the optional *variable-list*. If a pointer is present in the *variable-list*, the pointer itself is flushed, not the object the pointer refers to.

A `flush` directive without a *variable-list* synchronizes all shared objects except inaccessible objects with automatic storage duration. (This is likely to have more overhead than a `flush` with a *variable-list*.) A `flush` directive without a *variable-list* is implied for the following directives:

- `barrier`
- At entry to and exit from `critical`
- At entry to and exit from `ordered`
- At entry to and exit from `parallel`
- At exit from `for`
- At exit from `sections`
- At exit from `single`
- At entry to and exit from `parallel for`
- At entry to and exit from `parallel sections`

The directive isn't implied if a `nowait` clause is present. It should be noted that the `flush` directive isn't implied for any of the following:

- At entry to `for`
- At entry to or exit from `master`
- At entry to `sections`
- At entry to `single`

A reference that accesses the value of an object with a volatile-qualified type behaves as if there were a `flush` directive specifying that object at the previous sequence point. A reference that modifies the value of an object with a volatile-qualified type behaves as if there were a `flush` directive specifying that object at the subsequent sequence point.

Because the `flush` directive doesn't have a C language statement as part of its syntax, there are some restrictions on its placement within a program. For more information about the formal grammar, see [appendix C](#). The example below illustrates these restrictions.

```
/* ERROR - The flush directive cannot be the immediate
 *      substatement of an if statement.
 */
if (x!=0)
    #pragma omp flush (x)
    ...

/* OK - The flush directive is enclosed in a
 *      compound statement
 */
if (x!=0) {
    #pragma omp flush (x)
}
```

Restrictions to the `flush` directive are as follows:

- A variable specified in a `flush` directive must not have a reference type.

### 2.6.6 ordered construct

The structured block following an `ordered` directive is executed in the order in which iterations would be executed in a sequential loop. The syntax of the `ordered` directive is as follows:

```
#pragma omp ordered new-linestructured-block
```

An `ordered` directive must be within the dynamic extent of a `for` or `parallel for` construct. The `for` or `parallel for` directive to which the `ordered` construct binds must have an `ordered` clause specified as described in [section 2.4.1](#). In the execution of a `for` or `parallel for` construct with an `ordered` clause, `ordered` constructs are executed strictly in the order in which they would be executed in a sequential execution of the loop.

Restrictions to the `ordered` directive are as follows:

- An iteration of a loop with a `for` construct must not execute the same ordered directive more than once, and it must not execute more than one `ordered` directive.

## 2.7 Data environment

This section presents a directive and several clauses for controlling the data environment during the execution of parallel regions, as follows:

- A `threadprivate` directive is provided to make file- scope, namespace- scope, or static block- scope variables local to a thread.
- Clauses that may be specified on the directives to control the sharing attributes of variables for the duration of the parallel or work-sharing constructs are described in [section 2.7.2](#).

### 2.7.1 threadprivate directive

The `threadprivate` directive makes the named file- scope, namespace- scope, or static block- scope variables specified in the *variable-list* private to a thread. *variable-list* is a comma-separated list of variables that don't have

an incomplete type. The syntax of the `threadprivate` directive is as follows:

```
#pragma omp threadprivate(variable-list) new-line
```

Each copy of a `threadprivate` variable is initialized once, at an unspecified point in the program prior to the first reference to that copy, and in the usual manner (i.e., as the master copy would be initialized in a serial execution of the program). Note that if an object is referenced in an explicit initializer of a `threadprivate` variable, and the value of the object is modified prior to the first reference to a copy of the variable, then the behavior is unspecified.

As with any private variable, a thread must not reference another thread's copy of a `threadprivate` object. During serial regions and master regions of the program, references will be to the master thread's copy of the object.

After the first parallel region executes, the data in the `threadprivate` objects is guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads remains unchanged for all parallel regions.

The restrictions to the `threadprivate` directive are as follows:

- A `threadprivate` directive for file-scope or namespace-scope variables must appear outside any definition or declaration, and must lexically precede all references to any of the variables in its list.
- Each variable in the *variable-list* of a `threadprivate` directive at file or namespace scope must refer to a variable declaration at file or namespace scope that lexically precedes the directive.
- A `threadprivate` directive for static block-scope variables must appear in the scope of the variable and not in a nested scope. The directive must lexically precede all references to any of the variables in its list.
- Each variable in the *variable-list* of a `threadprivate` directive in block scope must refer to a variable declaration in the same scope that lexically precedes the directive. The variable declaration must use the static storage-class specifier.
- If a variable is specified in a `threadprivate` directive in one translation unit, it must be specified in a `threadprivate` directive in every translation unit in which it's declared.
- A `threadprivate` variable must not appear in any clause except the `copyin`, `copyprivate`, `schedule`, `num_threads`, or the `if` clause.
- The address of a `threadprivate` variable isn't an address constant.
- A `threadprivate` variable must not have an incomplete type or a reference type.
- A `threadprivate` variable with non-POD class type must have an accessible, unambiguous copy constructor if it's declared with an explicit initializer.

The following example illustrates how modifying a variable that appears in an initializer can cause unspecified behavior, and also how to avoid this problem by using an auxiliary object and a copy-constructor.

```

int x = 1;
T a(x);
const T b_aux(x); /* Capture value of x = 1 */
T b(b_aux);
#pragma omp threadprivate(a, b)

void f(int n) {
    x++;
    #pragma omp parallel for
    /* In each thread:
    * Object a is constructed from x (with value 1 or 2?)
    * Object b is copy-constructed from b_aux
    */
    for (int i=0; i<n; i++) {
        g(a, b); /* Value of a is unspecified. */
    }
}

```

#### Cross-references

- [dynamic threads](#)
- [OMP\\_DYNAMIC](#) environment variable

### 2.7.2 Data-sharing attribute clauses

Several directives accept clauses that allow a user to control the sharing attributes of variables for the duration of the region. Sharing attribute clauses apply only to variables in the lexical extent of the directive on which the clause appears. Not all of the following clauses are allowed on all directives. The list of clauses that are valid on a particular directive are described with the directive.

If a variable is visible when a parallel or work-sharing construct is encountered, and the variable isn't specified in a sharing attribute clause or `threadprivate` directive, then the variable is shared. Static variables declared within the dynamic extent of a parallel region are shared. Heap allocated memory (for example, using `malloc()` in C or C++ or the `new` operator in C++) is shared. (The pointer to this memory, however, can be either private or shared.) Variables with automatic storage duration declared within the dynamic extent of a parallel region are private.

Most of the clauses accept a *variable-list* argument, which is a comma-separated list of variables that are visible. If a variable referenced in a data-sharing attribute clause has a type derived from a template, and there are no other references to that variable in the program, the behavior is undefined.

All variables that appear within directive clauses must be visible. Clauses may be repeated as needed, but no variable may be specified in more than one clause, except that a variable can be specified in both a `firstprivate` and a `lastprivate` clause.

The following sections describe the data-sharing attribute clauses:

- [private](#)
- [firstprivate](#)
- [lastprivate](#)
- [shared](#)
- [default](#)
- [reduction](#)
- [copyin](#)
- [copyprivate](#)

#### 2.7.2.1 private

The `private` clause declares the variables in *variable-list* to be private to each thread in a team. The syntax of the `private` clause is as follows:

```
private(variable-list)
```

The behavior of a variable specified in a `private` clause is as follows. A new object with automatic storage duration is allocated for the construct. The size and alignment of the new object are determined by the type of the variable. This allocation occurs once for each thread in the team, and a default constructor is invoked for a class object if necessary; otherwise the initial value is indeterminate. The original object referenced by the variable has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and has an indeterminate value upon exit from the construct.

In the lexical extent of the directive construct, the variable references the new private object allocated by the thread.

The restrictions to the `private` clause are as follows:

- A variable with a class type that's specified in a `private` clause must have an accessible, unambiguous default constructor.
- A variable specified in a `private` clause must not have a `const`-qualified type unless it has a class type with a `mutable` member.
- A variable specified in a `private` clause must not have an incomplete type or a reference type.
- Variables that appear in the `reduction` clause of a `parallel` directive can't be specified in a `private` clause on a work-sharing directive that binds to the parallel construct.

#### 2.7.2.2 `firstprivate`

The `firstprivate` clause provides a superset of the functionality provided by the `private` clause. The syntax of the `firstprivate` clause is as follows:

```
firstprivate(variable-list)
```

Variables specified in *variable-list* have `private` clause semantics, as described in [section 2.7.2.1](#). The initialization or construction happens as if it were done once per thread, prior to the thread's execution of the construct. For a `firstprivate` clause on a parallel construct, the initial value of the new private object is the value of the original object that exists immediately prior to the parallel construct for the thread that encounters it. For a `firstprivate` clause on a work-sharing construct, the initial value of the new private object for each thread that executes the work-sharing construct is the value of the original object that exists prior to the point in time that the same thread encounters the work-sharing construct. In addition, for C++ objects, the new private object for each thread is copy constructed from the original object.

The restrictions to the `firstprivate` clause are as follows:

- A variable specified in a `firstprivate` clause must not have an incomplete type or a reference type.
- A variable with a class type that's specified as `firstprivate` must have an accessible, unambiguous copy constructor.
- Variables that are private within a parallel region or that appear in the `reduction` clause of a `parallel` directive can't be specified in a `firstprivate` clause on a work-sharing directive that binds to the parallel construct.

#### 2.7.2.3 `lastprivate`

The `lastprivate` clause provides a superset of the functionality provided by the `private` clause. The syntax of the `lastprivate` clause is as follows:

```
lastprivate(variable-list)
```

Variables specified in the *variable-list* have `private` clause semantics. When a `lastprivate` clause appears on the directive that identifies a work-sharing construct, the value of each `lastprivate` variable from the sequentially last iteration of the associated loop, or the lexically last section directive, is assigned to the variable's original object. Variables that aren't assigned a value by the last iteration of the `for` or `parallel for`, or by the lexically last section of the `sections` or `parallel sections` directive, have indeterminate values after the construct. Unassigned subobjects also have an indeterminate value after the construct.

The restrictions to the `lastprivate` clause are as follows:

- All restrictions for `private` apply.
- A variable with a class type that's specified as `lastprivate` must have an accessible, unambiguous copy assignment operator.
- Variables that are private within a parallel region or that appear in the `reduction` clause of a `parallel` directive can't be specified in a `lastprivate` clause on a work-sharing directive that binds to the parallel construct.

#### 2.7.2.4 shared

This clause shares variables that appear in the *variable-list* among all the threads in a team. All threads within a team access the same storage area for `shared` variables.

The syntax of the `shared` clause is as follows:

```
shared(variable-list)
```

#### 2.7.2.5 default

The `default` clause allows the user to affect the data-sharing attributes of variables. The syntax of the `default` clause is as follows:

```
default(shared | none)
```

Specifying `default(shared)` is equivalent to explicitly listing each currently visible variable in a `shared` clause, unless it's `threadprivate` or `const`-qualified. In the absence of an explicit `default` clause, the default behavior is the same as if `default(shared)` were specified.

Specifying `default(none)` requires that at least one of the following must be true for every reference to a variable in the lexical extent of the parallel construct:

- The variable is explicitly listed in a data-sharing attribute clause of a construct that contains the reference.
- The variable is declared within the parallel construct.
- The variable is `threadprivate`.
- The variable has a `const`-qualified type.
- The variable is the loop control variable for a `for` loop that immediately follows a `for` or `parallel for` directive, and the variable reference appears inside the loop.

Specifying a variable on a `firstprivate`, `lastprivate`, or `reduction` clause of an enclosed directive causes an implicit reference to the variable in the enclosing context. Such implicit references are also subject to the requirements listed above.

Only a single `default` clause may be specified on a `parallel` directive.

A variable's default data-sharing attribute can be overridden by using the `private`, `firstprivate`, `lastprivate`, `reduction`, and `shared` clauses, as demonstrated by the following example:

```
#pragma omp parallel for default(shared) firstprivate(i)\
    private(x) private(r) lastprivate(i)
```

### 2.7.2.6 reduction

This clause performs a reduction on the scalar variables that appear in *variable-list*, with the operator *op*. The syntax of the `reduction` clause is as follows:

```
reduction( op : variable-list )
```

A reduction is typically specified for a statement with one of the following forms:

- $x = x \text{ op } \textit{expr}$
- $x \textit{ binop} = \textit{expr}$
- $x = \textit{expr op } x$  (except for subtraction)
- $x ++$
- $++ x$
- $x --$
- $-- x$

where:

*x*

One of the reduction variables specified in the list.

*variable-list*

A comma-separated list of scalar reduction variables.

*expr*

An expression with scalar type that doesn't reference *x*.

*op*

Not an overloaded operator but one of `+`, `*`, `-`, `&`, `^`, `|`, `&&`, or `||`.

*binop*

Not an overloaded operator but one of `+`, `*`, `-`, `&`, `^`, or `|`.

The following is an example of the `reduction` clause:

```
#pragma omp parallel for reduction(+: a, y) reduction(||: am)
for (i=0; i<n; i++) {
    a += b[i];
    y = sum(y, c[i]);
    am = am || b[i] == c[i];
}
```

As shown in the example, an operator may be hidden inside a function call. The user should be careful that the operator specified in the `reduction` clause matches the reduction operation.

Although the right operand of the `||` operator has no side effects in this example, they're permitted, but should be used with care. In this context, a side effect that's guaranteed not to occur during sequential execution of the loop may occur during parallel execution. This difference can occur because the order of execution of the iterations is

indeterminate.

The operator is used to determine the initial value of any private variables used by the compiler for the reduction and to determine the finalization operator. Specifying the operator explicitly allows the reduction statement to be outside the lexical extent of the construct. Any number of `reduction` clauses may be specified on the directive, but a variable may appear in at most one `reduction` clause for that directive.

A private copy of each variable in *variable-list* is created, one for each thread, as if the `private` clause had been used. The private copy is initialized according to the operator (see the following table).

At the end of the region for which the `reduction` clause was specified, the original object is updated to reflect the result of combining its original value with the final value of each of the private copies using the operator specified. The reduction operators are all associative (except for subtraction), and the compiler may freely reassociate the computation of the final value. (The partial results of a subtraction reduction are added to form the final value.)

The value of the original object becomes indeterminate when the first thread reaches the containing clause and remains so until the reduction computation is complete. Normally, the computation will be complete at the end of the construct; however, if the `reduction` clause is used on a construct to which `nowait` is also applied, the value of the original object remains indeterminate until a barrier synchronization has been performed to ensure that all threads have completed the `reduction` clause.

The following table lists the operators that are valid and their canonical initialization values. The actual initialization value will be consistent with the data type of the reduction variable.

OPERATOR	INITIALIZATION
<code>+</code>	0
<code>*</code>	1
<code>-</code>	0
<code>&amp;</code>	<code>~0</code>
<code> </code>	0
<code>^</code>	0
<code>&amp;&amp;</code>	1
<code>  </code>	0

The restrictions to the `reduction` clause are as follows:

- The type of the variables in the `reduction` clause must be valid for the reduction operator except that pointer types and reference types are never permitted.
- A variable that's specified in the `reduction` clause must not be `const`-qualified.
- Variables that are private within a parallel region or that appear in the `reduction` clause of a `parallel` directive can't be specified in a `reduction` clause on a work-sharing directive that binds to the parallel construct.

```

#pragma omp parallel private(y)
{ /* ERROR - private variable y cannot be specified
   in a reduction clause */
  #pragma omp for reduction(+: y)
  for (i=0; i<n; i++)
    y += b[i];
}

/* ERROR - variable x cannot be specified in both
   a shared and a reduction clause */
#pragma omp parallel for shared(x) reduction(+: x)

```

### 2.7.2.7 copyin

The `copyin` clause provides a mechanism to assign the same value to `threadprivate` variables for each thread in the team executing the parallel region. For each variable specified in a `copyin` clause, the value of the variable in the master thread of the team is copied, as if by assignment, to the thread-private copies at the beginning of the parallel region. The syntax of the `copyin` clause is as follows:

```

copyin(
variable-list
)

```

The restrictions to the `copyin` clause are as follows:

- A variable that's specified in the `copyin` clause must have an accessible, unambiguous copy assignment operator.
- A variable that's specified in the `copyin` clause must be a `threadprivate` variable.

### 2.7.2.8 copyprivate

The `copyprivate` clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members. It's an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level). The `copyprivate` clause can only appear on the `single` directive.

The syntax of the `copyprivate` clause is as follows:

```

copyprivate(
variable-list
)

```

The effect of the `copyprivate` clause on the variables in its `variable-list` occurs after the execution of the structured block associated with the `single` construct, and before any of the threads in the team have left the barrier at the end of the construct. Then, in all other threads in the team, for each variable in the `variable-list`, that variable becomes defined (as if by assignment) with the value of the corresponding variable in the thread that executed the construct's structured block.

Restrictions to the `copyprivate` clause are as follows:

- A variable that's specified in the `copyprivate` clause must not appear in a `private` or `firstprivate` clause for the same `single` directive.
- If a `single` directive with a `copyprivate` clause is encountered in the dynamic extent of a parallel region, all variables specified in the `copyprivate` clause must be private in the enclosing context.

- A variable that's specified in the `copyprivate` clause must have an accessible unambiguous copy assignment operator.

## 2.8 Directive binding

Dynamic binding of directives must adhere to the following rules:

- The `for`, `sections`, `single`, `master`, and `barrier` directives bind to the dynamically enclosing `parallel`, if one exists, regardless of the value of any `if` clause that may be present on that directive. If no parallel region is currently being executed, the directives are executed by a team composed of only the master thread.
- The `ordered` directive binds to the dynamically enclosing `for`.
- The `atomic` directive enforces exclusive access with respect to `atomic` directives in all threads, not just the current team.
- The `critical` directive enforces exclusive access with respect to `critical` directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest dynamically enclosing `parallel`.

## 2.9 Directive nesting

Dynamic nesting of directives must adhere to the following rules:

- A `parallel` directive dynamically inside another `parallel` logically establishes a new team, which is composed of only the current thread, unless nested parallelism is enabled.
- `for`, `sections`, and `single` directives that bind to the same `parallel` aren't allowed to be nested inside each other.
- `critical` directives with the same name aren't allowed to be nested inside each other. Note that this restriction isn't sufficient to prevent deadlock.
- `for`, `sections`, and `single` directives aren't permitted in the dynamic extent of `critical`, `ordered`, and `master` regions if the directives bind to the same `parallel` as the regions.
- `barrier` directives aren't permitted in the dynamic extent of `for`, `ordered`, `sections`, `single`, `master`, and `critical` regions if the directives bind to the same `parallel` as the regions.
- `master` directives aren't permitted in the dynamic extent of `for`, `sections`, and `single` directives if the `master` directives bind to the same `parallel` as the work-sharing directives.
- `ordered` directives aren't allowed in the dynamic extent of `critical` regions if the directives bind to the same `parallel` as the regions.
- Any directive that's permitted when executed dynamically inside a parallel region is also permitted when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed by a team composed of only the master thread.

# 3. Run-time library functions

5/14/2019 • 11 minutes to read • [Edit Online](#)

This section describes the OpenMP C and C++ run-time library functions. The header `<omp.h>` declares two types, several functions that can be used to control and query the parallel execution environment, and lock functions that can be used to synchronize access to data.

The type `omp_lock_t` is an object type capable of representing that a lock is available, or that a thread owns a lock. These locks are referred to as *simple locks*.

The type `omp_nest_lock_t` is an object type capable of representing either that a lock is available, or both the identity of the thread that owns the lock and a *nesting count* (described below). These locks are referred to as *nestable locks*.

The library functions are external functions with "C" linkage.

The descriptions in this chapter are divided into the following topics:

- [Execution environment functions](#)
- [Lock functions](#)
- [Timing routines](#)

## 3.1 Execution environment functions

The functions described in this section affect and monitor threads, processors, and the parallel environment:

- [omp\\_set\\_num\\_threads](#)
- [omp\\_get\\_num\\_threads](#)
- [omp\\_get\\_max\\_threads](#)
- [omp\\_get\\_thread\\_num](#)
- [omp\\_get\\_num\\_procs](#)
- [omp\\_in\\_parallel](#)
- [omp\\_set\\_dynamic](#)
- [omp\\_get\\_dynamic](#)
- [omp\\_set\\_nested](#)
- [omp\\_get\\_nested](#)

### 3.1.1 `omp_set_num_threads` function

The `omp_set_num_threads` function sets the default number of threads to use for later parallel regions that don't specify a `num_threads` clause. The format is as follows:

```
#include <omp.h>
void omp_set_num_threads(int num_threads);
```

The value of the parameter `num_threads` must be a positive integer. Its effect depends upon whether dynamic adjustment of the number of threads is enabled. For a comprehensive set of rules about the interaction between the `omp_set_num_threads` function and dynamic adjustment of threads, see [section 2.3](#).

This function has the effects described above when called from a portion of the program where the

`omp_in_parallel` function returns zero. If it's called from a portion of the program where the `omp_in_parallel`

function returns a nonzero value, the behavior of this function is undefined.

This call has precedence over the `OMP_NUM_THREADS` environment variable. The default value for the number of threads, which may be established by calling `omp_set_num_threads` or by setting the `OMP_NUM_THREADS` environment variable, can be explicitly overridden on a single `parallel` directive by specifying the `num_threads` clause.

For more information, see [omp\\_set\\_dynamic](#).

#### Cross-references

- [omp\\_set\\_dynamic](#) function
- [omp\\_get\\_dynamic](#) function
- `OMP_NUM_THREADS` environment variable
- `num_threads` clause

### 3.1.2 `omp_get_num_threads` function

The `omp_get_num_threads` function returns the number of threads currently in the team executing the parallel region from which it's called. The format is as follows:

```
#include <omp.h>
int omp_get_num_threads(void);
```

The `num_threads` clause, the `omp_set_num_threads` function, and the `OMP_NUM_THREADS` environment variable control the number of threads in a team.

If the number of threads hasn't been explicitly set by the user, the default is implementation-defined. This function binds to the closest enclosing `parallel` directive. If called from a serial portion of a program, or from a nested parallel region that's serialized, this function returns 1.

For more information, see [omp\\_set\\_dynamic](#).

#### Cross-references

- `OMP_NUM_THREADS`
- `num_threads`
- `parallel`

### 3.1.3 `omp_get_max_threads` function

The `omp_get_max_threads` function returns an integer that's guaranteed to be at least as large as the number of threads that would be used to form a team if a parallel region without a `num_threads` clause were to be seen at that point in the code. The format is as follows:

```
#include <omp.h>
int omp_get_max_threads(void);
```

The following expresses a lower bound on the value of `omp_get_max_threads`:

```
threads-used-for-next-team
<= omp_get_max_threads
```

Note that if another parallel region uses the `num_threads` clause to request a specific number of threads, the guarantee on the lower bound of the result of `omp_get_max_threads` no longer holds.

The `omp_get_max_threads` function's return value can be used to dynamically allocate sufficient storage for all threads in the team formed at the next parallel region.

#### Cross-references

- [omp\\_get\\_num\\_threads](#)
- [omp\\_set\\_num\\_threads](#)
- [omp\\_set\\_dynamic](#)
- [num\\_threads](#)

### 3.1.4 omp\_get\_thread\_num function

The `omp_get_thread_num` function returns the thread number, within its team, of the thread executing the function. The thread number lies between 0 and `omp_get_num_threads() - 1`, inclusive. The master thread of the team is thread 0.

The format is as follows:

```
#include <omp.h>
int omp_get_thread_num(void);
```

If called from a serial region, `omp_get_thread_num` returns 0. If called from within a nested parallel region that's serialized, this function returns 0.

#### Cross-references

- [omp\\_get\\_num\\_threads](#) function

### 3.1.5 omp\_get\_num\_procs function

The `omp_get_num_procs` function returns the number of processors that are available to the program at the time the function is called. The format is as follows:

```
#include <omp.h>
int omp_get_num_procs(void);
```

### 3.1.6 omp\_in\_parallel function

The `omp_in_parallel` function returns a nonzero value if it's called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0. The format is as follows:

```
#include <omp.h>
int omp_in_parallel(void);
```

This function returns a nonzero value when called from within a region executing in parallel, including nested regions that are serialized.

### 3.1.7 omp\_set\_dynamic function

The `omp_set_dynamic` function enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. The format is as follows:

```
#include <omp.h>
void omp_set_dynamic(int dynamic_threads);
```

If *dynamic\_threads* evaluates to a nonzero value, the number of threads that are used for executing upcoming parallel regions may be adjusted automatically by the run-time environment to best use system resources. As a consequence, the number of threads specified by the user is the maximum thread count. The number of threads in the team executing a parallel region stays fixed for the duration of that parallel region and is reported by the `omp_get_num_threads` function.

If *dynamic\_threads* evaluates to 0, dynamic adjustment is disabled.

This function has the effects described above when called from a portion of the program where the `omp_in_parallel` function returns zero. If it's called from a portion of the program where the `omp_in_parallel` function returns a nonzero value, the behavior of this function is undefined.

A call to `omp_set_dynamic` has precedence over the `OMP_DYNAMIC` environment variable.

The default for the dynamic adjustment of threads is implementation-defined. As a result, user codes that depend on a specific number of threads for correct execution should explicitly disable dynamic threads. Implementations aren't required to provide the ability to dynamically adjust the number of threads, but they're required to provide the interface to support portability across all platforms.

#### Microsoft specific

The current support of `omp_get_dynamic` and `omp_set_dynamic` is as follows:

The input parameter to `omp_set_dynamic` does not affect the threading policy and does not change the number of threads. `omp_get_num_threads` always returns either the user-defined number, if that is set, or the default thread number. In the current Microsoft implementation, `omp_set_dynamic(0)` turns off dynamic threading so that the existing set of threads can be reused for the following parallel region. `omp_set_dynamic(1)` turns on dynamic threading by discarding the existing set of threads and creating a new set for the upcoming parallel region. The number of threads in the new set is the same as the old set, and is based on the return value of `omp_get_num_threads`. Therefore, for best performance, use `omp_set_dynamic(0)` to reuse the existing threads.

#### Cross-references

- [omp\\_get\\_num\\_threads](#)
- [OMP\\_DYNAMIC](#)
- [omp\\_in\\_parallel](#)

### 3.1.8 `omp_get_dynamic` function

The `omp_get_dynamic` function returns a nonzero value if dynamic adjustment of threads is enabled, and returns 0 otherwise. The format is as follows:

```
#include <omp.h>
int omp_get_dynamic(void);
```

If the implementation doesn't implement dynamic adjustment of the number of threads, this function always returns 0. For more information, see [omp\\_set\\_dynamic](#).

#### Cross-references

- For a description of dynamic thread adjustment, see [omp\\_set\\_dynamic](#).

### 3.1.9 `omp_set_nested` function

The `omp_set_nested` function enables or disables nested parallelism. The format is as follows:

```
#include <omp.h>
void omp_set_nested(int nested);
```

If *nested* evaluates to 0, nested parallelism is disabled, which is the default, and nested parallel regions are serialized and executed by the current thread. Otherwise, nested parallelism is enabled, and parallel regions that are nested may deploy additional threads to form nested teams.

This function has the effects described above when called from a portion of the program where the `omp_in_parallel` function returns zero. If it's called from a portion of the program where the `omp_in_parallel` function returns a nonzero value, the behavior of this function is undefined.

This call has precedence over the `OMP_NESTED` environment variable.

When nested parallelism is enabled, the number of threads used to execute nested parallel regions is implementation-defined. As a result, OpenMP-compliant implementations are allowed to serialize nested parallel regions even when nested parallelism is enabled.

#### Cross-references

- [OMP\\_NESTED](#)
- [omp\\_in\\_parallel](#)

### 3.1.10 `omp_get_nested` function

The `omp_get_nested` function returns a nonzero value if nested parallelism is enabled and 0 if it's disabled. For more information on nested parallelism, see [omp\\_set\\_nested](#). The format is as follows:

```
#include <omp.h>
int omp_get_nested(void);
```

If an implementation doesn't implement nested parallelism, this function always returns 0.

## 3.2 Lock functions

The functions described in this section manipulate locks used for synchronization.

For the following functions, the lock variable must have type `omp_lock_t`. This variable must only be accessed through these functions. All lock functions require an argument that has a pointer to `omp_lock_t` type.

- The [omp\\_init\\_lock](#) function initializes a simple lock.
- The [omp\\_destroy\\_lock](#) function removes a simple lock.
- The [omp\\_set\\_lock](#) function waits until a simple lock is available.
- The [omp\\_unset\\_lock](#) function releases a simple lock.
- The [omp\\_test\\_lock](#) function tests a simple lock.

For the following functions, the lock variable must have type `omp_nest_lock_t`. This variable must only be accessed through these functions. All nestable lock functions require an argument that has a pointer to `omp_nest_lock_t` type.

- The [omp\\_init\\_nest\\_lock](#) function initializes a nestable lock.
- The [omp\\_destroy\\_nest\\_lock](#) function removes a nestable lock.
- The [omp\\_set\\_nest\\_lock](#) function waits until a nestable lock is available.
- The [omp\\_unset\\_nest\\_lock](#) function releases a nestable lock.
- The [omp\\_test\\_nest\\_lock](#) function tests a nestable lock.

The OpenMP lock functions access the lock variable in such a way that they always read and update the most current value of the lock variable. Therefore, it isn't necessary for an OpenMP program to include explicit `flush` directives to make sure that the lock variable's value is consistent among different threads. (There may be a need for `flush` directives to make the values of other variables consistent.)

### 3.2.1 `omp_init_lock` and `omp_init_nest_lock` functions

These functions provide the only means of initializing a lock. Each function initializes the lock associated with the parameter *lock* for use in upcoming calls. The format is as follows:

```
#include <omp.h>
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

The initial state is unlocked (that is, no thread owns the lock). For a nestable lock, the initial nesting count is zero.

It's noncompliant to call either of these routines with a lock variable that has already been initialized.

### 3.2.2 `omp_destroy_lock` and `omp_destroy_nest_lock` functions

These functions make sure that the pointed to lock variable *lock* is uninitialized. The format is as follows:

```
#include <omp.h>
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

It's noncompliant to call either of these routines with a lock variable that's uninitialized or unlocked.

### 3.2.3 `omp_set_lock` and `omp_set_nest_lock` functions

Each of these functions blocks the thread executing the function until the specified lock is available and then sets the lock. A simple lock is available if it's unlocked. A nestable lock is available if it's unlocked or if it's already owned by the thread executing the function. The format is as follows:

```
#include <omp.h>
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

For a simple lock, the argument to the `omp_set_lock` function must point to an initialized lock variable. Ownership of the lock is granted to the thread executing the function.

For a nestable lock, the argument to the `omp_set_nest_lock` function must point to an initialized lock variable. The nesting count is incremented, and the thread is granted, or keeps, ownership of the lock.

### 3.2.4 `omp_unset_lock` and `omp_unset_nest_lock` functions

These functions provide the means of releasing ownership of a lock. The format is as follows:

```
#include <omp.h>
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

The argument to each of these functions must point to an initialized lock variable owned by the thread executing the function. The behavior is undefined if the thread doesn't own that lock.

For a simple lock, the `omp_unset_lock` function releases the thread executing the function from ownership of the lock.

For a nestable lock, the `omp_unset_nest_lock` function decrements the nesting count, and releases the thread executing the function from ownership of the lock if the resulting count is zero.

### 3.2.5 `omp_test_lock` and `omp_test_nest_lock` functions

These functions attempt to set a lock but don't block execution of the thread. The format is as follows:

```
#include <omp.h>
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

The argument must point to an initialized lock variable. These functions attempt to set a lock in the same manner as `omp_set_lock` and `omp_set_nest_lock`, except that they don't block execution of the thread.

For a simple lock, the `omp_test_lock` function returns a nonzero value if the lock is successfully set; otherwise, it returns zero.

For a nestable lock, the `omp_test_nest_lock` function returns the new nesting count if the lock is successfully set; otherwise, it returns zero.

## 3.3 Timing routines

The functions described in this section support a portable wall-clock timer:

- The `omp_get_wtime` function returns elapsed wall-clock time.
- The `omp_get_wtick` function returns seconds between successive clock ticks.

### 3.3.1 `omp_get_wtime` function

The `omp_get_wtime` function returns a double-precision floating point value equal to the elapsed wall clock time in seconds since some "time in the past". The actual "time in the past" is arbitrary, but it's guaranteed not to change during the execution of the application program. The format is as follows:

```
#include <omp.h>
double omp_get_wtime(void);
```

It's anticipated that the function will be used to measure elapsed times as shown in the following example:

```
double start;
double end;
start = omp_get_wtime();
... work to be timed ...
end = omp_get_wtime();
printf_s("Work took %f sec. time.\n", end-start);
```

The times returned are "per-thread times" by which is meant they aren't required to be globally consistent across all the threads participating in an application.

### 3.3.2 `omp_get_wtick` function

The `omp_get_wtick` function returns a double-precision floating point value equal to the number of seconds between successive clock ticks. The format is as follows:

```
#include <omp.h>
double omp_get_wtick(void);
```

# 4. Environment variables

2/11/2019 • 2 minutes to read • [Edit Online](#)

This chapter describes the OpenMP C and C++ API environment variables (or similar platform-specific mechanisms) that control the execution of parallel code. The names of environment variables must be uppercase. The values assigned to them are case insensitive and may have leading and trailing white space. Modifications to the values after the program has started are ignored.

The environment variables are as follows:

- `OMP_SCHEDULE` sets the run-time schedule type and chunk size.
- `OMP_NUM_THREADS` sets the number of threads to use during execution.
- `OMP_DYNAMIC` enables or disables the dynamic adjustment of the number of threads.
- `OMP_NESTED` enables or disables nested parallelism.

The examples in this chapter only demonstrate how these variables might be set in Unix C shell (csh) environments. In the Korn shell and DOS environments, the actions are similar:

csh:

```
setenv OMP_SCHEDULE "dynamic"
```

ksh:

```
export OMP_SCHEDULE="dynamic"
```

DOS:

```
set OMP_SCHEDULE="dynamic"
```

## 4.1 OMP\_SCHEDULE

`OMP_SCHEDULE` applies only to `for` and `parallel for` directives that have the schedule type `runtime`. The schedule type and chunk size for all such loops can be set at run time. Set this environment variable to any recognized schedule type and to an optional *chunk\_size*.

For `for` and `parallel for` directives that have a schedule type other than `runtime`, `OMP_SCHEDULE` is ignored. The default value for this environment variable is implementation-defined. If the optional *chunk\_size* is set, the value must be positive. If *chunk\_size* isn't set, a value of 1 is assumed, except when the schedule is `static`. For a `static` schedule, the default chunk size is set to the loop iteration space divided by the number of threads applied to the loop.

Example:

```
setenv OMP_SCHEDULE "guided,4"  
setenv OMP_SCHEDULE "dynamic"
```

### Cross-references

- [for](#) directive
- [parallel for](#) directive

## 4.2 OMP\_NUM\_THREADS

The `OMP_NUM_THREADS` environment variable sets the default number of threads to use during execution.

`OMP_NUM_THREADS` is ignored if that number is explicitly changed by calling the `omp_set_num_threads` library routine. It's also ignored if there's an explicit `num_threads` clause on a `parallel` directive.

The value of the `OMP_NUM_THREADS` environment variable must be a positive integer. Its effect depends upon whether dynamic adjustment of the number of threads is enabled. For a comprehensive set of rules about the interaction between the `OMP_NUM_THREADS` environment variable and dynamic adjustment of threads, see [section 2.3](#).

The number of threads to use is implementation-defined if:

- the `OMP_NUM_THREADS` environment variable isn't specified,
- the value specified isn't a positive integer, or
- the value is greater than the maximum number of threads that the system can support.

Example:

```
setenv OMP_NUM_THREADS 16
```

### Cross-references

- [num\\_threads](#) clause
- [omp\\_set\\_num\\_threads](#) function
- [omp\\_set\\_dynamic](#) function

## 4.3 OMP\_DYNAMIC

The `OMP_DYNAMIC` environment variable enables or disables dynamic adjustment of the number of threads available for the execution of parallel regions. `OMP_DYNAMIC` is ignored when dynamic adjustment is explicitly enabled or disabled by calling the `omp_set_dynamic` library routine. Its value must be `TRUE` or `FALSE`.

If `OMP_DYNAMIC` is set to `TRUE`, the number of threads that are used for executing parallel regions may be adjusted by the runtime environment to best use system resources. If `OMP_DYNAMIC` is set to `FALSE`, dynamic adjustment is disabled. The default condition is implementation-defined.

Example:

```
setenv OMP_DYNAMIC TRUE
```

### Cross-references

- [Parallel regions](#)
- [omp\\_set\\_dynamic](#) function

## 4.4 OMP\_NESTED

The `OMP_NESTED` environment variable enables or disables nested parallelism unless nested parallelism is enabled or disabled by calling the `omp_set_nested` library routine. If `OMP_NESTED` is set to `TRUE`, nested parallelism is enabled. If `OMP_NESTED` is set to `FALSE`, nested parallelism is disabled. The default value is `FALSE`.

Example:

```
setenv OMP_NESTED TRUE
```

### Cross-reference

- [omp\\_set\\_nested](#) function

# A. Examples

1/28/2019 • 17 minutes to read • [Edit Online](#)

The following are examples of the constructs defined in this document. A statement following a directive is compound only when necessary, and a non-compound statement is indented from a directive preceding it.

## A.1 A simple loop in parallel

The following example demonstrates how to parallelize a loop using the `parallel for` directive. The loop iteration variable is private by default, so it isn't necessary to specify it explicitly in a private clause.

```
#pragma omp parallel for
  for (i=1; i<n; i++)
    b[i] = (a[i] + a[i-1]) / 2.0;
```

## A.2 Conditional compilation

The following examples illustrate the use of conditional compilation using the OpenMP macro `_OPENMP`. With OpenMP compilation, the `_OPENMP` macro becomes defined.

```
# ifdef _OPENMP
  printf_s("Compiled by an OpenMP-compliant implementation.\n");
# endif
```

The defined preprocessor operator allows more than one macro to be tested in a single directive.

```
# if defined(_OPENMP) && defined(VERBOSE)
  printf_s("Compiled by an OpenMP-compliant implementation.\n");
# endif
```

## A.3 Parallel regions

The `parallel` directive can be used in coarse-grain parallel programs. In the following example, each thread in the parallel region decides what part of the global array `x` to work on, based on the thread number:

```
#pragma omp parallel shared(x, npoints) private(iam, np, ipoints)
{
  iam = omp_get_thread_num();
  np = omp_get_num_threads();
  ipoints = npoints / np;
  subdomain(x, iam, ipoints);
}
```

## A.4 The `nowait` clause

If there are many independent loops within a parallel region, you can use the `nowait` clause to avoid the implied barrier at the end of the `for` directive, as follows:

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
    #pragma omp for nowait
    for (i=0; i<m; i++)
        y[i] = sqrt(z[i]);
}
```

## A.5 The critical directive

The following example includes several `critical` directives. The example illustrates a queuing model in which a task is dequeued and worked on. To guard against many threads dequeuing the same task, the dequeuing operation must be in a `critical` section. Because the two queues in this example are independent, they're protected by `critical` directives with different names, *xaxis* and *yaxis*.

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
    #pragma omp critical ( xaxis )
        x_next = dequeue(x);
    work(x_next);
    #pragma omp critical ( yaxis )
        y_next = dequeue(y);
    work(y_next);
}
```

## A.6 The lastprivate clause

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables as arguments to a `lastprivate` clause so that the values of the variables are the same as when the loop is executed sequentially.

```
#pragma omp parallel
{
    #pragma omp for lastprivate(i)
    for (i=0; i<n-1; i++)
        a[i] = b[i] + b[i+1];
}
a[i]=b[i];
```

In the preceding example, the value of `i` at the end of the parallel region will equal `n-1`, as in the sequential case.

## A.7 The reduction clause

The following example demonstrates the `reduction` clause:

```
#pragma omp parallel for private(i) shared(x, y, n) \
    reduction(+: a, b)
for (i=0; i<n; i++) {
    a = a + x[i];
    b = b + y[i];
}
```

## A.8 Parallel sections

In the following example (for [section 2.4.2](#)), functions *xaxis*, *yaxis*, and *zaxis* can be executed concurrently. The first `section` directive is optional. All `section` directives need to appear in the lexical extent of the `parallel sections` construct.

```
#pragma omp parallel sections
{
    #pragma omp section
        xaxis();
    #pragma omp section
        yaxis();
    #pragma omp section
        zaxis();
}
```

## A.9 Single directives

The following example demonstrates the `single` directive. In the example, only one thread (usually the first thread that encounters the `single` directive) prints the progress message. The user must not make any assumptions as to which thread will execute the `single` section. All other threads will skip the `single` section and stop at the barrier at the end of the `single` construct. If other threads can proceed without waiting for the thread executing the `single` section, a `nowait` clause can be specified on the `single` directive.

```
#pragma omp parallel
{
    #pragma omp single
        printf_s("Beginning work1.\n");
    work1();
    #pragma omp single
        printf_s("Finishing work1.\n");
    #pragma omp single nowait
        printf_s("Finished work1 and beginning work2.\n");
    work2();
}
```

## A.10 Sequential ordering

[Ordered sections](#) are useful for sequentially ordering the output from work that's done in parallel. The following program prints out the indexes in sequential order:

```
#pragma omp for ordered schedule(dynamic)
for (i=lb; i<ub; i+=st)
    work(i);
void work(int k)
{
    #pragma omp ordered
        printf_s(" %d", k);
}
```

## A.11 A fixed number of threads

Some programs rely on a fixed, prespecified number of threads to execute correctly. Because the default setting for the dynamic adjustment of the number of threads is implementation-defined, such programs can choose to turn off the dynamic threads capability and set the number of threads explicitly to keep portability. The following example shows how to do this using `omp_set_dynamic`, and `omp_set_num_threads`:

```

omp_set_dynamic(0);
omp_set_num_threads(16);
#pragma omp parallel shared(x, npoints) private(iam, ipoints)
{
    if (omp_get_num_threads() != 16)
        abort();
    iam = omp_get_thread_num();
    ipoints = npoints/16;
    do_by_16(x, iam, ipoints);
}

```

In this example, the program executes correctly only if it's executed by 16 threads. If the implementation isn't capable of supporting 16 threads, the behavior of this example is implementation-defined.

The number of threads executing a parallel region stays constant during a parallel region, regardless of the dynamic threads setting. The dynamic threads mechanism determines the number of threads to use at the start of the parallel region and keeps it constant for the duration of the region.

## A.12 The atomic directive

The following example avoids race conditions (simultaneous updates of an element of  $x$  by many threads) by using the `atomic` directive:

```

#pragma omp parallel for shared(x, y, index, n)
for (i=0; i<n; i++)
{
    #pragma omp atomic
    x[index[i]] += work1(i);
    y[i] += work2(i);
}

```

The advantage of using the `atomic` directive in this example is that it allows updates of two different elements of  $x$  to occur in parallel. If a `critical` directive is used instead, then all updates to elements of  $x$  are executed serially (though not in any guaranteed order).

The `atomic` directive applies only to the C or C++ statement immediately following it. As a result, elements of  $y$  aren't updated atomically in this example.

## A.13 A flush directive with a list

The following example uses the `flush` directive for point-to-point synchronization of specific objects between pairs of threads:

```

int   sync[NUMBER_OF_THREADS];
float work[NUMBER_OF_THREADS];
#pragma omp parallel private(iam,neighbor) shared(work,sync)
{
    iam = omp_get_thread_num();
    sync[iam] = 0;
    #pragma omp barrier

    // Do computation into my portion of work array
    work[iam] = ...;

    // Announce that I am done with my work
    // The first flush ensures that my work is
    // made visible before sync.
    // The second flush ensures that sync is made visible.
    #pragma omp flush(work)
    sync[iam] = 1;
    #pragma omp flush(sync)

    // Wait for neighbor
    neighbor = (iam>0 ? iam : omp_get_num_threads()) - 1;
    while (sync[neighbor]==0)
    {
        #pragma omp flush(sync)
    }

    // Read neighbor's values of work array
    ... = work[neighbor];
}

```

## A.14 A flush directive without a list

The following example (for [section 2.6.5](#)) distinguishes the shared objects affected by a `flush` directive with no list from the shared objects that aren't affected:

```

// omp_flush_without_list.c
#include <omp.h>

int x, *p = &x;

void f1(int *q)
{
    *q = 1;
    #pragma omp flush
    // x, p, and *q are flushed
    // because they are shared and accessible
    // q is not flushed because it is not shared.
}

void f2(int *q)
{
    #pragma omp barrier
    *q = 2;

    #pragma omp barrier
    // a barrier implies a flush
    // x, p, and *q are flushed
    // because they are shared and accessible
    // q is not flushed because it is not shared.
}

int g(int n)
{
    int i = 1, j, sum = 0;
    *p = 1;

    #pragma omp parallel reduction(+: sum) num_threads(10)
    {
        f1(&j);
        // i, n and sum were not flushed
        // because they were not accessible in f1
        // j was flushed because it was accessible
        sum += j;
        f2(&j);
        // i, n, and sum were not flushed
        // because they were not accessible in f2
        // j was flushed because it was accessible
        sum += i + j + *p + n;
    }
    return sum;
}

int main()
{
}

```

## A.15 The number of threads used

Consider the following incorrect example (for [section 3.1.2](#)):

```

np = omp_get_num_threads(); // misplaced
#pragma omp parallel for schedule(static)
    for (i=0; i<np; i++)
        work(i);

```

The `omp_get_num_threads()` call returns 1 in the serial section of the code, so `np` will always be equal to 1 in the preceding example. To determine the number of threads that will be deployed for the parallel region, the call should be inside the parallel region.

The following example shows how to rewrite this program without including a query for the number of threads:

```
#pragma omp parallel private(i)
{
    i = omp_get_thread_num();
    work(i);
}
```

## A.16 Locks

In the following example (for [section 3.2](#)), the argument to the lock functions should have type `omp_lock_t`, and that there's no need to flush it. The lock functions cause the threads to be idle while waiting for entry to the first critical section, but to do other work while waiting for entry to the second. The `omp_set_lock` function blocks, but the `omp_test_lock` function doesn't, allowing the work in `skip()` to be done.

```
// omp_using_locks.c
// compile with: /openmp /c
#include <stdio.h>
#include <omp.h>

void work(int);
void skip(int);

int main() {
    omp_lock_t lck;
    int id;

    omp_init_lock(&lck);
    #pragma omp parallel shared(lck) private(id)
    {
        id = omp_get_thread_num();

        omp_set_lock(&lck);
        printf_s("My thread id is %d.\n", id);

        // only one thread at a time can execute this printf
        omp_unset_lock(&lck);

        while (! omp_test_lock(&lck)) {
            skip(id); // we do not yet have the lock,
                    // so we must do something else
        }
        work(id); // we now have the lock
                // and can do the work
        omp_unset_lock(&lck);
    }
    omp_destroy_lock(&lck);
}
```

## A.17 Nestable locks

The following example (for [section 3.2](#)) demonstrates how a nestable lock can be used to synchronize updates both to a whole structure and to one of its members.

```

#include <omp.h>
typedef struct {int a,b; omp_nest_lock_t lck;} pair;

void incr_a(pair *p, int a)
{
    // Called only from incr_pair, no need to lock.
    p->a += a;
}

void incr_b(pair *p, int b)
{
    // Called both from incr_pair and elsewhere,
    // so need a nestable lock.

    omp_set_nest_lock(&p->lck);
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}

void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}

void f(pair *p)
{
    extern int work1(), work2(), work3();
    #pragma omp parallel sections
    {
        #pragma omp section
            incr_pair(p, work1(), work2());
        #pragma omp section
            incr_b(p, work3());
    }
}

```

## A.18 Nested for directives

The following example of `for` [directive nesting](#) is compliant because the inner and outer `for` directives bind to different parallel regions:

```

#pragma omp parallel default(shared)
{
    #pragma omp for
    for (i=0; i<n; i++)
    {
        #pragma omp parallel shared(i, n)
        {
            #pragma omp for
            for (j=0; j<n; j++)
                work(i, j);
        }
    }
}

```

A following variation of the preceding example is also compliant:

```

#pragma omp parallel default(shared)
{
    #pragma omp for
    for (i=0; i<n; i++)
        work1(i, n);
}

void work1(int i, int n)
{
    int j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for
        for (j=0; j<n; j++)
            work2(i, j);
    }
    return;
}

```

## A.19 Examples showing incorrect nesting of work-sharing directives

The examples in this section illustrate the [directive nesting](#) rules.

The following example is noncompliant because the inner and outer `for` directives are nested and bind to the same `parallel` directive:

```

void wrong1(int n)
{
    #pragma omp parallel default(shared)
    {
        int i, j;
        #pragma omp for
        for (i=0; i<n; i++) {
            #pragma omp for
            for (j=0; j<n; j++)
                work(i, j);
        }
    }
}

```

The following dynamically nested version of the preceding example is also noncompliant:

```

void wrong2(int n)
{
    #pragma omp parallel default(shared)
    {
        int i;
        #pragma omp for
        for (i=0; i<n; i++)
            work1(i, n);
    }
}

void work1(int i, int n)
{
    int j;
    #pragma omp for
    for (j=0; j<n; j++)
        work2(i, j);
}

```

The following example is noncompliant because the `for` and `single` directives are nested, and they bind to the same parallel region:

```
void wrong3(int n)
{
    #pragma omp parallel default(shared)
    {
        int i;
        #pragma omp for
        for (i=0; i<n; i++) {
            #pragma omp single
            work(i);
        }
    }
}
```

The following example is noncompliant because a `barrier` directive inside a `for` can result in deadlock:

```
void wrong4(int n)
{
    #pragma omp parallel default(shared)
    {
        int i;
        #pragma omp for
        for (i=0; i<n; i++) {
            work1(i);
            #pragma omp barrier
            work2(i);
        }
    }
}
```

The following example is noncompliant because the `barrier` results in deadlock due to the fact that only one thread at a time can enter the critical section:

```
void wrong5()
{
    #pragma omp parallel
    {
        #pragma omp critical
        {
            work1();
            #pragma omp barrier
            work2();
        }
    }
}
```

The following example is noncompliant because the `barrier` results in deadlock due to the fact that only one thread executes the `single` section:

```

void wrong6()
{
  #pragma omp parallel
  {
    setup();
    #pragma omp single
    {
      work1();
      #pragma omp barrier
      work2();
    }
    finish();
  }
}

```

## A.20 Bind barrier directives

The directive binding rules call for a `barrier` directive to bind to the closest enclosing `parallel` directive. For more information on directive binding, see [section 2.8](#).

In the following example, the call from *main* to *sub2* is compliant because the `barrier` (in *sub3*) binds to the parallel region in *sub2*. The call from *main* to *sub1* is compliant because the `barrier` binds to the parallel region in subroutine *sub2*. The call from *main* to *sub3* is compliant because the `barrier` doesn't bind to any parallel region and is ignored. Also, the `barrier` only synchronizes the team of threads in the enclosing parallel region and not all the threads created in *sub1*.

```

int main()
{
  sub1(2);
  sub2(2);
  sub3(2);
}

void sub1(int n)
{
  int i;
  #pragma omp parallel private(i) shared(n)
  {
    #pragma omp for
    for (i=0; i<n; i++)
      sub2(i);
  }
}

void sub2(int k)
{
  #pragma omp parallel shared(k)
  sub3(k);
}

void sub3(int n)
{
  work(n);
  #pragma omp barrier
  work(n);
}

```

## A.21 Scope variables with the private clause

The values of `i` and `j` in the following example are undefined on exit from the parallel region:

```

int i, j;
i = 1;
j = 2;
#pragma omp parallel private(i) firstprivate(j)
{
    i = 3;
    j = j + 2;
}
printf_s("%d %d\n", i, j);

```

For more information on the `private` clause, see [section 2.7.2.1](#).

## A.22 The default(none) clause

The following example distinguishes the variables that are affected by the `default(none)` clause from the variables that aren't:

```

// openmp_using_clausedefault.c
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int x, y, z[1000];
#pragma omp threadprivate(x)

void fun(int a) {
    const int c = 1;
    int i = 0;

    #pragma omp parallel default(none) private(a) shared(z)
    {
        int j = omp_get_num_thread();
        //O.K. - j is declared within parallel region
        a = z[j]; // O.K. - a is listed in private clause
                // - z is listed in shared clause
        x = c; // O.K. - x is threadprivate
                // - c has const-qualified type
        z[i] = y; // C3052 error - cannot reference i or y here

        #pragma omp for firstprivate(y)
        for (i=0; i<10 ; i++) {
            z[i] = y; // O.K. - i is the loop control variable
                    // - y is listed in firstprivate clause
        }
        z[i] = y; // Error - cannot reference i or y here
    }
}

```

For more information on the `default` clause, see [section 2.7.2.5](#).

## A.23 Examples of the ordered directive

It's possible to have many ordered sections with a `for` specified with the `ordered` clause. The first example is noncompliant because the API specifies the following rule:

"An iteration of a loop with a `for` construct must not execute the same `ordered` directive more than once, and it must not execute more than one `ordered` directive." (See [section 2.6.6](#).)

In this noncompliant example, all iterations execute two ordered sections:

```

#pragma omp for ordered
for (i=0; i<n; i++)
{
    ...
    #pragma omp ordered
    { ... }
    ...
    #pragma omp ordered
    { ... }
    ...
}

```

The following compliant example shows a `for` with more than one ordered section:

```

#pragma omp for ordered
for (i=0; i<n; i++)
{
    ...
    if (i <= 10)
    {
        ...
        #pragma omp ordered
        { ... }
    }
    ...
    (i > 10)
    {
        ...
        #pragma omp ordered
        { ... }
    }
    ...
}

```

## A.24 Example of the private clause

The `private` clause of a parallel region is only in effect for the lexical extent of the region, not for the dynamic extent of the region. Therefore, in the example that follows, any uses of the variable `a` within the `for` loop in the routine `f` refers to a private copy of `a`, while a usage in routine `g` refers to the global `a`.

```

int a;

void f(int n)
{
    a = 0;

    #pragma omp parallel for private(a)
    for (int i=1; i<n; i++)
    {
        a = i;
        g(i, n);
        d(a);    // Private copy of "a"
        ...
    }
    ...
}

void g(int k, int n)
{
    h(k,a); // The global "a", not the private "a" in f
}

```

## A.25 Examples of the copyprivate data attribute clause

**Example 1:** The `copyprivate` clause can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads.

```
float x, y;
#pragma omp threadprivate(x, y)

void init( )
{
    float a;
    float b;

    #pragma omp single copyprivate(a,b,x,y)
    {
        get_values(a,b,x,y);
    }

    use_values(a, b, x, y);
}
```

If routine *init* is called from a serial region, its behavior isn't affected by the presence of the directives. After the call to the *get\_values* routine has been executed by one thread, no thread leaves the construct until the private objects designated by *a*, *b*, *x*, and *y* in all threads have become defined with the values read.

**Example 2:** In contrast to the previous example, suppose the read must be performed by a particular thread, say the master thread. In this case, the `copyprivate` clause can't be used to do the broadcast directly, but it can be used to provide access to a temporary shared object.

```
float read_next( )
{
    float * tmp;
    float return_val;

    #pragma omp single copyprivate(tmp)
    {
        tmp = (float *) malloc(sizeof(float));
    }

    #pragma omp master
    {
        get_float( tmp );
    }

    #pragma omp barrier
    return_val = *tmp;
    #pragma omp barrier

    #pragma omp single
    {
        free(tmp);
    }

    return return_val;
}
```

**Example 3:** Suppose that the number of lock objects required within a parallel region can't easily be determined prior to entering it. The `copyprivate` clause can be used to provide access to shared lock objects that are allocated within that parallel region.

```

#include <omp.h>

omp_lock_t *new_lock()
{
    omp_lock_t *lock_ptr;

    #pragma omp single copyprivate(lock_ptr)
    {
        lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
        omp_init_lock( lock_ptr );
    }

    return lock_ptr;
}

```

## A.26 The threadprivate directive

The following examples demonstrate how to use the [threadprivate](#) directive to give each thread a separate counter.

### Example 1

```

int counter = 0;
#pragma omp threadprivate(counter)

int sub()
{
    counter++;
    return(counter);
}

```

### Example 2

```

int sub()
{
    static int counter = 0;
    #pragma omp threadprivate(counter)
    counter++;
    return(counter);
}

```

## A.27 C99 variable length arrays

The following example demonstrates how to use C99 Variable Length Arrays (VLAs) in a [firstprivate](#) directive.

### NOTE

Variable length arrays aren't currently supported in Visual C++.

```

void f(int m, int C[m][m])
{
    double v1[m];
    ...
    #pragma omp parallel firstprivate(C, v1)
    ...
}

```

## A.28 The num\_threads clause

The following example demonstrates the `num_threads` clause. The parallel region is executed with a maximum of 10 threads.

```
#include <omp.h>
main()
{
    omp_set_dynamic(1);
    ...
    #pragma omp parallel num_threads(10)
    {
        ... parallel region ...
    }
}
```

## A.29 Work-sharing constructs inside a critical construct

The following example demonstrates using a work-sharing construct inside a `critical` construct. This example is compliant because the work-sharing construct and the `critical` construct don't bind to the same parallel region.

```
void f()
{
    int i = 1;
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            #pragma omp critical (name)
            {
                #pragma omp parallel
                {
                    #pragma omp single
                    {
                        i++;
                    }
                }
            }
        }
    }
}
```

## A.30 Reprivatization

The following example demonstrates the reprivatization of variables. Private variables can be marked `private` again in a nested directive. You don't need to share those variables in the enclosing parallel region.

```
int i, a;
...
#pragma omp parallel private(a)
{
    ...
    #pragma omp parallel for private(a)
    for (i=0; i<10; i++)
    {
        ...
    }
}
```

## A.31 Thread-safe lock functions

The following C++ example demonstrates how to initialize an array of locks in a parallel region by using [omp\\_init\\_lock](#).

```
// A_13_omp_init_lock.cpp
// compile with: /openmp
#include <omp.h>

omp_lock_t *new_locks() {
    int i;
    omp_lock_t *lock = new omp_lock_t[1000];
    #pragma omp parallel for private(i)
    for (i = 0 ; i < 1000 ; i++)
        omp_init_lock(&lock[i]);

    return lock;
}

int main () {}
```

# B. Stubs for run-time library functions

1/28/2019 • 2 minutes to read • [Edit Online](#)

This section provides stubs for the run-time library functions defined in the OpenMP C and C++ API. The stubs are provided to enable portability to platforms that don't support the OpenMP C and C++ API. On these platforms, OpenMP programs must be linked with a library containing these stub functions. The stub functions assume that the directives in the OpenMP program are ignored. As such, they emulate serial semantics.

## NOTE

The lock variable that appears in the lock functions must be accessed exclusively through these functions. It should not be initialized or otherwise modified in the user program. Users should not make assumptions about mechanisms used by OpenMP C and C++ implementations to implement locks based on the scheme used by the stub functions.

## Code

```
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"
#ifdef __cplusplus
extern "C" {
#endif

void omp_set_num_threads(int num_threads)
{
}
int omp_get_num_threads(void)
{
    return 1;
}
int omp_get_max_threads(void)
{
    return 1;
}
int omp_get_thread_num(void)
{
    return 0;
}
int omp_get_num_procs(void)
{
    return 1;
}
void omp_set_dynamic(int dynamic_threads)
{
}
int omp_get_dynamic(void)
{
    return 0;
}
int omp_in_parallel(void)
{
    return 0;
}
void omp_set_nested(int nested)
{
}
int omp_get_nested(void)
{
```

```

    return 0;
}
enum {UNLOCKED = -1, INIT, LOCKED};
void omp_init_lock(omp_lock_t *lock)
{
    *lock = UNLOCKED;
}
void omp_destroy_lock(omp_lock_t *lock)
{
    *lock = INIT;
}
void omp_set_lock(omp_lock_t *lock)
{
    if (*lock == UNLOCKED)
    {
        *lock = LOCKED;
    }
    else
        if (*lock == LOCKED)
        {
            fprintf_s(stderr, "error: deadlock in using lock variable\n");
            exit(1);
        } else {
            fprintf_s(stderr, "error: lock not initialized\n");
            exit(1);
        }
}

void omp_unset_lock(omp_lock_t *lock)
{
    if (*lock == LOCKED)
    {
        *lock = UNLOCKED;
    }
    else
        if (*lock == UNLOCKED)
        {
            fprintf_s(stderr, "error: lock not set\n");
            exit(1);
        } else {
            fprintf_s(stderr, "error: lock not initialized\n");
            exit(1);
        }
}

int omp_test_lock(omp_lock_t *lock)
{
    if (*lock == UNLOCKED)
    {
        *lock = LOCKED;
        return 1;
    } else if (*lock == LOCKED) {
        return 0;
    } else {
        fprintf_s(stderr, "error: lock not initialized\n");
        exit(1);
    }
}

#ifdef OMP_NEST_LOCK_T
typedef struct { // This really belongs in omp.h
    int owner;
    int count;
} omp_nest_lock_t;
#endif
enum {MASTER = 0};
void omp_init_nest_lock(omp_nest_lock_t *lock)
{
    lock->owner = UNLOCKED;
}

```

```

    lock->count = 0;
}
void omp_destroy_nest_lock(omp_nest_lock_t *lock)
{
    lock->owner = UNLOCKED;
    lock->count = UNLOCKED;
}

void omp_set_nest_lock(omp_nest_lock_t *lock)
{
    if (lock->owner == MASTER && lock->count >= 1)
    {
        lock->count++;
    } else
        if (lock->owner == UNLOCKED && lock->count == 0)
        {
            lock->owner = MASTER;
            lock->count = 1;
        } else
        {
            fprintf_s(stderr, "error: lock corrupted or not initialized\n");
            exit(1);
        }
}

void omp_unset_nest_lock(omp_nest_lock_t *lock)
{
    if (lock->owner == MASTER && lock->count >= 1)
    {
        lock->count--;
        if (lock->count == 0)
        {
            lock->owner = UNLOCKED;
        }
    } else
        if (lock->owner == UNLOCKED && lock->count == 0)
        {
            fprintf_s(stderr, "error: lock not set\n");
            exit(1);
        } else
        {
            fprintf_s(stderr, "error: lock corrupted or not initialized\n");
            exit(1);
        }
}

int omp_test_nest_lock(omp_nest_lock_t *lock)
{
    omp_set_nest_lock(lock);
    return lock->count;
}

double omp_get_wtime(void)
{
    // This function does not provide a working
    // wallclock timer. Replace it with a version
    // customized for the target machine.
    return 0.0;
}

double omp_get_wtick(void)
{
    // This function does not provide a working
    // clock tick function. Replace it with
    // a version customized for the target machine.
    return 365. * 86400.;
}

#ifdef _cplusplus

```

```
#endif _epscppad
}  
#endif
```

# C. OpenMP C and C++ grammar

1/18/2019 • 2 minutes to read • [Edit Online](#)

C.1 Notation

C.2 Rules

## C.1 Notation

The grammar rules consist of the name for a non-terminal, followed by a colon, followed by replacement alternatives on separate lines.

The syntactic expression  $term_{opt}$  indicates that the term is optional within the replacement.

The syntactic expression  $term_{optseq}$  is equivalent to  $term-seq_{opt}$  with the following additional rules:

*term-seq*:

*term*

*term-seq term*

*term-seq*   *term*

## C.2 Rules

The notation is described in section 6.1 of the C standard. This grammar appendix shows the extensions to the base language grammar for the OpenMP C and C++ directives.

***/\* in C++ (ISO/IEC 14882:1998) \*/***

*statement-seq*:

*statement*

*openmp-directive*

*statement-seq statement*

*statement-seq openmp-directive*

***/\* in C90 (ISO/IEC 9899:1990) \*/***

*statement-list*:

*statement*

*openmp-directive*

*statement-list statement*

*statement-list openmp-directive*

***/\* in C99 (ISO/IEC 9899:1999) \*/***

*block-item*:

*declaration*

*statement*

*openmp-directive*

***/\* standard statements \*/***

*statement*:

*openmp-construct*

*openmp-construct*:

*parallel-construct*  
*for-construct*  
*sections-construct*  
*single-construct*  
*parallel-for-construct*  
*parallel-sections-construct*  
*master-construct*  
*critical-construct*  
*atomic-construct*  
*ordered-construct*

*openmp-directive:*

*barrier-directive*  
*flush-directive*

*structured-block:*

*statement*

*parallel-construct:*

*parallel-directive structured-block*

*parallel-directive:*

```
# pragma omp parallel parallel-clauseoptseq new-line
```

*parallel-clause:*

*unique-parallel-clause*  
*data-clause*

*unique-parallel-clause:*

```
if ( expression )  
num_threads ( expression )
```

*for-construct:*

*for-directive iteration-statement*

*for-directive:*

```
# pragma omp for for-clauseoptseq new-line
```

*for-clause:*

*unique-for-clause*  
*data-clause*  
nowait

*unique-for-clause:*

```
ordered  
schedule ( schedule-kind )  
schedule ( schedule-kind , expression )
```

*schedule-kind:*

```
static  
dynamic  
guided  
runtime
```

*sections-construct:*

*sections-directive section-scope*

*sections-directive:*

```
# pragma omp sections sections-clauseoptseq new-line
```

*sections-clause:*

*data-clause*

```
nowait
```

*section-scope:*

{ *section-sequence* }

*section-sequence:*

*section-directive*<sub>opt</sub> *structured-block*

*section-sequence* *section-directive* *structured-block*

*section-directive:*

```
# pragma omp section new-line
```

*single-construct:*

*single-directive* *structured-block*

*single-directive:*

```
# pragma omp single single-clauseoptseq new-line
```

*single-clause:*

*data-clause*

```
nowait
```

*parallel-for-construct:*

*parallel-for-directive* *iteration-statement*

*parallel-for-directive:*

```
# pragma omp parallel for parallel-for-clauseoptseq new-line
```

*parallel-for-clause:*

*unique-parallel-clause*

*unique-for-clause*

*data-clause*

*parallel-sections-construct:*

*parallel-sections-directive* *section-scope*

*parallel-sections-directive:*

```
# pragma omp parallel sections parallel-sections-clauseoptseq new-line
```

*parallel-sections-clause:*

*unique-parallel-clause*

*data-clause*

*master-construct:*

*master-directive* *structured-block*

*master-directive:*

```
# pragma omp master new-line
```

*critical-construct:*

*critical-directive* *structured-block*

*critical-directive:*

```
# pragma omp critical region-phraseopt new-line
```

*region-phrase:*

*(identifier)*

*barrier-directive:*

```
# pragma omp barrier new-line
```

*atomic-construct:*

*atomic-directive expression-statement*

*atomic-directive:*

```
# pragma omp atomic new-line
```

*flush-directive:*

```
# pragma omp flush flush-varsopt new-line
```

*flush-vars:*

*(variable-list)*

*ordered-construct:*

*ordered-directive structured-block*

*ordered-directive:*

```
# pragma omp ordered new-line
```

**/\* standard declarations \*/**

*declaration:*

*threadprivate-directive*

*threadprivate-directive:*

```
# pragma omp threadprivate ( variable-list ) new-line
```

*data-clause:*

```
private ( variable-list )
```

```
copyprivate ( variable-list )
```

```
firstprivate ( variable-list )
```

```
lastprivate ( variable-list )
```

```
shared ( variable-list )
```

```
default ( shared )
```

```
default ( none )
```

```
reduction ( reduction-operator : variable-list )
```

```
copyin ( variable-list )
```

*reduction-operator:*

One of: + \\* - & ^ | && ||

**/\* in C \*/**

*variable-list:*

*identifier*

*variable-list* , *identifier*

**/\* in C++ \*/**

*variable-list:*

*id-expression*

*variable-list* , *id-expression*

## D. The schedule clause

1/28/2019 • 5 minutes to read • [Edit Online](#)

A parallel region has at least one barrier, at its end, and may have additional barriers within it. At each barrier, the other members of the team must wait for the last thread to arrive. To minimize this wait time, shared work should be distributed so that all threads arrive at the barrier at about the same time. If some of that shared work is contained in `for` constructs, the `schedule` clause can be used for this purpose.

When there are repeated references to the same objects, the choice of schedule for a `for` construct may be determined primarily by characteristics of the memory system, such as the presence and size of caches and whether memory access times are uniform or nonuniform. Such considerations may make it preferable to have each thread consistently refer to the same set of elements of an array in a series of loops, even if some threads are assigned relatively less work in some of the loops. This setup can be done by using the `static` schedule with the same bounds for all the loops. In the following example, zero is used as the lower bound in the second loop, even though `k` would be more natural if the schedule were not important.

```
#pragma omp parallel
{
  #pragma omp for schedule(static)
  for(i=0; i<n; i++)
    a[i] = work1(i);
  #pragma omp for schedule(static)
  for(i=0; i<n; i++)
    if(i>=k) a[i] += work2(i);
}
```

In the remaining examples, it's assumed that memory access isn't the dominant consideration. Unless otherwise stated, that all threads receive comparable computational resources. In these cases, the choice of schedule for a `for` construct depends on all the shared work that's to be performed between the nearest preceding barrier and either the implied closing barrier or the nearest upcoming barrier, if there's a `nowait` clause. For each kind of schedule, a short example shows how that schedule kind is likely to be the best choice. A brief discussion follows each example.

The `static` schedule is also appropriate for the simplest case, a parallel region containing a single `for` construct, with each iteration requiring the same amount of work.

```
#pragma omp parallel for schedule(static)
for(i=0; i<n; i++) {
  invariant_amount_of_work(i);
}
```

The `static` schedule is characterized by the properties that each thread gets approximately the same number of iterations as any other thread, and each thread can independently determine the iterations assigned to it. Thus no synchronization is required to distribute the work, and, under the assumption that each iteration requires the same amount of work, all threads should finish at about the same time.

For a team of  $p$  threads, let  $\text{ceiling}(n/p)$  be the integer  $q$ , which satisfies  $n = p*q - r$  with  $0 \leq r < p$ . One implementation of the `static` schedule for this example would assign  $q$  iterations to the first  $p-1$  threads, and  $q-r$  iterations to the last thread. Another acceptable implementation would assign  $q$  iterations to the first  $p-r$  threads, and  $q-1$  iterations to the remaining  $r$  threads. This example illustrates why a program shouldn't rely on the details of a particular implementation.

The `dynamic` schedule is appropriate for the case of a `for` construct with the iterations requiring varying, or even unpredictable, amounts of work.

```
#pragma omp parallel for schedule(dynamic)
for(i=0; i<n; i++) {
    unpredictable_amount_of_work(i);
}
```

The `dynamic` schedule is characterized by the property that no thread waits at the barrier for longer than it takes another thread to execute its final iteration. This requirement means iterations must be assigned one at a time to threads as they become available, with synchronization for each assignment. The synchronization overhead can be reduced by specifying a minimum chunk size  $k$  greater than 1, so that threads are assigned  $k$  at a time until fewer than  $k$  remain. This guarantees that no thread waits at the barrier longer than it takes another thread to execute its final chunk of (at most)  $k$  iterations.

The `dynamic` schedule can be useful if the threads receive varying computational resources, which has much the same effect as varying amounts of work for each iteration. Similarly, the dynamic schedule can also be useful if the threads arrive at the `for` construct at varying times, though in some of these cases the `guided` schedule may be preferable.

The `guided` schedule is appropriate for the case in which the threads may arrive at varying times at a `for` construct with each iteration requiring about the same amount of work. This situation can happen if, for example, the `for` construct is preceded by one or more sections or `for` constructs with `nowait` clauses.

```
#pragma omp parallel
{
    #pragma omp sections nowait
    {
        // ...
    }
    #pragma omp for schedule(guided)
    for(i=0; i<n; i++) {
        invariant_amount_of_work(i);
    }
}
```

Like `dynamic`, the `guided` schedule guarantees that no thread waits at the barrier longer than it takes another thread to execute its final iteration, or final  $k$  iterations if a chunk size of  $k$  is specified. Among such schedules, the `guided` schedule is characterized by the property that it requires the fewest synchronizations. For chunk size  $k$ , a typical implementation will assign  $q = \text{ceiling}(n/p)$  iterations to the first available thread, set  $n$  to the larger of  $n-q$  and  $p*k$ , and repeat until all iterations are assigned.

When the choice of the optimum schedule isn't as clear as it is for these examples, the `runtime` schedule is convenient for experimenting with different schedules and chunk sizes without having to modify and recompile the program. It can also be useful when the optimum schedule depends (in some predictable way) on the input data to which the program is applied.

To see an example of the trade-offs between different schedules, consider sharing 1000 iterations among eight threads. Suppose there's an invariant amount of work in each iteration, and use that as the unit of time.

If all threads start at the same time, the `static` schedule will cause the construct to execute in 125 units, with no synchronization. But suppose that one thread is 100 units late in arriving. Then the remaining seven threads wait for 100 units at the barrier, and the execution time for the whole construct increases to 225.

Because both the `dynamic` and `guided` schedules make sure that no thread waits for more than one unit at the barrier, the delayed thread causes their execution times for the construct to increase only to 138 units, possibly

increased by delays from synchronization. If such delays aren't negligible, it becomes important that the number of synchronizations is 1000 for `dynamic` but only 41 for `guided`, assuming the default chunk size of one. With a chunk size of 25, `dynamic` and `guided` both finish in 150 units, plus any delays from the required synchronizations, which now number only 40 and 20, respectively.

# E. Implementation-defined behaviors in OpenMP

## C/C++

1/28/2019 • 2 minutes to read • [Edit Online](#)

This appendix summarizes the behaviors that are described as "implementation-defined" in this API. Each behavior is cross-referenced back to its description in the main specification.

## Remarks

An implementation is required to define and document its behavior in these cases, but this list may be incomplete.

- **Number of threads:** If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads requested for the parallel region is more than the number that the run-time system can supply, the behavior of the program is implementation-defined (see page 9).

In Visual C++, for a non-nested parallel region, 64 threads (the maximum) will be provided.

- **Number of processors:** The number of physical processors actually hosting the threads at any given time is implementation-defined (see page 10).

In Visual C++, this number isn't constant, and is controlled by the operating system.

- **Creating teams of threads:** The number of threads in a team that execute a nested parallel region is implementation-defined (see page 10).

In Visual C++, this number is determined by the operating system.

- **schedule(runtime):** The decision about scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable. If this environment variable isn't set, the resulting schedule is implementation-defined (see page 13).

In Visual C++, schedule type is `static` with no chunk size.

- **Default scheduling:** In the absence of the schedule clause, the default schedule is implementation-defined (see page 13).

In Visual C++, the default schedule type is `static` with no chunk size.

- **ATOMIC:** It's implementation-defined whether an implementation replaces all `atomic` directives with `critical` directives that have the same unique name (see page 20).

In Visual C++, if data modified by `atomic` isn't on a natural alignment or if it's one or two bytes long, all atomic operations that satisfy that property will use one critical section. Otherwise, critical sections won't be used.

- **omp\_get\_num\_threads:** If the number of threads hasn't been explicitly set by the user, the default is implementation-defined (see page 9).

In Visual C++, the default number of threads is equal to the number of processors.

- **omp\_set\_dynamic:** The default for dynamic thread adjustment is implementation-defined.

In Visual C++, the default is `FALSE`.

- **omp\_set\_nested:** When nested parallelism is enabled, the number of threads used to execute nested

parallel regions is implementation-defined.

In Visual C++, the number of threads is determined by the operating system.

- `OMP_SCHEDULE` environment variable: The default value for this environment variable is implementation-defined.

In Visual C++, schedule type is `static` with no chunk size.

- `OMP_NUM_THREADS` environment variable: If no value is specified for the `OMP_NUM_THREADS` environment variable, or if the value specified isn't a positive integer, or if the value is greater than the maximum number of threads the system can support, the number of threads to use is implementation-defined.

In Visual C++, if value specified is zero or less, the number of threads is equal to the number of processors. If value is greater than 64, the number of threads is 64.

- `OMP_DYNAMIC` environment variable: The default value is implementation-defined.

In Visual C++, the default is `FALSE`.

# F. New features and clarifications in version 2.0

1/28/2019 • 2 minutes to read • [Edit Online](#)

This appendix summarizes the key changes made to the OpenMP C/C++ specification in moving from version 1.0 to version 2.0. The following items are new features added to the specification:

- Commas are allowed in OpenMP [directives](#).
- Addition of the `num_threads` clause. This clause allows a user to request a specific number of threads for a [parallel construct](#).
- The `threadprivate` directive has been extended to accept static block-scope variables.
- C99 Variable Length Arrays are complete types and can be specified anywhere complete types are allowed, such as in the lists of `private`, `firstprivate`, and `lastprivate` clauses (see [section 2.7.2](#)).
- A private variable in a parallel region can be marked `private` again in a nested directive.
- The `copyprivate` clause has been added. It provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members. It's an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level). The `copyprivate` clause can only appear on the `single` directive.
- Addition of timing routines `omp_get_wtick` and `omp_get_wtime` similar to the MPI routines. These functions are necessary to do wall clock timings.
- An appendix with a list of [implementation-defined behaviors](#) in OpenMP C/C++ has been added. An implementation is required to define and document its behavior in these cases.
- The following changes serve to clarify or correct features in the previous OpenMP API specification for C/C++:
  - Clarified that the behavior of `omp_set_nested` and `omp_set_dynamic` when `omp_in_parallel` returns nonzero is undefined.
  - Clarified [directive nesting](#) when nested parallel is used.
  - The [lock initialization](#) and [lock destruction](#) functions can be called in a parallel region.
  - New examples have been added to [appendix A](#).

# OpenMP Library Reference

4/22/2019 • 2 minutes to read • [Edit Online](#)

Provides links to constructs used in the OpenMP API.

The Visual C++ implementation of the OpenMP standard includes the following constructs.

CONSTRUCT	DESCRIPTION
<a href="#">Directives</a>	Provides links to directives used in the OpenMP API.
<a href="#">Clauses</a>	Provides links to clauses used in the OpenMP API.
<a href="#">Functions</a>	Provides links to functions used in the OpenMP API.
<a href="#">Environment Variables</a>	Provides links to environment variables used in the OpenMP API.

The Visual C++ OpenMP run-time library functions are contained in the following libraries.

OPENMP RUN-TIME LIBRARY	CHARACTERISTICS
VCOMP.LIB	Multithreaded, dynamic link (import library for VCOMP.LIB).
VCOMP.D.LIB	Multithreaded, dynamic link (import library for VCOMP.D.LIB) (debug)

If `_DEBUG` is defined in a compilation and if `#include omp.h` is in source code, `VCOMP.D.LIB` will be the default lib, otherwise, `VCOMP.LIB` will be used.

You can use `/NODEFAULTLIB (ignore libraries)` to remove the default lib and explicitly link with the lib of your choice.

The OpenMP DLLs are in the Visual C++ redistributable directory and need to be distributed with applications that use OpenMP.

## See also

[OpenMP](#)

# OpenMP Directives

4/22/2019 • 10 minutes to read • [Edit Online](#)

Provides links to directives used in the OpenMP API.

Visual C++ supports the following OpenMP directives.

For parallel work-sharing:

DIRECTIVE	DESCRIPTION
<a href="#">parallel</a>	Defines a parallel region, which is code that will be executed by multiple threads in parallel.
<a href="#">for</a>	Causes the work done in a <code>for</code> loop inside a parallel region to be divided among threads.
<a href="#">sections</a>	Identifies code sections to be divided among all threads.
<a href="#">single</a>	Lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.

For master and synchronization:

DIRECTIVE	DESCRIPTION
<a href="#">master</a>	Specifies that only the master thread should execute a section of the program.
<a href="#">critical</a>	Specifies that code is only executed on one thread at a time.
<a href="#">barrier</a>	Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.
<a href="#">atomic</a>	Specifies that a memory location that will be updated atomically.
<a href="#">flush</a>	Specifies that all threads have the same view of memory for all shared objects.
<a href="#">ordered</a>	Specifies that code under a parallelized <code>for</code> loop should be executed like a sequential loop.

For data environment:

DIRECTIVE	DESCRIPTION
<a href="#">threadprivate</a>	Specifies that a variable is private to a thread.

atomic

Specifies that a memory location that will be updated atomically.

```
#pragma omp atomic
expression
```

### Parameters

*expression*

The statement that has the *lvalue*, whose memory location you want to protect against more than one write.

### Remarks

The `atomic` directive supports no clauses.

For more information, see [2.6.4 atomic construct](#).

### Example

```
// omp_atomic.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

#define MAX 10

int main() {
    int count = 0;
    #pragma omp parallel num_threads(MAX)
    {
        #pragma omp atomic
        count++;
    }
    printf_s("Number of threads: %d\n", count);
}
```

```
Number of threads: 10
```

## barrier

Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.

```
#pragma omp barrier
```

### Remarks

The `barrier` directive supports no clauses.

For more information, see [2.6.3 barrier directive](#).

### Example

For a sample of how to use `barrier`, see [master](#).

## critical

Specifies that code is only be executed on one thread at a time.

```
#pragma omp critical [(name)]
{
    code_block
}
```

## Parameters

*name*

(Optional) A name to identify the critical code. The name must be enclosed in parentheses.

## Remarks

The `critical` directive supports no clauses.

For more information, see [2.6.2 critical construct](#).

## Example

```
// omp_critical.cpp
// compile with: /openmp
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

int main()
{
    int i;
    int max;
    int a[SIZE];

    for (i = 0; i < SIZE; i++)
    {
        a[i] = rand();
        printf_s("%d\n", a[i]);
    }

    max = a[0];
    #pragma omp parallel for num_threads(4)
    for (i = 1; i < SIZE; i++)
    {
        if (a[i] > max)
        {
            #pragma omp critical
            {
                // compare a[i] and max again because max
                // could have been changed by another thread after
                // the comparison outside the critical section
                if (a[i] > max)
                    max = a[i];
            }
        }
    }

    printf_s("max = %d\n", max);
}
```

```
41
18467
6334
26500
19169
15724
11478
29358
26962
24464
max = 29358
```

## flush

Specifies that all threads have the same view of memory for all shared objects.

```
#pragma omp flush [(var)]
```

### Parameters

*var*

(Optional) A comma-separated list of variables that represent objects you want to synchronize. If *var* isn't specified, all memory is flushed.

### Remarks

The `flush` directive supports no clauses.

For more information, see [2.6.5 flush directive](#).

### Example

```

// omp_flush.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

void read(int *data) {
    printf_s("read data\n");
    *data = 1;
}

void process(int *data) {
    printf_s("process data\n");
    (*data)++;
}

int main() {
    int data;
    int flag;

    flag = 0;

#pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        {
            printf_s("Thread %d: ", omp_get_thread_num( ));
            read(&data);
            #pragma omp flush(data)
            flag = 1;
            #pragma omp flush(flag)
            // Do more work.
        }

        #pragma omp section
        {
            while (!flag) {
                #pragma omp flush(flag)
            }
            #pragma omp flush(data)

            printf_s("Thread %d: ", omp_get_thread_num( ));
            process(&data);
            printf_s("data = %d\n", data);
        }
    }
}

```

```

Thread 0: read data
Thread 1: process data
data = 2

```

## for

Causes the work done in a `for` loop inside a parallel region to be divided among threads.

```

#pragma omp [parallel] for [clauses]
    for_statement

```

### Parameters

*clauses*

(Optional) Zero or more clauses, see the **Remarks** section.

### *for\_statement*

A `for` loop. Undefined behavior will result if user code in the `for` loop changes the index variable.

### Remarks

The `for` directive supports the following clauses:

- `private`
- `firstprivate`
- `lastprivate`
- `reduction`
- `ordered`
- `schedule`
- `nowait`

If `parallel` is also specified, `clauses` can be any clause accepted by the `parallel` or `for` directives, except `nowait`.

For more information, see [2.4.1 for construct](#).

### Example

```

// omp_for.cpp
// compile with: /openmp
#include <stdio.h>
#include <math.h>
#include <omp.h>

#define NUM_THREADS 4
#define NUM_START 1
#define NUM_END 10

int main() {
    int i, nRet = 0, nSum = 0, nStart = NUM_START, nEnd = NUM_END;
    int nThreads = 0, nTmp = nStart + nEnd;
    unsigned uTmp = (unsigned((abs(nStart - nEnd) + 1)) *
                    unsigned(abs(nTmp))) / 2;

    int nSumCalc = uTmp;

    if (nTmp < 0)
        nSumCalc = -nSumCalc;

    omp_set_num_threads(NUM_THREADS);

#pragma omp parallel default(none) private(i) shared(nSum, nThreads, nStart, nEnd)
    {
        #pragma omp master
        nThreads = omp_get_num_threads();

        #pragma omp for
        for (i=nStart; i<=nEnd; ++i) {
            #pragma omp atomic
            nSum += i;
        }
    }

    if (nThreads == NUM_THREADS) {
        printf_s("%d OpenMP threads were used.\n", NUM_THREADS);
        nRet = 0;
    }
    else {
        printf_s("Expected %d OpenMP threads, but %d were used.\n",
                NUM_THREADS, nThreads);
        nRet = 1;
    }

    if (nSum != nSumCalc) {
        printf_s("The sum of %d through %d should be %d, "
                "but %d was reported!\n",
                NUM_START, NUM_END, nSumCalc, nSum);
        nRet = 1;
    }
    else
        printf_s("The sum of %d through %d is %d\n",
                NUM_START, NUM_END, nSum);
}

```

```

4 OpenMP threads were used.
The sum of 1 through 10 is 55

```

## master

Specifies that only the master thread should execute a section of the program.

```
#pragma omp master
{
    code_block
}
```

## Remarks

The `master` directive supports no clauses.

The `single` directive lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.

For more information, see [2.6.1 master construct](#).

## Example

```
// omp_master.cpp
// compile with: /openmp
#include <omp.h>
#include <stdio.h>

int main( )
{
    int a[5], i;

    #pragma omp parallel
    {
        // Perform some computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;

        // Print intermediate results.
        #pragma omp master
        for (i = 0; i < 5; i++)
            printf_s("a[%d] = %d\n", i, a[i]);

        // Wait.
        #pragma omp barrier

        // Continue with the computation.
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] += i;
    }
}
```

```
a[0] = 0
a[1] = 1
a[2] = 4
a[3] = 9
a[4] = 16
```

## ordered

Specifies that code under a parallelized `for` loop should be executed like a sequential loop.

```
#pragma omp ordered
    structured-block
```

## Remarks

The `ordered` directive must be within the dynamic extent of a `for` or `parallel for` construct with an `ordered` clause.

The `ordered` directive supports no clauses.

For more information, see [2.6.6 ordered construct](#).

## Example

```
// omp_ordered.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

static float a[1000], b[1000], c[1000];

void test(int first, int last)
{
    #pragma omp for schedule(static) ordered
    for (int i = first; i <= last; ++i) {
        // Do something here.
        if (i % 2)
        {
            #pragma omp ordered
            printf_s("test() iteration %d\n", i);
        }
    }
}

void test2(int iter)
{
    #pragma omp ordered
    printf_s("test2() iteration %d\n", iter);
}

int main( )
{
    int i;
    #pragma omp parallel
    {
        test(1, 8);
        #pragma omp for ordered
        for (i = 0 ; i < 5 ; i++)
            test2(i);
    }
}
```

```
test() iteration 1
test() iteration 3
test() iteration 5
test() iteration 7
test2() iteration 0
test2() iteration 1
test2() iteration 2
test2() iteration 3
test2() iteration 4
```

## parallel

Defines a parallel region, which is code that will be executed by multiple threads in parallel.

```
#pragma omp parallel [clauses]
{
    code_block
}
```

## Parameters

*clauses*

(Optional) Zero or more clauses, see the **Remarks** section.

## Remarks

The `parallel` directive supports the following clauses:

- [if](#)
- [private](#)
- [firstprivate](#)
- [default](#)
- [shared](#)
- [copyin](#)
- [reduction](#)
- [num\\_threads](#)

`parallel` can also be used with the [for](#) and [sections](#) directives.

For more information, see [2.3 parallel construct](#).

## Example

The following sample shows how to set the number of threads and define a parallel region. The number of threads is equal by default to the number of logical processors on the machine. For example, if you have a machine with one physical processor that has hyperthreading enabled, it will have two logical processors and two threads. The order of output can vary on different machines.

```
// omp_parallel.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel num_threads(4)
    {
        int i = omp_get_thread_num();
        printf_s("Hello from thread %d\n", i);
    }
}
```

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

## sections

Identifies code sections to be divided among all threads.

```
#pragma omp [parallel] sections [clauses]
{
  #pragma omp section
  {
    code_block
  }
}
```

## Parameters

*clauses*

(Optional) Zero or more clauses, see the **Remarks** section.

## Remarks

The `sections` directive can contain zero or more `section` directives.

The `sections` directive supports the following clauses:

- `private`
- `firstprivate`
- `lastprivate`
- `reduction`
- `nowait`

If `parallel` is also specified, `clauses` can be any clause accepted by the `parallel` or `sections` directives, except `nowait`.

For more information, see [2.4.2 sections construct](#).

## Example

```
// omp_sections.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main() {
  #pragma omp parallel sections num_threads(4)
  {
    printf_s("Hello from thread %d\n", omp_get_thread_num());
    #pragma omp section
    printf_s("Hello from thread %d\n", omp_get_thread_num());
  }
}
```

```
Hello from thread 0
Hello from thread 0
```

## single

Lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.

```
#pragma omp single [clauses]
{
  code_block
}
```

## Parameters

*clauses*

(Optional) Zero or more clauses, see the **Remarks** section.

## Remarks

The `single` directive supports the following clauses:

- `private`
- `firstprivate`
- `copyprivate`
- `nowait`

The `master` directive lets you specify that a section of code should be executed only on the master thread.

For more information, see [2.4.3 single construct](#).

## Example

```
// omp_single.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel num_threads(2)
    {
        #pragma omp single
        // Only a single thread can read the input.
        printf_s("read input\n");

        // Multiple threads in the team compute the results.
        printf_s("compute results\n");

        #pragma omp single
        // Only a single thread can write the output.
        printf_s("write output\n");
    }
}
```

```
read input
compute results
compute results
write output
```

## threadprivate

Specifies that a variable is private to a thread.

```
#pragma omp threadprivate(var)
```

## Parameters

*var*

A comma-separated list of variables that you want to make private to a thread. *var* must be either a global- or namespace-scoped variable or a local static variable.

## Remarks

The `threadprivate` directive supports no clauses.

The `threadprivate` directive is based on the `thread` attribute using the `__declspec` keyword; limits on `__declspec(thread)` apply to `threadprivate`. For example, a `threadprivate` variable will exist in any thread started in the process, not just those threads that are part of a thread team spawned by a parallel region. Be aware of this implementation detail; you may notice that constructors for a `threadprivate` user-defined type are called more often than expected.

You can use `threadprivate` in a DLL that is statically loaded at process startup, however you can't use `threadprivate` in any DLL that will be loaded via `LoadLibrary` such as DLLs that are loaded with `/DELAYLOAD (delay load import)`, which also uses `LoadLibrary`.

A `threadprivate` variable of a *destructible* type isn't guaranteed to have its destructor called. For example:

```
struct MyType
{
    ~MyType();
};

MyType threaded_var;
#pragma omp threadprivate(threaded_var)
int main()
{
    #pragma omp parallel
    {}
}
```

Users have no control as to when the threads constituting the parallel region will terminate. If those threads exist when the process exits, the threads won't be notified about the process exit, and the destructor won't be called for `threaded_var` on any thread except the one that exits (here, the primary thread). So code shouldn't count on proper destruction of `threadprivate` variables.

For more information, see [2.7.1 threadprivate directive](#).

### Example

For a sample of using `threadprivate`, see [private](#).

# OpenMP Clauses

4/22/2019 • 14 minutes to read • [Edit Online](#)

Provides links to clauses used in the OpenMP API.

Visual C++ supports the following OpenMP clauses.

For general attributes:

CLAUSE	DESCRIPTION
<a href="#">if</a>	Specifies whether a loop should be executed in parallel or in serial.
<a href="#">num_threads</a>	Sets the number of threads in a thread team.
<a href="#">ordered</a>	Required on a parallel <a href="#">for</a> statement if an <a href="#">ordered</a> directive is to be used in the loop.
<a href="#">schedule</a>	Applies to the <a href="#">for</a> directive.
<a href="#">nowait</a>	Overrides the barrier implicit in a directive.

For data-sharing attributes:

CLAUSE	DESCRIPTION
<a href="#">private</a>	Specifies that each thread should have its own instance of a variable.
<a href="#">firstprivate</a>	Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.
<a href="#">lastprivate</a>	Specifies that the enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section (#pragma sections).
<a href="#">shared</a>	Specifies that one or more variables should be shared among all threads.
<a href="#">default</a>	Specifies the behavior of unscoped variables in a parallel region.
<a href="#">reduction</a>	Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.
<a href="#">copyin</a>	Allows threads to access the master thread's value, for a <a href="#">threadprivate</a> variable.

CLAUSE	DESCRIPTION
<code>copyprivate</code>	Specifies that one or more variables should be shared among all threads.

## copyin

Allows threads to access the master thread's value, for a `threadprivate` variable.

```
copyin(var)
```

### Parameters

*var*

The `threadprivate` variable that will be initialized with the variable's value in the master thread, as it exists before the parallel construct.

### Remarks

`copyin` applies to the following directives:

- [parallel](#)
- [for](#)
- [sections](#)

For more information, see [2.7.2.7 copyin](#).

### Example

See [threadprivate](#) for an example of using `copyin`.

## copyprivate

Specifies that one or more variables should be shared among all threads.

```
copyprivate(var)
```

### Parameters

*var*

One or more variables to share. If more than one variable is specified, separate variable names with a comma.

### Remarks

`copyprivate` applies to the [single](#) directive.

For more information, see [2.7.2.8 copyprivate](#).

### Example

```

// omp_copyprivate.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

float x, y, fGlobal = 1.0;
#pragma omp threadprivate(x, y)

float get_float() {
    fGlobal += 0.001;
    return fGlobal;
}

void use_float(float f, int t) {
    printf_s("Value = %f, thread = %d\n", f, t);
}

void CopyPrivate(float a, float b) {
    #pragma omp single copyprivate(a, b, x, y)
    {
        a = get_float();
        b = get_float();
        x = get_float();
        y = get_float();
    }

    use_float(a, omp_get_thread_num());
    use_float(b, omp_get_thread_num());
    use_float(x, omp_get_thread_num());
    use_float(y, omp_get_thread_num());
}

int main() {
    float a = 9.99, b = 123.456;

    printf_s("call CopyPrivate from a single thread\n");
    CopyPrivate(9.99, 123.456);

    printf_s("call CopyPrivate from a parallel region\n");
    #pragma omp parallel
    {
        CopyPrivate(a, b);
    }
}

```

```

call CopyPrivate from a single thread
Value = 1.001000, thread = 0
Value = 1.002000, thread = 0
Value = 1.003000, thread = 0
Value = 1.004000, thread = 0
call CopyPrivate from a parallel region
Value = 1.005000, thread = 0
Value = 1.005000, thread = 1
Value = 1.006000, thread = 0
Value = 1.006000, thread = 1
Value = 1.007000, thread = 0
Value = 1.007000, thread = 1
Value = 1.008000, thread = 0
Value = 1.008000, thread = 1

```

## default

Specifies the behavior of unscoped variables in a parallel region.

```
default(shared | none)
```

### Remarks

`shared`, which is in effect if the `default` clause is unspecified, means that any variable in a parallel region will be treated as if it were specified with the `shared` clause. `none` means that any variables used in a parallel region that aren't scoped with the `private`, `shared`, `reduction`, `firstprivate`, or `lastprivate` clause will cause a compiler error.

`default` applies to the following directives:

- [parallel](#)
- [for](#)
- [sections](#)

For more information, see [2.7.2.5 default](#).

### Example

See [private](#) for an example of using `default`.

## firstprivate

Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.

```
firstprivate(var)
```

### Parameters

*var*

The variable to have instances in each thread and that will be initialized with the variable's value, because it exists before the parallel construct. If more than one variable is specified, separate variable names with a comma.

### Remarks

`firstprivate` applies to the following directives:

- [for](#)
- [parallel](#)
- [sections](#)
- [single](#)

For more information, see [2.7.2.2 firstprivate](#).

### Example

For an example of using `firstprivate`, see the example in [private](#).

## if (OpenMP)

Specifies whether a loop should be executed in parallel or in serial.

```
if(expression)
```

### Parameters

*expression*

An integral expression that, if it evaluates to true (nonzero), causes the code in the parallel region to execute in

parallel. If the expression evaluates to false (zero), the parallel region is executed in serial (by a single thread).

### Remarks

`if` applies to the following directives:

- [parallel](#)
- [for](#)
- [sections](#)

For more information, see [2.3 parallel construct](#).

### Example

```
// omp_if.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

void test(int val)
{
    #pragma omp parallel if (val)
    if (omp_in_parallel())
    {
        #pragma omp single
        printf_s("val = %d, parallelized with %d threads\n",
            val, omp_get_num_threads());
    }
    else
    {
        printf_s("val = %d, serialized\n", val);
    }
}

int main( )
{
    omp_set_num_threads(2);
    test(0);
    test(2);
}
```

```
val = 0, serialized
val = 2, parallelized with 2 threads
```

## lastprivate

Specifies that the enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section (`#pragma sections`).

```
lastprivate(var)
```

### Parameters

*var*

The variable that is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section (`#pragma sections`).

### Remarks

`lastprivate` applies to the following directives:

- [for](#)
- [sections](#)

For more information, see [2.7.2.3 lastprivate](#).

### Example

See [schedule](#) for an example of using `lastprivate` clause.

## nowait

Overrides the barrier implicit in a directive.

```
nowait
```

### Remarks

`nowait` applies to the following directives:

- [for](#)
- [sections](#)
- [single](#)

For more information, see [2.4.1 for construct](#), [2.4.2 sections construct](#), and [2.4.3 single construct](#).

### Example

```
// omp_nowait.cpp
// compile with: /openmp /c
#include <stdio.h>

#define SIZE 5

void test(int *a, int *b, int *c, int size)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i = 0; i < size; i++)
            b[i] = a[i] * a[i];

        #pragma omp for nowait
        for (i = 0; i < size; i++)
            c[i] = a[i]/2;
    }
}

int main( )
{
    int a[SIZE], b[SIZE], c[SIZE];
    int i;

    for (i=0; i<SIZE; i++)
        a[i] = i;

    test(a,b,c, SIZE);

    for (i=0; i<SIZE; i++)
        printf_s("%d, %d, %d\n", a[i], b[i], c[i]);
}
```

```
0, 0, 0
1, 1, 0
2, 4, 1
3, 9, 1
4, 16, 2
```

## num\_threads

Sets the number of threads in a thread team.

```
num_threads(num)
```

### Parameters

*num*

The number of threads

### Remarks

The `num_threads` clause has the same functionality as the `omp_set_num_threads` function.

`num_threads` applies to the following directives:

- [parallel](#)
- [for](#)
- [sections](#)

For more information, see [2.3 parallel construct](#).

### Example

See [parallel](#) for an example of using `num_threads` clause.

## ordered

Required on a parallel [for](#) statement if an `ordered` directive is to be used in the loop.

```
ordered
```

### Remarks

`ordered` applies to the [for](#) directive.

For more information, see [2.4.1 for construct](#).

### Example

See [ordered](#) for an example of using `ordered` clause.

## private

Specifies that each thread should have its own instance of a variable.

```
private(var)
```

### Parameters

*var*

The variable to have instances in each thread.

## Remarks

`private` applies to the following directives:

- [for](#)
- [parallel](#)
- [sections](#)
- [single](#)

For more information, see [2.7.2.1 private](#).

## Example

```
// openmp_private.c
// compile with: /openmp
#include <windows.h>
#include <assert.h>
#include <stdio.h>
#include <omp.h>

#define NUM_THREADS 4
#define SLEEP_THREAD 1
#define NUM_LOOPS 2

enum Types {
    ThreadPrivate,
    Private,
    FirstPrivate,
    LastPrivate,
    Shared,
    MAX_TYPES
};

int nSave[NUM_THREADS][MAX_TYPES][NUM_LOOPS] = {{0}};
int nThreadPrivate;

#pragma omp threadprivate(nThreadPrivate)
#pragma warning(disable:4700)

int main() {
    int nPrivate = NUM_THREADS;
    int nFirstPrivate = NUM_THREADS;
    int nLastPrivate = NUM_THREADS;
    int nShared = NUM_THREADS;
    int nRet = 0;
    int i;
    int j;
    int nLoop = 0;

    nThreadPrivate = NUM_THREADS;
    printf_s("These are the variables before entry "
            "into the parallel region.\n");
    printf_s("nThreadPrivate = %d\n", nThreadPrivate);
    printf_s("    nPrivate = %d\n", nPrivate);
    printf_s(" nFirstPrivate = %d\n", nFirstPrivate);
    printf_s(" nLastPrivate = %d\n", nLastPrivate);
    printf_s("    nShared = %d\n\n", nShared);
    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel copyin(nThreadPrivate) private(nPrivate) shared(nShared) firstprivate(nFirstPrivate)
    {
        #pragma omp for schedule(static) lastprivate(nLastPrivate)
        for (i = 0 ; i < NUM_THREADS ; ++i) {
            for (j = 0 ; j < NUM_LOOPS ; ++j) {
                int nThread = omp_get_thread_num();
```

```

assert(nThread < NUM_THREADS);

if (nThread == SLEEP_THREAD)
    Sleep(100);
nSave[nThread][ThreadPrivate][j] = nThreadPrivate;
nSave[nThread][Private][j] = nPrivate;
nSave[nThread][Shared][j] = nShared;
nSave[nThread][FirstPrivate][j] = nFirstPrivate;
nSave[nThread][LastPrivate][j] = nLastPrivate;
nThreadPrivate = nThread;
nPrivate = nThread;
nShared = nThread;
nLastPrivate = nThread;
--nFirstPrivate;
    }
}
}

for (i = 0 ; i < NUM_LOOPS ; ++i) {
    for (j = 0 ; j < NUM_THREADS ; ++j) {
        printf_s("These are the variables at entry of "
            "loop %d of thread %d.\n", i + 1, j);
        printf_s("nThreadPrivate = %d\n",
            nSave[j][ThreadPrivate][i]);
        printf_s("    nPrivate = %d\n",
            nSave[j][Private][i]);
        printf_s(" nFirstPrivate = %d\n",
            nSave[j][FirstPrivate][i]);
        printf_s(" nLastPrivate = %d\n",
            nSave[j][LastPrivate][i]);
        printf_s("    nShared = %d\n\n",
            nSave[j][Shared][i]);
    }
}

printf_s("These are the variables after exit from "
    "the parallel region.\n");
printf_s("nThreadPrivate = %d (The last value in the "
    "master thread)\n", nThreadPrivate);
printf_s("    nPrivate = %d (The value prior to "
    "entering parallel region)\n", nPrivate);
printf_s(" nFirstPrivate = %d (The value prior to "
    "entering parallel region)\n", nFirstPrivate);
printf_s(" nLastPrivate = %d (The value from the "
    "last iteration of the loop)\n", nLastPrivate);
printf_s("    nShared = %d (The value assigned, "
    "from the delayed thread, %d)\n\n",
    nShared, SLEEP_THREAD);
}

```

These are the variables before entry into the parallel region.

```

nThreadPrivate = 4
    nPrivate = 4
nFirstPrivate = 4
    nLastPrivate = 4
        nShared = 4

```

These are the variables at entry of loop 1 of thread 0.

```

nThreadPrivate = 4
    nPrivate = 1310720
nFirstPrivate = 4
    nLastPrivate = 1245104
        nShared = 3

```

These are the variables at entry of loop 1 of thread 1.

```

nThreadPrivate = 4
    nPrivate = 4488

```

```
nFirstPrivate = 4
nLastPrivate = 19748
nShared = 0
```

These are the variables at entry of loop 1 of thread 2.

```
nThreadPrivate = 4
nPrivate = -132514848
nFirstPrivate = 4
nLastPrivate = -513199792
nShared = 4
```

These are the variables at entry of loop 1 of thread 3.

```
nThreadPrivate = 4
nPrivate = 1206
nFirstPrivate = 4
nLastPrivate = 1204
nShared = 2
```

These are the variables at entry of loop 2 of thread 0.

```
nThreadPrivate = 0
nPrivate = 0
nFirstPrivate = 3
nLastPrivate = 0
nShared = 0
```

These are the variables at entry of loop 2 of thread 1.

```
nThreadPrivate = 1
nPrivate = 1
nFirstPrivate = 3
nLastPrivate = 1
nShared = 1
```

These are the variables at entry of loop 2 of thread 2.

```
nThreadPrivate = 2
nPrivate = 2
nFirstPrivate = 3
nLastPrivate = 2
nShared = 2
```

These are the variables at entry of loop 2 of thread 3.

```
nThreadPrivate = 3
nPrivate = 3
nFirstPrivate = 3
nLastPrivate = 3
nShared = 3
```

These are the variables after exit from the parallel region.

```
nThreadPrivate = 0 (The last value in the master thread)
nPrivate = 4 (The value prior to entering parallel region)
nFirstPrivate = 4 (The value prior to entering parallel region)
nLastPrivate = 3 (The value from the last iteration of the loop)
nShared = 1 (The value assigned, from the delayed thread, 1)
```

## reduction

Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.

```
reduction(operation:var)
```

### Parameters

*operation*

The operator for the operation to do on the variables *var* at the end of the parallel region.

*var*

One or more variables on which to do scalar reduction. If more than one variable is specified, separate variable names with a comma.

## Remarks

`reduction` applies to the following directives:

- [parallel](#)
- [for](#)
- [sections](#)

For more information, see [2.7.2.6 reduction](#).

## Example

```
// omp_reduction.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

#define NUM_THREADS 4
#define SUM_START 1
#define SUM_END 10
#define FUNC_RETS {1, 1, 1, 1, 1}

int bRets[5] = FUNC_RETS;
int nSumCalc = ((SUM_START + SUM_END) * (SUM_END - SUM_START + 1)) / 2;

int func1( ) {return bRets[0];}
int func2( ) {return bRets[1];}
int func3( ) {return bRets[2];}
int func4( ) {return bRets[3];}
int func5( ) {return bRets[4];}

int main( )
{
    int nRet = 0,
        nCount = 0,
        nSum = 0,
        i,
        bSucceed = 1;

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel reduction(+ : nCount)
    {
        nCount += 1;

        #pragma omp for reduction(+ : nSum)
        for (i = SUM_START ; i <= SUM_END ; ++i)
            nSum += i;

        #pragma omp sections reduction(&& : bSucceed)
        {
            #pragma omp section
            {
                bSucceed = bSucceed && func1( );
            }

            #pragma omp section
            {
                bSucceed = bSucceed && func2( );
            }

            #pragma omp section
            {
```

```

        {
            bSucceed = bSucceed && func3( );
        }

        #pragma omp section
        {
            bSucceed = bSucceed && func4( );
        }

        #pragma omp section
        {
            bSucceed = bSucceed && func5( );
        }
    }
}

printf_s("The parallel section was executed %d times "
        "in parallel.\n", nCount);
printf_s("The sum of the consecutive integers from "
        "%d to %d, is %d\n", 1, 10, nSum);

if (bSucceed)
    printf_s("All of the functions, func1 through "
            "func5 succeeded!\n");
else
    printf_s("One or more of the functions, func1 "
            "through func5 failed!\n");

if (nCount != NUM_THREADS)
{
    printf_s("ERROR: For %d threads, %d were counted!\n",
            NUM_THREADS, nCount);
    nRet |= 0x1;
}

if (nSum != nSumCalc)
{
    printf_s("ERROR: The sum of %d through %d should be %d, "
            "but %d was reported!\n",
            SUM_START, SUM_END, nSumCalc, nSum);
    nRet |= 0x10;
}

if (bSucceed != (bRets[0] && bRets[1] &&
                bRets[2] && bRets[3] && bRets[4]))
{
    printf_s("ERROR: The sum of %d through %d should be %d, "
            "but %d was reported!\n",
            SUM_START, SUM_END, nSumCalc, nSum);
    nRet |= 0x100;
}
}

```

The parallel section was executed 4 times in parallel.  
The sum of the consecutive integers from 1 to 10, is 55  
All of the functions, func1 through func5 succeeded!

## schedule

Applies to the [for](#) directive.

```
schedule(type[,size])
```

## Parameters

*type*

The kind of scheduling, either `dynamic`, `guided`, `runtime`, or `static`.

*size*

(Optional) Specifies the size of iterations. *size* must be an integer. Not valid when *type* is `runtime`.

## Remarks

For more information, see [2.4.1 for construct](#).

## Example

```
// omp_schedule.cpp
// compile with: /openmp
#include <windows.h>
#include <stdio.h>
#include <omp.h>

#define NUM_THREADS 4
#define STATIC_CHUNK 5
#define DYNAMIC_CHUNK 5
#define NUM_LOOPS 20
#define SLEEP_EVERY_N 3

int main( )
{
    int nStatic1[NUM_LOOPS],
        nStaticN[NUM_LOOPS];
    int nDynamic1[NUM_LOOPS],
        nDynamicN[NUM_LOOPS];
    int nGuided[NUM_LOOPS];

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        #pragma omp for schedule(static, 1)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
                Sleep(0);
            nStatic1[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(static, STATIC_CHUNK)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
                Sleep(0);
            nStaticN[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(dynamic, 1)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
                Sleep(0);
            nDynamic1[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(dynamic, DYNAMIC_CHUNK)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == 0)
                Sleep(0);
            nDynamicN[i] = omp_get_thread_num( );
        }
    }
}
```

```

}

#pragma omp for schedule(guided)
for (int i = 0 ; i < NUM_LOOPS ; ++i)
{
    if ((i % SLEEP_EVERY_N) == 0)
        Sleep(0);
    nGuided[i] = omp_get_thread_num( );
}
}

printf_s("-----\n");
printf_s("| static | static | dynamic | dynamic | guided |\n");
printf_s("| 1 | %d | 1 | %d | |\n",
    STATIC_CHUNK, DYNAMIC_CHUNK);
printf_s("-----\n");

for (int i=0; i<NUM_LOOPS; ++i)
{
    printf_s("| %d | %d | %d | %d |"
        " %d |\n",
        nStatic1[i], nStaticN[i],
        nDynamic1[i], nDynamicN[i], nGuided[i]);
}

printf_s("-----\n");
}

```

```

-----
| static | static | dynamic | dynamic | guided |
| 1 | 5 | 1 | 5 | |
-----
| 0 | 0 | 0 | 2 | 1 |
| 1 | 0 | 3 | 2 | 1 |
| 2 | 0 | 3 | 2 | 1 |
| 3 | 0 | 3 | 2 | 1 |
| 0 | 0 | 2 | 2 | 1 |
| 1 | 1 | 2 | 3 | 3 |
| 2 | 1 | 2 | 3 | 3 |
| 3 | 1 | 0 | 3 | 3 |
| 0 | 1 | 0 | 3 | 3 |
| 1 | 1 | 0 | 3 | 2 |
| 2 | 2 | 1 | 0 | 2 |
| 3 | 2 | 1 | 0 | 2 |
| 0 | 2 | 1 | 0 | 3 |
| 1 | 2 | 2 | 0 | 3 |
| 2 | 2 | 2 | 0 | 0 |
| 3 | 3 | 2 | 1 | 0 |
| 0 | 3 | 3 | 1 | 1 |
| 1 | 3 | 3 | 1 | 1 |
| 2 | 3 | 3 | 1 | 1 |
| 3 | 3 | 0 | 1 | 3 |
-----

```

## shared

Specifies that one or more variables should be shared among all threads.

```
shared(var)
```

### Parameters

*var*

One or more variables to share. If more than one variable is specified, separate variable names with a comma.

### Remarks

Another way to share variables among threads is with the [copyprivate](#) clause.

`shared` applies to the following directives:

- [parallel](#)
- [for](#)
- [sections](#)

For more information, see [2.7.2.4 shared](#).

### Example

See [private](#) for an example of using `shared`.

# OpenMP Functions

4/22/2019 • 13 minutes to read • [Edit Online](#)

Provides links to functions used in the OpenMP API.

The Visual C++ implementation of the OpenMP standard includes the following functions and data types.

For environment execution:

FUNCTION	DESCRIPTION
<a href="#">omp_set_num_threads</a>	Sets the number of threads in upcoming parallel regions, unless overridden by a <a href="#">num_threads</a> clause.
<a href="#">omp_get_num_threads</a>	Returns the number of threads in the parallel region.
<a href="#">omp_get_max_threads</a>	Returns an integer that is equal to or greater than the number of threads that would be available if a parallel region without <a href="#">num_threads</a> were defined at that point in the code.
<a href="#">omp_get_thread_num</a>	Returns the thread number of the thread executing within its thread team.
<a href="#">omp_get_num_procs</a>	Returns the number of processors that are available when the function is called.
<a href="#">omp_in_parallel</a>	Returns nonzero if called from within a parallel region.
<a href="#">omp_set_dynamic</a>	Indicates that the number of threads available in upcoming parallel regions can be adjusted by the run time.
<a href="#">omp_get_dynamic</a>	Returns a value that indicates if the number of threads available in upcoming parallel regions can be adjusted by the run time.
<a href="#">omp_set_nested</a>	Enables nested parallelism.
<a href="#">omp_get_nested</a>	Returns a value that indicates if nested parallelism is enabled.

For lock:

FUNCTION	DESCRIPTION
<a href="#">omp_init_lock</a>	Initializes a simple lock.
<a href="#">omp_init_nest_lock</a>	Initializes a lock.
<a href="#">omp_destroy_lock</a>	Uninitializes a lock.
<a href="#">omp_destroy_nest_lock</a>	Uninitializes a nestable lock.

FUNCTION	DESCRIPTION
<a href="#">omp_set_lock</a>	Blocks thread execution until a lock is available.
<a href="#">omp_set_nest_lock</a>	Blocks thread execution until a lock is available.
<a href="#">omp_unset_lock</a>	Releases a lock.
<a href="#">omp_unset_nest_lock</a>	Releases a nestable lock.
<a href="#">omp_test_lock</a>	Attempts to set a lock but doesn't block thread execution.
<a href="#">omp_test_nest_lock</a>	Attempts to set a nestable lock but doesn't block thread execution.

DATA TYPE	DESCRIPTION
<code>omp_lock_t</code>	A type that holds the status of a lock, whether the lock is available or if a thread owns a lock.
<code>omp_nest_lock_t</code>	A type that holds one of the following pieces of information about a lock: whether the lock is available, and the identity of the thread that owns the lock and a nesting count.

For timing routines:

FUNCTION	DESCRIPTION
<a href="#">omp_get_wtime</a>	Returns a value in seconds of the time elapsed from some point.
<a href="#">omp_get_wtick</a>	Returns the number of seconds between processor clock ticks.

## omp\_destroy\_lock

Uninitializes a lock.

```
void omp_destroy_lock(
    omp_lock_t *lock
);
```

### Parameters

*lock*

A variable of type `omp_lock_t` that was initialized with [omp\\_init\\_lock](#).

### Remarks

For more information, see [3.2.2 omp\\_destroy\\_lock and omp\\_destroy\\_nest\\_lock functions](#).

### Example

See [omp\\_init\\_lock](#) for an example of using `omp_destroy_lock`.

## omp\_destroy\_nest\_lock

Uninitializes a nestable lock.

```
void omp_destroy_nest_lock(  
    omp_nest_lock_t *lock  
);
```

### Parameters

*lock*

A variable of type `omp_nest_lock_t` that was initialized with [omp\\_init\\_nest\\_lock](#).

### Remarks

For more information, see [3.2.2 omp\\_destroy\\_lock and omp\\_destroy\\_nest\\_lock functions](#).

### Example

See [omp\\_init\\_nest\\_lock](#) for an example of using `omp_destroy_nest_lock`.

## omp\_get\_dynamic

Returns a value that indicates if the number of threads available in upcoming parallel regions can be adjusted by the run time.

```
int omp_get_dynamic();
```

### Return value

A nonzero value means threads will be dynamically adjusted.

### Remarks

Dynamic adjustment of threads is specified with [omp\\_set\\_dynamic](#) and [OMP\\_DYNAMIC](#).

For more information, see [3.1.7 omp\\_set\\_dynamic function](#).

### Example

See [omp\\_set\\_dynamic](#) for an example of using `omp_get_dynamic`.

## omp\_get\_max\_threads

Returns an integer that is equal to or greater than the number of threads that would be available if a parallel region without [num\\_threads](#) were defined at that point in the code.

```
int omp_get_max_threads( )
```

### Remarks

For more information, see [3.1.3 omp\\_get\\_max\\_threads function](#).

### Example

```

// omp_get_max_threads.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main( )
{
    omp_set_num_threads(8);
    printf_s("%d\n", omp_get_max_threads( ));
    #pragma omp parallel
        #pragma omp master
        {
            printf_s("%d\n", omp_get_max_threads( ));
        }

    printf_s("%d\n", omp_get_max_threads( ));

    #pragma omp parallel num_threads(3)
        #pragma omp master
        {
            printf_s("%d\n", omp_get_max_threads( ));
        }

    printf_s("%d\n", omp_get_max_threads( ));
}

```

```

8
8
8
8
8

```

## omp\_get\_nested

Returns a value that indicates if nested parallelism is enabled.

```
int omp_get_nested( );
```

### Return value

A nonzero value means nested parallelism is enabled.

### Remarks

Nested parallelism is specified with [omp\\_set\\_nested](#) and [OMP\\_NESTED](#).

For more information, see [3.1.10 omp\\_get\\_nested function](#).

### Example

See [omp\\_set\\_nested](#) for an example of using `omp_get_nested`.

## omp\_get\_num\_procs

Returns the number of processors that are available when the function is called.

```
int omp_get_num_procs();
```

### Remarks

For more information, see [3.1.5 omp\\_get\\_num\\_procs function](#).

### Example

```
// omp_get_num_procs.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main( )
{
    printf_s("%d\n", omp_get_num_procs( ));
    #pragma omp parallel
        #pragma omp master
        {
            printf_s("%d\n", omp_get_num_procs( ));
        }
}
```

```
// Expect the following output when the example is run on a two-processor machine:
2
2
```

## omp\_get\_num\_threads

Returns the number of threads in the parallel region.

```
int omp_get_num_threads( );
```

### Remarks

For more information, see [3.1.2 omp\\_get\\_num\\_threads function](#).

### Example

```
// omp_get_num_threads.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main()
{
    omp_set_num_threads(4);
    printf_s("%d\n", omp_get_num_threads( ));
    #pragma omp parallel
        #pragma omp master
        {
            printf_s("%d\n", omp_get_num_threads( ));
        }

    printf_s("%d\n", omp_get_num_threads( ));

    #pragma omp parallel num_threads(3)
        #pragma omp master
        {
            printf_s("%d\n", omp_get_num_threads( ));
        }

    printf_s("%d\n", omp_get_num_threads( ));
}
```

```
1
4
1
3
1
```

## omp\_get\_thread\_num

Returns the thread number of the thread executing within its thread team.

```
int omp_get_thread_num( );
```

### Remarks

For more information, see [3.1.4 omp\\_get\\_thread\\_num function](#).

### Example

See [parallel](#) for an example of using `omp_get_thread_num`.

## omp\_get\_wtick

Returns the number of seconds between processor clock ticks.

```
double omp_get_wtick( );
```

### Remarks

For more information, see [3.3.2 omp\\_get\\_wtick function](#).

### Example

See [omp\\_get\\_wtime](#) for an example of using `omp_get_wtick`.

## omp\_get\_wtime

Returns a value in seconds of the time elapsed from some point.

```
double omp_get_wtime( );
```

### Return value

Returns a value in seconds of the time elapsed from some arbitrary, but consistent point.

### Remarks

That point will remain consistent during program execution, making upcoming comparisons possible.

For more information, see [3.3.1 omp\\_get\\_wtime function](#).

### Example

```

// omp_get_wtime.cpp
// compile with: /openmp
#include "omp.h"
#include <stdio.h>
#include <windows.h>

int main() {
    double start = omp_get_wtime( );
    Sleep(1000);
    double end = omp_get_wtime( );
    double wtick = omp_get_wtick( );

    printf_s("start = %.16g\nend = %.16g\ndiff = %.16g\n",
            start, end, end - start);

    printf_s("wtick = %.16g\n1/wtick = %.16g\n",
            wtick, 1.0 / wtick);
}

```

```

start = 594255.3671159324
end = 594256.3664474116
diff = 0.9993314791936427
wtick = 2.793651148400146e-007
1/wtick = 3579545

```

## omp\_in\_parallel

Returns nonzero if called from within a parallel region.

```
int omp_in_parallel( );
```

### Remarks

For more information, see [3.1.6 omp\\_in\\_parallel function](#).

### Example

```

// omp_in_parallel.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main( )
{
    omp_set_num_threads(4);
    printf_s("%d\n", omp_in_parallel( ));

    #pragma omp parallel
    #pragma omp master
    {
        printf_s("%d\n", omp_in_parallel( ));
    }
}

```

```

0
1

```

## omp\_init\_lock

Initializes a simple lock.

```
void omp_init_lock(  
    omp_lock_t *lock  
);
```

### Parameters

*lock*

A variable of type `omp_lock_t`.

### Remarks

For more information, see [3.2.1 omp\\_init\\_lock and omp\\_init\\_nest\\_lock functions](#).

### Example

```
// omp_init_lock.cpp  
// compile with: /openmp  
#include <stdio.h>  
#include <omp.h>  
  
omp_lock_t my_lock;  
  
int main() {  
    omp_init_lock(&my_lock);  
  
    #pragma omp parallel num_threads(4)  
    {  
        int tid = omp_get_thread_num( );  
        int i, j;  
  
        for (i = 0; i < 5; ++i) {  
            omp_set_lock(&my_lock);  
            printf_s("Thread %d - starting locked region\n", tid);  
            printf_s("Thread %d - ending locked region\n", tid);  
            omp_unset_lock(&my_lock);  
        }  
    }  
  
    omp_destroy_lock(&my_lock);  
}
```

```
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 0 - starting locked region
Thread 0 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 1 - starting locked region
Thread 1 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 2 - starting locked region
Thread 2 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
Thread 3 - starting locked region
Thread 3 - ending locked region
```

## omp\_init\_nest\_lock

Initializes a lock.

```
void omp_init_nest_lock(  
    omp_nest_lock_t *lock  
);
```

### Parameters

*lock*

A variable of type `omp_nest_lock_t`.

### Remarks

The initial nesting count is zero.

For more information, see [3.2.1 omp\\_init\\_lock and omp\\_init\\_nest\\_lock functions](#).

### Example

```

// omp_init_nest_lock.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

omp_nest_lock_t my_lock;

void Test() {
    int tid = omp_get_thread_num( );
    omp_set_nest_lock(&my_lock);
    printf_s("Thread %d - starting nested locked region\n", tid);
    printf_s("Thread %d - ending nested locked region\n", tid);
    omp_unset_nest_lock(&my_lock);
}

int main() {
    omp_init_nest_lock(&my_lock);

    #pragma omp parallel num_threads(4)
    {
        int i, j;

        for (i = 0; i < 5; ++i) {
            omp_set_nest_lock(&my_lock);
            if (i % 3)
                Test();
            omp_unset_nest_lock(&my_lock);
        }
    }

    omp_destroy_nest_lock(&my_lock);
}

```

```

Thread 0 - starting nested locked region
Thread 0 - ending nested locked region
Thread 0 - starting nested locked region
Thread 0 - ending nested locked region
Thread 3 - starting nested locked region
Thread 3 - ending nested locked region
Thread 3 - starting nested locked region
Thread 3 - ending nested locked region
Thread 3 - starting nested locked region
Thread 3 - ending nested locked region
Thread 2 - starting nested locked region
Thread 2 - ending nested locked region
Thread 2 - starting nested locked region
Thread 2 - ending nested locked region
Thread 2 - starting nested locked region
Thread 2 - ending nested locked region
Thread 1 - starting nested locked region
Thread 1 - ending nested locked region
Thread 1 - starting nested locked region
Thread 1 - ending nested locked region
Thread 1 - starting nested locked region
Thread 1 - ending nested locked region
Thread 0 - starting nested locked region
Thread 0 - ending nested locked region

```

## omp\_set\_dynamic

Indicates that the number of threads available in upcoming parallel regions can be adjusted by the run time.

```
void omp_set_dynamic(  
    int val  
);
```

### Parameters

*val*

A value that indicates if the number of threads available in upcoming parallel regions can be adjusted by the runtime. If nonzero, the runtime can adjust the number of threads, if zero, the runtime won't dynamically adjust the number of threads.

### Remarks

The number of threads will never exceed the value set by [omp\\_set\\_num\\_threads](#) or by `OMP_NUM_THREADS`.

Use [omp\\_get\\_dynamic](#) to display the current setting of `omp_set_dynamic`.

The setting for `omp_set_dynamic` will override the setting of the `OMP_DYNAMIC` environment variable.

For more information, see [3.1.7 omp\\_set\\_dynamic function](#).

### Example

```
// omp_set_dynamic.cpp  
// compile with: /openmp  
#include <stdio.h>  
#include <omp.h>  
  
int main()  
{  
    omp_set_dynamic(9);  
    omp_set_num_threads(4);  
    printf_s("%d\n", omp_get_dynamic( ));  
    #pragma omp parallel  
        #pragma omp master  
        {  
            printf_s("%d\n", omp_get_dynamic( ));  
        }  
}
```

```
1  
1
```

## omp\_set\_lock

Blocks thread execution until a lock is available.

```
void omp_set_lock(  
    omp_lock_t *lock  
);
```

### Parameters

*lock*

A variable of type `omp_lock_t` that was initialized with [omp\\_init\\_lock](#).

### Remarks

For more information, see [3.2.3 omp\\_set\\_lock and omp\\_set\\_nest\\_lock functions](#).

## Examples

See [omp\\_init\\_lock](#) for an example of using `omp_set_lock`.

# omp\_set\_nest\_lock

Blocks thread execution until a lock is available.

```
void omp_set_nest_lock(  
    omp_nest_lock_t *lock  
);
```

## Parameters

*lock*

A variable of type `omp_nest_lock_t` that was initialized with [omp\\_init\\_nest\\_lock](#).

## Remarks

For more information, see [3.2.3 omp\\_set\\_lock and omp\\_set\\_nest\\_lock functions](#).

## Examples

See [omp\\_init\\_nest\\_lock](#) for an example of using `omp_set_nest_lock`.

# omp\_set\_nested

Enables nested parallelism.

```
void omp_set_nested(  
    int val  
);
```

## Parameters

*val*

A nonzero value enables nested parallelism, while zero disables nested parallelism.

## Remarks

OMP nested parallelism can be turned on with `omp_set_nested`, or by setting the `OMP_NESTED` environment variable.

The setting for `omp_set_nested` will override the setting of the `OMP_NESTED` environment variable.

Enabling the environment variable can break an otherwise operational program, because the number of threads increases exponentially when nesting parallel regions. For example, a function that recurses six times with the number of OMP threads set to 4 requires 4,096 (4 to the power of 6) threads. Except with I/O-bound applications, the performance of an application generally degrades if there are more threads than processors.

Use [omp\\_get\\_nested](#) to display the current setting of `omp_set_nested`.

For more information, see [3.1.9 omp\\_set\\_nested function](#).

## Example

```
// omp_set_nested.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

int main( )
{
    omp_set_nested(1);
    omp_set_num_threads(4);
    printf_s("%d\n", omp_get_nested( ));
    #pragma omp parallel
        #pragma omp master
            {
                printf_s("%d\n", omp_get_nested( ));
            }
}
```

```
1
1
```

## omp\_set\_num\_threads

Sets the number of threads in upcoming parallel regions, unless overridden by a [num\\_threads](#) clause.

```
void omp_set_num_threads(
    int num_threads
);
```

### Parameters

*num\_threads*

The number of threads in the parallel region.

### Remarks

For more information, see [3.1.1 omp\\_set\\_num\\_threads function](#).

### Example

See [omp\\_get\\_num\\_threads](#) for an example of using `omp_set_num_threads`.

## omp\_test\_lock

Attempts to set a lock but doesn't block thread execution.

```
int omp_test_lock(
    omp_lock_t *lock
);
```

### Parameters

*lock*

A variable of type `omp_lock_t` that was initialized with [omp\\_init\\_lock](#).

### Remarks

For more information, see [3.2.5 omp\\_test\\_lock and omp\\_test\\_nest\\_lock functions](#).

### Example

```

// omp_test_lock.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

omp_lock_t simple_lock;

int main() {
    omp_init_lock(&simple_lock);

    #pragma omp parallel num_threads(4)
    {
        int tid = omp_get_thread_num();

        while (!omp_test_lock(&simple_lock))
            printf_s("Thread %d - failed to acquire simple_lock\n",
                    tid);

        printf_s("Thread %d - acquired simple_lock\n", tid);

        printf_s("Thread %d - released simple_lock\n", tid);
        omp_unset_lock(&simple_lock);
    }

    omp_destroy_lock(&simple_lock);
}

```

```

Thread 1 - acquired simple_lock
Thread 1 - released simple_lock
Thread 0 - failed to acquire simple_lock
Thread 3 - failed to acquire simple_lock
Thread 0 - failed to acquire simple_lock
Thread 3 - failed to acquire simple_lock
Thread 2 - acquired simple_lock
Thread 0 - failed to acquire simple_lock
Thread 3 - failed to acquire simple_lock
Thread 0 - failed to acquire simple_lock
Thread 3 - failed to acquire simple_lock
Thread 2 - released simple_lock
Thread 0 - failed to acquire simple_lock
Thread 3 - failed to acquire simple_lock
Thread 0 - acquired simple_lock
Thread 3 - failed to acquire simple_lock
Thread 0 - released simple_lock
Thread 3 - failed to acquire simple_lock
Thread 3 - acquired simple_lock
Thread 3 - released simple_lock

```

## omp\_test\_nest\_lock

Attempts to set a nestable lock but doesn't block thread execution.

```

int omp_test_nest_lock(
    omp_nest_lock_t *lock
);

```

### Parameters

*lock*

A variable of type `omp_nest_lock_t` that was initialized with `omp_init_nest_lock`.

### Remarks

For more information, see [3.2.5 omp\\_test\\_lock and omp\\_test\\_nest\\_lock functions](#).

## Example

```
// omp_test_nest_lock.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>

omp_nest_lock_t nestable_lock;

int main() {
    omp_init_nest_lock(&nestable_lock);

    #pragma omp parallel num_threads(4)
    {
        int tid = omp_get_thread_num();
        while (!omp_test_nest_lock(&nestable_lock))
            printf_s("Thread %d - failed to acquire nestable_lock\n",
                    tid);

        printf_s("Thread %d - acquired nestable_lock\n", tid);

        if (omp_test_nest_lock(&nestable_lock)) {
            printf_s("Thread %d - acquired nestable_lock again\n",
                    tid);
            printf_s("Thread %d - released nestable_lock\n",
                    tid);
            omp_unset_nest_lock(&nestable_lock);
        }

        printf_s("Thread %d - released nestable_lock\n", tid);
        omp_unset_nest_lock(&nestable_lock);
    }

    omp_destroy_nest_lock(&nestable_lock);
}
```

```
Thread 1 - acquired nestable_lock
Thread 0 - failed to acquire nestable_lock
Thread 1 - acquired nestable_lock again
Thread 0 - failed to acquire nestable_lock
Thread 1 - released nestable_lock
Thread 0 - failed to acquire nestable_lock
Thread 1 - released nestable_lock
Thread 0 - failed to acquire nestable_lock
Thread 3 - acquired nestable_lock
Thread 0 - failed to acquire nestable_lock
Thread 3 - acquired nestable_lock again
Thread 0 - failed to acquire nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 3 - released nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 3 - released nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 0 - acquired nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 0 - acquired nestable_lock again
Thread 2 - failed to acquire nestable_lock
Thread 0 - released nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 0 - released nestable_lock
Thread 2 - failed to acquire nestable_lock
Thread 2 - acquired nestable_lock
Thread 2 - acquired nestable_lock again
Thread 2 - released nestable_lock
Thread 2 - released nestable_lock
```

## omp\_unset\_lock

Releases a lock.

```
void omp_unset_lock(
    omp_lock_t *lock
);
```

### Parameters

*lock*

A variable of type `omp_lock_t` that was initialized with [omp\\_init\\_lock](#), owned by the thread and executing in the function.

### Remarks

For more information, see [3.2.4 omp\\_unset\\_lock and omp\\_unset\\_nest\\_lock functions](#).

### Example

See [omp\\_init\\_lock](#) for an example of using `omp_unset_lock`.

## omp\_unset\_nest\_lock

Releases a nestable lock.

```
void omp_unset_nest_lock(
    omp_nest_lock_t *lock
);
```

**Parameters**

*lock*

A variable of type `omp_nest_lock_t` that was initialized with `omp_init_nest_lock`, owned by the thread and executing in the function.

**Remarks**

For more information, see [3.2.4 `omp\_unset\_lock` and `omp\_unset\_nest\_lock` functions](#).

**Example**

See [`omp\_init\_nest\_lock`](#) for an example of using `omp_unset_nest_lock`.

# OpenMP Environment Variables

4/22/2019 • 2 minutes to read • [Edit Online](#)

Provides links to environment variables used in the OpenMP API.

The Visual C++ implementation of the OpenMP standard includes the following environment variables. These environment variables are read at program startup and modifications to their values are ignored at runtime (for example, using `_putenv`, `_wputenv`).

ENVIRONMENT VARIABLE	DESCRIPTION
<a href="#">OMP_SCHEDULE</a>	Modifies the behavior of the <code>schedule</code> clause when <code>schedule(runtime)</code> is specified in a <code>for</code> or <code>parallel for</code> directive.
<a href="#">OMP_NUM_THREADS</a>	Sets the maximum number of threads in the parallel region, unless overridden by <code>omp_set_num_threads</code> or <code>num_threads</code> .
<a href="#">OMP_DYNAMIC</a>	Specifies whether the OpenMP run time can adjust the number of threads in a parallel region.
<a href="#">OMP_NESTED</a>	Specifies whether nested parallelism is enabled, unless nested parallelism is enabled or disabled with <code>omp_set_nested</code> .

## OMP\_DYNAMIC

Specifies whether the OpenMP run time can adjust the number of threads in a parallel region.

```
set OMP_DYNAMIC[=TRUE | =FALSE]
```

### Remarks

The `OMP_DYNAMIC` environment variable can be overridden by the `omp_set_dynamic` function.

The default value in the Visual C++ implementation of the OpenMP standard is `OMP_DYNAMIC=FALSE`.

For more information, see [4.3 OMP\\_DYNAMIC](#).

### Example

The following command sets the `OMP_DYNAMIC` environment variable to TRUE:

```
set OMP_DYNAMIC=TRUE
```

The following command displays the current setting of the `OMP_DYNAMIC` environment variable:

```
set OMP_DYNAMIC
```

## OMP\_NESTED

Specifies whether nested parallelism is enabled, unless nested parallelism is enabled or disabled with

```
omp_set_nested .
```

```
set OMP_NESTED[=TRUE | =FALSE]
```

### Remarks

The `OMP_NESTED` environment variable can be overridden by the `omp_set_nested` function.

The default value in the Visual C++ implementation of the OpenMP standard is `OMP_DYNAMIC=FALSE`.

For more information, see [4.4 OMP\\_NESTED](#).

### Example

The following command sets the `OMP_NESTED` environment variable to TRUE:

```
set OMP_NESTED=TRUE
```

The following command displays the current setting of the `OMP_NESTED` environment variable:

```
set OMP_NESTED
```

## OMP\_NUM\_THREADS

Sets the maximum number of threads in the parallel region, unless overridden by `omp_set_num_threads` or `num_threads`.

```
set OMP_NUM_THREADS[=num]
```

### Parameters

*num*

The maximum number of threads you want in the parallel region, up to 64 in the Visual C++ implementation.

### Remarks

The `OMP_NUM_THREADS` environment variable can be overridden by the `omp_set_num_threads` function or by `num_threads`.

The default value of `num` in the Visual C++ implementation of the OpenMP standard is the number of virtual processors, including hyperthreading CPUs.

For more information, see [4.2 OMP\\_NUM\\_THREADS](#).

### Example

The following command sets the `OMP_NUM_THREADS` environment variable to `16`:

```
set OMP_NUM_THREADS=16
```

The following command displays the current setting of the `OMP_NUM_THREADS` environment variable:

```
set OMP_NUM_THREADS
```

## OMP\_SCHEDULE

Modifies the behavior of the [schedule](#) clause when `schedule(runtime)` is specified in a `for` or `parallel for` directive.

```
set OMP_SCHEDULE[=type[,size]]
```

### Parameters

*size*

(Optional) Specifies the size of iterations. *size* must be a positive integer. The default is `1`, except when *type* is `static`. Not valid when *type* is `runtime`.

*type*

The kind of scheduling, either `dynamic`, `guided`, `runtime`, or `static`.

### Remarks

The default value in the Visual C++ implementation of the OpenMP standard is `OMP_SCHEDULE=static,0`.

For more information, see [4.1 OMP\\_SCHEDULE](#).

### Example

The following command sets the `OMP_SCHEDULE` environment variable:

```
set OMP_SCHEDULE="guided,2"
```

The following command displays the current setting of the `OMP_SCHEDULE` environment variable:

```
set OMP_SCHEDULE
```

# Multithreading Support for Older Code (Visual C++)

10/31/2018 • 2 minutes to read • [Edit Online](#)

Visual C++ allows you to have multiple concurrent threads of execution running simultaneously. With multithreading, you can spin off background tasks, manage simultaneous streams of input, manage a user interface, and much more.

## In This Section

### [Multithreading with C and Win32](#)

Provides support for creating multithread applications with Microsoft Windows

### [Multithreading with C++ and MFC](#)

Describes what processes and threads are and what the MFC approach to multithreading is.

### [Multithreading and Locales](#)

Discusses issues that arise when using the locale functionality of both the C Runtime Library and the C++ Standard Library in a multithreaded application.

## Related Sections

### [CWinThread](#)

Represents a thread of execution within an application.

### [CSyncObject](#)

Describes a pure virtual class that provides functionality common to the synchronization objects in Win32.

### [CSemaphore](#)

Represents a semaphore, which is a synchronization object that allows a limited number of threads in one or more processes to access a resource.

### [CMutex](#)

Represents a mutex, which is a synchronization object that allows one thread mutually exclusive access to a resource.

### [CCriticalSection](#)

Represents a critical section, which is a synchronization object that allows one thread at a time to access a resource or section of code.

### [CEvent](#)

Represents an event, which is a synchronization object that allows one thread to notify another that an event has occurred.

### [CMultiLock](#)

Represents the access-control mechanism used in controlling access to resources in a multithreaded program.

### [CSingleLock](#)

Represents the access-control mechanism used in controlling access to a resource in a multithreaded program.

# Multithreading with C and Win32

3/4/2019 • 2 minutes to read • [Edit Online](#)

Microsoft Visual C++ provides support for creating multithread applications. You should consider using more than one thread if your application needs to perform expensive operations that would cause the user-interface to become unresponsive.

With Visual C++, there are two ways to program with multiple threads: use the Microsoft Foundation Class (MFC) library or the C run-time library and the Win32 API. For information about creating multithread applications with MFC, see [Multithreading with C++ and MFC](#) after reading the following topics about multithreading in C.

These topics explain the features in Visual C++ that support the creation of multithread programs.

## What do you want to know more about?

- [What multithreading is about](#)
- [Library support for multithreading](#)
- [Include files for multithreading](#)
- [C Run-Time library functions for thread control](#)
- [Sample multithread program in C](#)
- [Writing a Multithread Win32 Program](#)
- [Compiling and linking multithread programs](#)
- [Avoiding problem areas with multithread programs](#)
- [Thread local storage \(TLS\)](#)

## See also

[Multithreading Support for Older Code \(Visual C++\)](#)

# Multithread Programs

3/4/2019 • 2 minutes to read • [Edit Online](#)

A thread is basically a path of execution through a program. It is also the smallest unit of execution that Win32 schedules. A thread consists of a stack, the state of the CPU registers, and an entry in the execution list of the system scheduler. Each thread shares all the process's resources.

A process consists of one or more threads and the code, data, and other resources of a program in memory. Typical program resources are open files, semaphores, and dynamically allocated memory. A program executes when the system scheduler gives one of its threads execution control. The scheduler determines which threads should run and when they should run. Threads of lower priority might have to wait while higher priority threads complete their tasks. On multiprocessor machines, the scheduler can move individual threads to different processors to balance the CPU load.

Each thread in a process operates independently. Unless you make them visible to each other, the threads execute individually and are unaware of the other threads in a process. Threads sharing common resources, however, must coordinate their work by using semaphores or another method of interprocess communication. For more information about synchronizing threads, see [Writing a Multithreaded Win32 Program](#).

## See also

[Multithreading with C and Win32](#)

# Library Support for Multithreading

3/4/2019 • 2 minutes to read • [Edit Online](#)

All versions of the CRT now support multi threading with the exception of the non-locking versions of some functions. See [Multithreaded Libraries Performance](#) for more information.

See [CRT Library Features](#) for more information on CRT versions.

## See also

[Multithreading with C and Win32](#)

# Include Files for Multithreading

3/4/2019 • 2 minutes to read • [Edit Online](#)

Standard include files declare C run-time library functions as they are implemented in the libraries. If you use the [Full Optimization \(/Ox\)](#) or [fastcall Calling Convention \(/Gr\)](#) option, the compiler assumes that all functions should be called using the register calling convention. The run-time library functions were compiled using the C calling convention, and the declarations in the standard include files tell the compiler to generate correct external references to these functions.

## See also

[Multithreading with C and Win32](#)

# C Run-Time Library Functions for Thread Control

3/4/2019 • 2 minutes to read • [Edit Online](#)

All Win32 programs have at least one thread. Any thread can create additional threads. A thread can complete its work quickly and then terminate, or it can stay active for the life of the program.

The LIBCMT and MSVCRT C run-time libraries provide the following functions for thread creation and termination: `_beginthread`, `_beginthreadex` and `_endthread`, `_endthreadex`.

The `_beginthread` and `_beginthreadex` functions create a new thread and return a thread identifier if the operation is successful. The thread terminates automatically if it completes execution, or it can terminate itself with a call to `_endthread` or `_endthreadex`.

## NOTE

If you are going to call C run-time routines from a program built with `Libcmt.lib`, you must start your threads with the `_beginthread` or `_beginthreadex` function. Do not use the Win32 functions `ExitThread` and `CreateThread`. Using `SuspendThread` can lead to a deadlock when more than one thread is blocked waiting for the suspended thread to complete its access to a C run-time data structure.

## The `_beginthread` and `_beginthreadex` Functions

The `_beginthread` and `_beginthreadex` functions create a new thread. A thread shares the code and data segments of a process with other threads in the process but has its own unique register values, stack space, and current instruction address. The system gives CPU time to each thread, so that all threads in a process can execute concurrently.

`_beginthread` and `_beginthreadex` are similar to the `CreateThread` function in the Win32 API but has these differences:

- They initialize certain C run-time library variables. This is important only if you use the C run-time library in your threads.
- `CreateThread` helps provide control over security attributes. You can use this function to start a thread in a suspended state.

`_beginthread` and `_beginthreadex` return a handle to the new thread if successful or an error code if there was an error.

## The `_endthread` and `_endthreadex` Functions

The `_endthread` function terminates a thread created by `_beginthread` (and similarly, `_endthreadex` terminates a thread created by `_beginthreadex`). Threads terminate automatically when they finish. `_endthread` and `_endthreadex` are useful for conditional termination from within a thread. A thread dedicated to communications processing, for example, can quit if it is unable to get control of the communications port.

## See also

[Multithreading with C and Win32](#)

# Sample Multithread C Program

3/4/2019 • 3 minutes to read • [Edit Online](#)

Bounce.c is a sample multithread program that creates a new thread each time the letter `a` or `A` is typed. Each thread bounces a happy face of a different color around the screen. Up to 32 threads can be created. The program's normal termination occurs when `q` or `Q` is typed. For information about compiling and linking Bounce.c, see [Compiling and Linking Multithread Programs](#).

## Example

### Code

```
// sample_multithread_c_program.c
// compile with: /c
//
// Bounce - Creates a new thread each time the letter 'a' is typed.
// Each thread bounces a happy face of a different color around
// the screen. All threads are terminated when the letter 'Q' is
// entered.
//
#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>

#define MAX_THREADS 32

// The function getrandom returns a random number between
// min and max, which must be in integer range.
#define getrandom( min, max ) (SHORT)((rand() % (int)((max) + 1) - \
    (min))) + (min))

int main( void );           // Thread 1: main
void KbdFunc( void );      // Keyboard input, thread dispatch
void BounceProc( void * MyID ); // Threads 2 to n: display
void ClearScreen( void ); // Screen clear
void ShutDown( void );    // Program shutdown
void WriteTitle( int ThreadNum ); // Display title bar information

HANDLE hConsoleOut;       // Handle to the console
HANDLE hRunMutex;         // "Keep Running" mutex
HANDLE hScreenMutex;     // "Screen update" mutex
int ThreadNr;            // Number of threads started
CONSOLE_SCREEN_BUFFER_INFO csbiInfo; // Console information

int main() // Thread One
{
    // Get display screen information & clear the screen.
    hConsoleOut = GetStdHandle( STD_OUTPUT_HANDLE );
    GetConsoleScreenBufferInfo( hConsoleOut, &csbiInfo );
    ClearScreen();
    WriteTitle( 0 );

    // Create the mutexes and reset thread count.
    hScreenMutex = CreateMutex( NULL, FALSE, NULL ); // Cleared
    hRunMutex = CreateMutex( NULL, TRUE, NULL );    // Set
    ThreadNr = 0;
}
```

```

// Start waiting for keyboard input to dispatch threads or exit.
KbdFunc();

// All threads done. Clean up handles.
CloseHandle( hScreenMutex );
CloseHandle( hRunMutex );
CloseHandle( hConsoleOut );
}

void ShutDown( void ) // Shut down threads
{
    while ( ThreadNr > 0 )
    {
        // Tell thread to die and record its death.
        ReleaseMutex( hRunMutex );
        ThreadNr--;
    }

    // Clean up display when done
    WaitForSingleObject( hScreenMutex, INFINITE );
    ClearScreen();
}

void KbdFunc( void ) // Dispatch and count threads.
{
    int      KeyInfo;

    do
    {
        KeyInfo = _getch();
        if ( tolower( KeyInfo ) == 'a' &&
            ThreadNr < MAX_THREADS )
        {
            ThreadNr++;
            _beginthread( BounceProc, 0, &ThreadNr );
            WriteTitle( ThreadNr );
        }
    } while( tolower( KeyInfo ) != 'q' );

    ShutDown();
}

void BounceProc( void *pMyID )
{
    char      MyCell, OldCell;
    WORD      MyAttrib, OldAttrib;
    char      BlankCell = 0x20;
    COORD      Coords, Delta;
    COORD      Old = {0,0};
    DWORD      Dummy;
    char      *MyID = (char*)pMyID;

    // Generate update increments and initial
    // display coordinates.
    srand( (unsigned int) *MyID * 3 );

    Coords.X = getrandom( 0, csbiInfo.dwSize.X - 1 );
    Coords.Y = getrandom( 0, csbiInfo.dwSize.Y - 1 );
    Delta.X = getrandom( -3, 3 );
    Delta.Y = getrandom( -3, 3 );

    // Set up "happy face" & generate color
    // attribute from thread number.
    if( *MyID > 16)
        MyCell = 0x01;          // outline face
    else
        MyCell = 0x02;          // solid face
    MyAttrib = *MyID & 0x0F;    // force black background

```

```

do
{
    // Wait for display to be available, then lock it.
    WaitForSingleObject( hScreenMutex, INFINITE );

    // If we still occupy the old screen position, blank it out.
    ReadConsoleOutputCharacter( hConsoleOut, &OldCell, 1,
                                Old, &Dummy );
    ReadConsoleOutputAttribute( hConsoleOut, &OldAttrib, 1,
                                Old, &Dummy );
    if ( ( OldCell == MyCell ) && ( OldAttrib == MyAttrib ) )
        WriteConsoleOutputCharacter( hConsoleOut, &BlankCell, 1,
                                      Old, &Dummy );

    // Draw new face, then clear screen lock
    WriteConsoleOutputCharacter( hConsoleOut, &MyCell, 1,
                                Coords, &Dummy );
    WriteConsoleOutputAttribute( hConsoleOut, &MyAttrib, 1,
                                Coords, &Dummy );
    ReleaseMutex( hScreenMutex );

    // Increment the coordinates for next placement of the block.
    Old.X = Coords.X;
    Old.Y = Coords.Y;
    Coords.X += Delta.X;
    Coords.Y += Delta.Y;

    // If we are about to go off the screen, reverse direction
    if( Coords.X < 0 || Coords.X >= csbiInfo.dwSize.X )
    {
        Delta.X = -Delta.X;
        Beep( 400, 50 );
    }
    if( Coords.Y < 0 || Coords.Y > csbiInfo.dwSize.Y )
    {
        Delta.Y = -Delta.Y;
        Beep( 600, 50 );
    }
}
// Repeat while RunMutex is still taken.
while ( WaitForSingleObject( hRunMutex, 75L ) == WAIT_TIMEOUT );
}

void WriteTitle( int ThreadNum )
{
    enum {
        sizeofNThreadMsg = 80
    };
    char    NThreadMsg[sizeofNThreadMsg];

    sprintf_s( NThreadMsg, sizeofNThreadMsg,
               "Threads running: %02d. Press 'A' "
               "to start a thread, 'Q' to quit.", ThreadNum );
    SetConsoleTitle( NThreadMsg );
}

void ClearScreen( void )
{
    DWORD    dummy;
    COORD    Home = { 0, 0 };
    FillConsoleOutputCharacter( hConsoleOut, ' ',
                                csbiInfo.dwSize.X * csbiInfo.dwSize.Y,
                                Home, &dummy );
}

```

## Input

a  
q

## See also

[Multithreading with C and Win32](#)

# Writing a Multithreaded Win32 Program

3/4/2019 • 4 minutes to read • [Edit Online](#)

When you write a program with multiple threads, you must coordinate their behavior and [use of the program's resources](#). You must also make sure that each thread receives [its own stack](#).

## Sharing Common Resources Between Threads

### NOTE

For a similar discussion from the MFC point of view, see [Multithreading: Programming Tips](#) and [Multithreading: When to Use the Synchronization Classes](#).

Each thread has its own stack and its own copy of the CPU registers. Other resources, such as files, static data, and heap memory, are shared by all threads in the process. Threads using these common resources must be synchronized. Win32 provides several ways to synchronize resources, including semaphores, critical sections, events, and mutexes.

When multiple threads are accessing static data, your program must provide for possible resource conflicts. Consider a program where one thread updates a static data structure containing  $x,y$  coordinates for items to be displayed by another thread. If the update thread alters the  $x$  coordinate and is preempted before it can change the  $y$  coordinate, the display thread might be scheduled before the  $y$  coordinate is updated. The item would be displayed at the wrong location. You can avoid this problem by using semaphores to control access to the structure.

A mutex (short for *mutual exclusion*) is a way of communicating among threads or processes that are executing asynchronously of one another. This communication is usually used to coordinate the activities of multiple threads or processes, typically by controlling access to a shared resource by locking and unlocking the resource. To solve this  $x,y$  coordinate update problem, the update thread sets a mutex indicating that the data structure is in use before performing the update. It would clear the mutex after both coordinates had been processed. The display thread must wait for the mutex to be clear before updating the display. This process of waiting for a mutex is often called blocking on a mutex because the process is blocked and cannot continue until the mutex clears.

The Bounce.c program shown in [Sample Multithread C Program](#) uses a mutex named `ScreenMutex` to coordinate screen updates. Each time one of the display threads is ready to write to the screen, it calls `WaitForSingleObject` with the handle to `ScreenMutex` and constant `INFINITE` to indicate that the `WaitForSingleObject` call should block on the mutex and not time out. If `ScreenMutex` is clear, the wait function sets the mutex so other threads cannot interfere with the display and continues executing the thread. Otherwise, the thread blocks until the mutex clears. When the thread completes the display update, it releases the mutex by calling `ReleaseMutex`.

Screen displays and static data are only two of the resources requiring careful management. For example, your program might have multiple threads accessing the same file. Because another thread might have moved the file pointer, each thread must reset the file pointer before reading or writing. In addition, each thread must make sure that it is not preempted between the time it positions the pointer and the time it accesses the file. These threads should use a semaphore to coordinate access to the file by bracketing each file access with `WaitForSingleObject` and `ReleaseMutex` calls. The following code example illustrates this technique:

```
HANDLE hIOMutex= CreateMutex (NULL, FALSE, NULL);

WaitForSingleObject( hIOMutex, INFINITE );
fseek( fp, desired_position, 0L );
fwrite( data, sizeof( data ), 1, fp );
ReleaseMutex( hIOMutex);
```

## Thread Stacks

All of an application's default stack space is allocated to the first thread of execution, which is known as thread 1. As a result, you must specify how much memory to allocate for a separate stack for each additional thread your program needs. The operating system allocates additional stack space for the thread, if necessary, but you must specify a default value.

The first argument in the `_beginthread` call is a pointer to the `BounceProc` function, which executes the threads. The second argument specifies the default stack size for the thread. The last argument is an ID number that is passed to `BounceProc`. `BounceProc` uses the ID number to seed the random number generator and to select the thread's color attribute and display character.

Threads that make calls to the C run-time library or to the Win32 API must allow sufficient stack space for the library and API functions they call. The C `printf` function requires more than 500 bytes of stack space, and you should have 2K of stack space available when calling Win32 API routines.

Because each thread has its own stack, you can avoid potential collisions over data items by using as little static data as possible. Design your program to use automatic stack variables for all data that can be private to a thread. The only global variables in the Bounce.c program are either mutexes or variables that never change after they are initialized.

Win32 also provides Thread-Local Storage (TLS) to store per-thread data. For more information, see [Thread Local Storage \(TLS\)](#).

## See also

[Multithreading with C and Win32](#)

# Compiling and Linking Multithread Programs

3/4/2019 • 2 minutes to read • [Edit Online](#)

The Bounce.c program is introduced in [Sample Multithread C Program](#).

Programs are compiled multithreaded by default.

## To compile and link the multithread program Bounce.c from within the development environment

1. On the **File** menu, click **New**, and then click **Project**.
2. In the **Project Types** pane, click **Win32**.
3. In the **Templates** pane, click **Win32 Console Application**, and then name the project.
4. Add the file containing the C source code to the project.
5. On the **Build** menu, build the project by clicking the **Build** command.

## To compile and link the multithread program Bounce.c from the command line

1. Compile and link the program:

```
CL BOUNCE.C
```

## See also

[Multithreading with C and Win32](#)

# Avoiding Problem Areas with Multithread Programs

3/4/2019 • 2 minutes to read • [Edit Online](#)

There are several problems you might encounter in creating, linking, or executing a multithread C program. Some of the more common problems are described in the following table. (For a similar discussion from the MFC point of view, see [Multithreading: Programming Tips](#).)

PROBLEM	PROBABLE CAUSE
You get a message box showing that your program caused a protection violation.	<p>Many Win32 programming errors cause protection violations. A common cause of protection violations is the indirect assignment of data to null pointers. Because this results in your program trying to access memory that does not belong to it, a protection violation is issued.</p> <p>An easy way to detect the cause of a protection violation is to compile your program with debugging information and then run it through the debugger in the Visual C++ environment. When the protection fault occurs, Windows transfers control to the debugger and the cursor is positioned on the line that caused the problem.</p>
Your program generates numerous compile and link errors.	You can eliminate many potential problems by setting the compiler's warning level to one of its highest values and heeding the warning messages. By using the level 3 or level 4 warning level options, you can detect unintentional data conversions, missing function prototypes, and use of non-ANSI features.

## See also

[Multithreading with C and Win32](#)

# Thread Local Storage (TLS)

5/8/2019 • 4 minutes to read • [Edit Online](#)

Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process can allocate locations in which to store thread-specific data. Dynamically bound (run-time) thread-specific data is supported by way of the TLS API ([TlsAlloc](#). Win32 and the Microsoft C++ compiler now support statically bound (load-time) per-thread data in addition to the existing API implementation.

## Compiler Implementation for TLS

**C++11:** The `thread_local` storage class specifier is the recommended way to specify thread-local storage for objects and class members. For more information, see [Storage classes \(C++\)](#).

Visual C++ also provides a Microsoft-specific attribute, `thread`, as extended storage class modifier. Use the `__declspec` keyword to declare a `thread` variable. For example, the following code declares an integer thread local variable and initializes it with a value:

```
__declspec( thread ) int tls_i = 1;
```

## Rules and limitations

The following guidelines must be observed when declaring statically bound thread local objects and variables. These guidelines apply both to `thread` and for the most part also to `thread_local`:

- The `thread` attribute can be applied only to class and data declarations and definitions. It cannot be used on function declarations or definitions. For example, the following code generates a compiler error:

```
__declspec( thread ) void func(); // This will generate an error.
```

- The `thread` modifier might be specified only on data items with `static` extent. This includes global data objects (both `static` and `extern`), local static objects, and static data members of C++ classes. Automatic data objects cannot be declared with the `thread` attribute. The following code generates compiler errors:

```
void func1()
{
    __declspec( thread ) int tls_i; // This will generate an error.
}

int func2(__declspec( thread ) int tls_i) // This will generate an error.
{
    return tls_i;
}
```

- The declarations and the definition of a thread local object must all specify the `thread` attribute. For example, the following code generates an error:

```
#define Thread __declspec( thread )
extern int tls_i; // This will generate an error, since the
int __declspec( thread ) tls_i; // declaration and definition differ.
```

- The **thread** attribute cannot be used as a type modifier. For example, the following code generates a compiler error:

```
char __declspec( thread ) *ch;          // Error
```

- Because the declaration of C++ objects that use the **thread** attribute is permitted, the following two examples are semantically equivalent:

```
__declspec( thread ) class B
{
// Code
} BObject; // OK--BObject is declared thread local.

class B
{
// Code
};
__declspec( thread ) B BObject; // OK--BObject is declared thread local.
```

- The address of a thread local object is not considered constant, and any expression involving such an address is not considered a constant expression. In standard C, the effect of this is to forbid the use of the address of a thread local variable as an initializer for an object or pointer. For example, the following code is flagged as an error by the C compiler:

```
__declspec( thread ) int tls_i;
int *p = &tls_i;          //This will generate an error in C.
```

This restriction does not apply in C++. Because C++ allows for dynamic initialization of all objects, you can initialize an object by using an expression that uses the address of a thread local variable. This is accomplished just like the construction of thread local objects. For example, the code shown earlier does not generate an error when it is compiled as a C++ source file. Note that the address of a thread local variable is valid only as long as the thread in which the address was taken still exists.

- Standard C allows for the initialization of an object or variable with an expression involving a reference to itself, but only for objects of nonstatic extent. Although C++ generally allows for such dynamic initialization of objects with an expression involving a reference to itself, this kind of initialization is not permitted with thread local objects. For example:

```
__declspec( thread ) int tls_i = tls_i;          // Error in C and C++
int j = j;                                     // OK in C++, error in C
__declspec( thread ) int tls_i = sizeof( tls_i ) // Legal in C and C++
```

Note that a `sizeof` expression that includes the object being initialized does not represent a reference to itself and is enabled in both C and C++.

C++ does not allow such dynamic initialization of thread data because of possible future enhancements to the thread local storage facility.

- On Windows operating systems before Windows Vista, `__declspec( thread )` has some limitations. If a DLL declares any data or object as `__declspec( thread )`, it can cause a protection fault if dynamically loaded. After the DLL is loaded with `LoadLibrary`, it causes system failure whenever the code references the `__declspec( thread )` data. Because the global variable space for a thread is allocated at run time, the size of this space is based on a calculation of the requirements of the application plus the requirements of all the DLLs that are statically linked. When you use `LoadLibrary`, you cannot extend this space to allow for the

thread local variables declared with `__declspec( thread )`. Use the TLS APIs, such as [TlsAlloc](#), in your DLL to allocate TLS if the DLL might be loaded with `LoadLibrary`.

## See also

[Multithreading with C and Win32](#)

# Multithreading with C++ and MFC

3/4/2019 • 2 minutes to read • [Edit Online](#)

The Microsoft Foundation Class (MFC) library provides support for multithreaded applications. This topic describes processes and threads and the MFC approach to multithreading.

A process is an executing instance of an application. For example, when you double-click the Notepad icon, you start a process that runs Notepad.

A thread is a path of execution within a process. When you start Notepad, the operating system creates a process and begins executing the primary thread of that process. When this thread terminates, so does the process. This primary thread is supplied to the operating system by the startup code in the form of a function address. Usually, it is the address of the `main` or `WinMain` function that is supplied.

You can create additional threads in your application if you want. You might want to do this to handle background or maintenance tasks when you do not want the user to wait for them to complete. All threads in MFC applications are represented by `CWinThread` objects. In most situations, you do not even have to explicitly create these objects; instead call the framework helper function `AfxBeginThread`, which creates the `CWinThread` object for you.

MFC distinguishes two types of threads: user-interface threads and worker threads. User-interface threads are commonly used to handle user input and respond to events and messages generated by the user. Worker threads are commonly used to complete tasks, such as recalculation, that do not require user input. The Win32 API does not distinguish between types of threads; it just needs to know the thread's starting address so it can begin to execute the thread. MFC handles user-interface threads specially by supplying a message pump for events in the user interface. `CWinApp` is an example of a user-interface thread object, because it derives from `CWinThread` and handles events and messages generated by the user.

Special attention should be given to situations where more than one thread might require access to the same object. [Multithreading: Programming Tips](#) describes techniques that you can use to get around problems that might arise in these situations. [Multithreading: How to Use the Synchronization Classes](#) describes how to use the classes that are available to synchronize access from multiple threads to a single object.

Writing and debugging multithreaded programming is inherently a complicated and tricky undertaking, because you must ensure that objects are not accessed by more than one thread at a time. The multithreading topics do not teach the basics of multithreaded programming, only how to use MFC in your multithreaded program. The multithreaded MFC samples included in Visual C++ illustrate a few multithreaded Adding Functionality and Win32 APIs not encompassed by MFC; however, they are only intended to be a starting point.

For more information about how the operating system handles processes and threads, see [Processes and Threads](#) in the Windows SDK.

For more information about MFC multithreading support, see the following topics:

- [Multithreading: Creating User-Interface Threads](#)
- [Multithreading: Creating Worker Threads](#)
- [Multithreading: How to Use the Synchronization Classes](#)
- [Multithreading: Terminating Threads](#)
- [Multithreading: Programming Tips](#)

- [Multithreading: When to Use the Synchronization Classes](#)

## See also

[Multithreading Support for Older Code \(Visual C++\)](#)

# Multithreading: Creating MFC User-Interface Threads

3/4/2019 • 2 minutes to read • [Edit Online](#)

A user-interface thread is commonly used to handle user input and respond to user events independently of threads executing other portions of the application. The main application thread (provided in your `CWinApp`-derived class) is already created and started for you. This topic describes the steps necessary to create additional user-interface threads.

The first thing you must do when creating a user-interface thread is derive a class from `CWinThread`. You must declare and implement this class, using the `DECLARE_DYNCREATE` and `IMPLEMENT_DYNCREATE` macros. This class must override some functions and can override others. These functions and what they should do are presented in the following table.

## Functions to Override When Creating a User-Interface Thread

FUNCTION	PURPOSE
<a href="#">ExitInstance</a>	Perform cleanup when thread terminates. Usually overridden.
<a href="#">InitInstance</a>	Perform thread instance initialization. Must be overridden.
<a href="#">OnIdle</a>	Perform thread-specific idle-time processing. Not usually overridden.
<a href="#">PreTranslateMessage</a>	Filter messages before they are dispatched to <code>TranslateMessage</code> and <code>DispatchMessage</code> . Not usually overridden.
<a href="#">ProcessWndProcException</a>	Intercept unhandled exceptions thrown by the thread's message and command handlers. Not usually overridden.
<a href="#">Run</a>	Controlling function for the thread. Contains the message pump. Rarely overridden.

MFC provides two versions of `AfxBeginThread` through parameter overloading: one that can only create worker threads and one that can create user-interface threads or worker threads. To start your user-interface thread, call the second overload of `AfxBeginThread`, providing the following information:

- The `RUNTIME_CLASS` of the class you derived from `CWinThread`.
- (Optional) The desired priority level. The default is normal priority. For more information about the available priority levels, see `SetThreadPriority` in the Windows SDK.
- (Optional) The desired stack size for the thread. The default is the same size stack as the creating thread.
- (Optional) `CREATE_SUSPENDED` if you want the thread to be created in a suspended state. The default is 0, or start the thread normally.
- (Optional) The desired security attributes. The default is the same access as the parent thread. For more information about the format of this security information, see `SECURITY_ATTRIBUTES` in the Windows SDK.

`AfxBeginThread` does most of the work for you. It creates a new object of your class, initializes it with the

information you supply, and calls [CWinThread::CreateThread](#) to start executing the thread. Checks are made throughout the procedure to make sure all objects are deallocated properly should any part of the creation fail.

## What do you want to know more about?

- [Multithreading: Terminating Threads](#)
- [Multithreading: Creating Worker Threads](#)
- [Processes and Threads](#)

## See also

[Multithreading with C++ and MFC](#)

# Multithreading: Creating Worker Threads in MFC

3/4/2019 • 3 minutes to read • [Edit Online](#)

A worker thread is commonly used to handle background tasks that the user should not have to wait for to continue using your application. Tasks such as recalculation and background printing are good examples of worker threads. This topic details the steps necessary to create a worker thread. Topics include:

- [Starting the thread](#)
- [Implementing the controlling function](#)
- [Example](#)

Creating a worker thread is a relatively simple task. Only two steps are required to get your thread running: implementing the controlling function and starting the thread. It is not necessary to derive a class from [CWinThread](#). You can derive a class if you need a special version of `CWinThread`, but it is not required for most simple worker threads. You can use `CWinThread` without modification.

## Starting the Thread

There are two overloaded versions of `AfxBeginThread`: one that can only create worker threads, and one that can create both user-interface threads and worker threads. To begin execution of your worker thread using the first overload, call [AfxBeginThread](#), providing the following information:

- The address of the controlling function.
- The parameter to be passed to the controlling function.
- (Optional) The desired priority of the thread. The default is normal priority. For more information about the available priority levels, see [SetThreadPriority](#) in the Windows SDK.
- (Optional) The desired stack size for the thread. The default is the same size stack as the creating thread.
- (Optional) `CREATE_SUSPENDED` if you want the thread to be created in a suspended state. The default is 0, or start the thread normally.
- (Optional) The desired security attributes. The default is the same access as the parent thread. For more information about the format of this security information, see [SECURITY\\_ATTRIBUTES](#) in the Windows SDK.

`AfxBeginThread` creates and initializes a `CWinThread` object for you, starts it, and returns its address so you can refer to it later. Checks are made throughout the procedure to make sure all objects are deallocated properly should any part of the creation fail.

## Implementing the Controlling Function

The controlling function defines the thread. When this function is entered, the thread starts, and when it exits, the thread terminates. This function should have the following prototype:

```
UINT MyControllingFunction( LPVOID pParam );
```

The parameter is a single value. The value the function receives in this parameter is the value that was passed to the constructor when the thread object was created. The controlling function can interpret this value in any

manner it chooses. It can be treated as a scalar value or a pointer to a structure containing multiple parameters, or it can be ignored. If the parameter refers to a structure, the structure can be used not only to pass data from the caller to the thread, but also to pass data back from the thread to the caller. If you use such a structure to pass data back to the caller, the thread needs to notify the caller when the results are ready. For information about communicating from the worker thread to the caller, see [Multithreading: Programming Tips](#).

When the function terminates, it should return a UINT value indicating the reason for termination. Typically, this exit code is 0 to indicate success with other values indicating different types of errors. This is purely implementation dependent. Some threads might maintain usage counts of objects and return the current number of uses of that object. To see how applications can retrieve this value, see [Multithreading: Terminating Threads](#).

There are some restrictions on what you can do in a multithreaded program written with the MFC library. For descriptions of these restrictions and other tips about using threads, see [Multithreading: Programming Tips](#).

## Controlling Function Example

The following example shows how to define a controlling function and use it from another portion of the program.

```
UINT MyThreadProc( LPVOID pParam )
{
    CMyObject* pObject = (CMyObject*)pParam;

    if (pObject == NULL ||
        !pObject->IsKindOf(RUNTIME_CLASS(CMyObject)))
        return 1; // if pObject is not valid

    // do something with 'pObject'

    return 0; // thread completed successfully
}

// inside a different function in the program
.
.
.
pNewObject = new CMyObject;
AfxBeginThread(MyThreadProc, pNewObject);
.
.
.
```

## What do you want to know more about?

- [Multithreading: Creating User-Interface Threads](#)

## See also

[Multithreading with C++ and MFC](#)

# Multithreading: When to Use the MFC Synchronization Classes

3/4/2019 • 2 minutes to read • [Edit Online](#)

The multithreaded classes provided with MFC fall into two categories: synchronization objects ([CSyncObject](#), [CSemaphore](#), [CMutex](#), [CCriticalSection](#), and [CEvent](#)) and synchronization access objects ([CMultiLock](#) and [CSingleLock](#)).

Synchronization classes are used when access to a resource must be controlled to ensure integrity of the resource. Synchronization access classes are used to gain access to these controlled resources. This topic describes when to use each class.

To determine which synchronization class you should use, ask the following series of questions:

1. Does the application have to wait for something to happen before it can access the resource (for example, data must be received from a communications port before it can be written to a file)?

If yes, use `CEvent`.

2. Can more than one thread within the same application access this resource at one time (for example, your application allows up to five windows with views on the same document)?

If yes, use `CSemaphore`.

3. Can more than one application use this resource (for example, the resource is in a DLL)?

If yes, use `CMutex`.

If no, use `CCriticalSection`.

`CSyncObject` is never used directly. It is the base class for the other four synchronization classes.

## Example 1: Using Three Synchronization Classes

As an example, take an application that maintains a linked list of accounts. This application allows up to three accounts to be examined in separate windows, but only one can be updated at any particular time. When an account is updated, the updated data is sent over the network to a data archive.

This example application uses all three types of synchronization classes. Because it allows up to three accounts to be examined at one time, it uses `CSemaphore` to limit access to three view objects. When an attempt to view a fourth account occurs, the application either waits until one of the first three windows closes or it fails. When an account is updated, the application uses `CCriticalSection` to ensure that only one account is updated at a time. After the update succeeds, it signals `CEvent`, which releases a thread waiting for the event to be signaled. This thread sends the new data to the data archive.

## Example 2: Using Synchronization Access Classes

Choosing which synchronization access class to use is even simpler. If your application is concerned with accessing a single controlled resource only, use `CSingleLock`. If it needs access to any one of a number of controlled resources, use `CMultiLock`. In example 1, `CSingleLock` would have been used, because in each case only one resource is needed at any particular time.

For information about how the synchronization classes are used, see [Multithreading: How to Use the](#)

[Synchronization Classes](#). For information about synchronization, see [Synchronization](#) in the Windows SDK. For information about multithreading support in MFC, see [Multithreading with C++ and MFC](#).

## See also

[Multithreading with C++ and MFC](#)

# Multithreading: How to Use the MFC Synchronization Classes

3/4/2019 • 3 minutes to read • [Edit Online](#)

Synchronizing resource access between threads is a common problem when writing multithreaded applications. Having two or more threads simultaneously access the same data can lead to undesirable and unpredictable results. For example, one thread could be updating the contents of a structure while another thread is reading the contents of the same structure. It is unknown what data the reading thread will receive: the old data, the newly written data, or possibly a mixture of both. MFC provides a number of synchronization and synchronization access classes to aid in solving this problem. This topic explains the classes available and how to use them to create thread-safe classes in a typical multithreaded application.

A typical multithreaded application has a class that represents a resource to be shared among threads. A properly designed, fully thread-safe class does not require you to call any synchronization functions. Everything is handled internally to the class, allowing you to concentrate on how to best use the class, not about how it might get corrupted. An effective technique for creating a fully thread-safe class is to merge the synchronization class into the resource class. Merging the synchronization classes into the shared class is a straightforward process.

As an example, take an application that maintains a linked list of accounts. This application allows up to three accounts to be examined in separate windows, but only one can be updated at any particular time. When an account is updated, the updated data is sent over the network to a data archive.

This example application uses all three types of synchronization classes. Because it allows up to three accounts to be examined at one time, it uses `CSemaphore` to limit access to three view objects. When an attempt to view a fourth account occurs, the application either waits until one of the first three windows closes or it fails. When an account is updated, the application uses `CCriticalSection` to ensure that only one account is updated at a time. After the update succeeds, it signals `CEvent`, which releases a thread waiting for the event to be signaled. This thread sends the new data to the data archive.

## Designing a Thread-Safe Class

To make a class fully thread-safe, first add the appropriate synchronization class to the shared classes as a data member. In the previous account-management example, a `CSemaphore` data member would be added to the view class, a `CCriticalSection` data member would be added to the linked-list class, and a `CEvent` data member would be added to the data storage class.

Next, add synchronization calls to all member functions that modify the data in the class or access a controlled resource. In each function, you should create either a `CSingleLock` or `CMultiLock` object and call that object's `Lock` function. When the lock object goes out of scope and is destroyed, the object's destructor calls `Unlock` for you, releasing the resource. Of course, you can call `Unlock` directly if you want.

Designing your thread-safe class in this fashion allows it to be used in a multithreaded application as easily as a non-thread-safe class, but with a higher level of safety. Encapsulating the synchronization object and synchronization access object into the resource's class provides all the benefits of fully thread-safe programming without the drawback of maintaining synchronization code.

The following code example demonstrates this method by using a data member, `m_CritSection` (of type `CCriticalSection`), declared in the shared resource class and a `CSingleLock` object. The synchronization of the shared resource (derived from `CWinThread`) is attempted by creating a `CSingleLock` object using the address of the `m_CritSection` object. An attempt is made to lock the resource and, when obtained, work is done on the

shared object. When the work is finished, the resource is unlocked with a call to `Unlock`.

```
CSingleLock singleLock(&m_CritSection);
singleLock.Lock();
// resource locked
//.usage of shared resource...

singleLock.Unlock();
```

#### NOTE

`CCriticalSection`, unlike other MFC synchronization classes, does not have the option of a timed lock request. The waiting period for a thread to become free is infinite.

The drawbacks to this approach are that the class will be slightly slower than the same class without the synchronization objects added. Also, if there is a chance that more than one thread might delete the object, the merged approach might not always work. In this situation, it is better to maintain separate synchronization objects.

For information about determining which synchronization class to use in different situations, see [Multithreading: When to Use the Synchronization Classes](#). For more information about synchronization, see [Synchronization](#) in the Windows SDK. For more information about multithreading support in MFC, see [Multithreading with C++ and MFC](#).

## See also

[Multithreading with C++ and MFC](#)

# Multithreading: Terminating Threads in MFC

3/4/2019 • 2 minutes to read • [Edit Online](#)

Two normal situations cause a thread to terminate: the controlling function exits or the thread is not allowed to run to completion. If a word processor used a thread for background printing, the controlling function would terminate normally if printing completed successfully. If the user wants to cancel the printing, however, the background printing thread has to be terminated prematurely. This topic explains both how to implement each situation and how to get the exit code of a thread after it terminates.

- [Normal Thread Termination](#)
- [Premature Thread Termination](#)
- [Retrieving the Exit Code of a Thread](#)

## Normal Thread Termination

For a worker thread, normal thread termination is simple: Exit the controlling function and return a value that signifies the reason for termination. You can use either the [AfxEndThread](#) function or a **return** statement. Typically, 0 signifies successful completion, but that is up to you.

For a user-interface thread, the process is just as simple: from within the user-interface thread, call [PostQuitMessage](#) in the Windows SDK. The only parameter that `PostQuitMessage` takes is the exit code of the thread. As for worker threads, 0 typically signifies successful completion.

## Premature Thread Termination

Terminating a thread prematurely is almost as simple: Call [AfxEndThread](#) from within the thread. Pass the desired exit code as the only parameter. This stops execution of the thread, deallocates the thread's stack, detaches all DLLs attached to the thread, and deletes the thread object from memory.

`AfxEndThread` must be called from within the thread to be terminated. If you want to terminate a thread from another thread, you must set up a communication method between the two threads.

## Retrieving the Exit Code of a Thread

To get the exit code of either the worker or the user-interface thread, call the [GetExitCodeThread](#) function. For information about this function, see the Windows SDK. This function takes the handle to the thread (stored in the `m_hThread` data member of `CWinThread` objects) and the address of a `DWORD`.

If the thread is still active, `GetExitCodeThread` places `STILL_ACTIVE` in the supplied `DWORD` address; otherwise, the exit code is placed in this address.

Retrieving the exit code of `CWinThread` objects takes an extra step. By default, when a `CWinThread` thread terminates, the thread object is deleted. This means you cannot access the `m_hThread` data member because the `CWinThread` object no longer exists. To avoid this situation, do one of the following:

- Set the `m_bAutoDelete` data member to `FALSE`. This allows the `CWinThread` object to survive after the thread has been terminated. You can then access the `m_hThread` data member after the thread has been terminated. If you use this technique, however, you are responsible for destroying the `CWinThread` object because the framework will not automatically delete it for you. This is the preferred method.

- Store the thread's handle separately. After the thread is created, copy its `m_hThread` data member (using `::DuplicateHandle`) to another variable and access it through that variable. This way the object is deleted automatically when termination occurs and you can still find out why the thread terminated. Be careful that the thread does not terminate before you can duplicate the handle. The safest way to do this is to pass `CREATE_SUSPENDED` to [AfxBeginThread](#), store the handle, and then resume the thread by calling [ResumeThread](#).

Either method allows you to determine why a `CWinThread` object terminated.

## See also

[Multithreading with C++ and MFC](#)

[\\_endthread, \\_endthreadex](#)

[\\_beginthread, \\_beginthreadex](#)

[ExitThread](#)

# Multithreading: MFC Programming Tips

3/4/2019 • 2 minutes to read • [Edit Online](#)

Multithreaded applications require stricter care than single-threaded applications to ensure that operations occur in the intended order, and any data that is accessed by multiple threads is not corrupted. This topic explains techniques for avoiding potential problems when programming multithreaded applications with the Microsoft Foundation Class (MFC) library.

- [Accessing Objects from Multiple Threads](#)
- [Accessing MFC Objects from Non-MFC Threads](#)
- [Windows Handle Maps](#)
- [Communicating Between Threads](#)

## Accessing Objects from Multiple Threads

MFC objects are not thread-safe by themselves. Two separate threads cannot manipulate the same object unless you use the MFC synchronization classes and/or the appropriate Win32 synchronization objects, such as critical sections. For more information about critical sections and other related objects, see [Synchronization](#) in the Windows SDK.

The class library uses critical sections internally to protect global data structures, such as those used by the debug memory allocation.

## Accessing MFC Objects from Non-MFC Threads

If you have a multithreaded application that creates a thread in a way other than using a [CWinThread](#) object, you cannot access other MFC objects from that thread. In other words, if you want to access any MFC object from a secondary thread, you must create that thread with one of the methods described in [Multithreading: Creating User-Interface Threads](#) or [Multithreading: Creating Worker Threads](#). These methods are the only ones that allow the class library to initialize the internal variables necessary to handle multithreaded applications.

## Windows Handle Maps

As a general rule, a thread can access only MFC objects that it created. This is because temporary and permanent Windows handle maps are kept in thread local storage to help maintain protection from simultaneous access from multiple threads. For example, a worker thread cannot perform a calculation and then call a document's `UpdateAllViews` member function to have the windows that contain views on the new data modified. This has no effect at all, because the map from `CWnd` objects to HWNDs is local to the primary thread. This means that one thread might have a mapping from a Windows handle to a C++ object, but another thread might map that same handle to a different C++ object. Changes made in one thread would not be reflected in the other.

There are several ways around this problem. The first is to pass individual handles (such as an HWND) rather than C++ objects to the worker thread. The worker thread then adds these objects to its temporary map by calling the appropriate `FromHandle` member function. You could also add the object to the thread's permanent map by calling `Attach`, but this should be done only if you are guaranteed that the object will exist longer than the thread.

Another method is to create new user-defined messages corresponding to the different tasks your worker threads will be performing and post these messages to the application's main window using `::PostMessage`. This

method of communication is similar to two different applications conversing except that both threads are executing in the same address space.

For more information about handle maps, see [Technical Note 3](#). For more information about thread local storage, see [Thread Local Storage](#) and [Using Thread Local Storage](#) in the Windows SDK.

## Communicating Between Threads

MFC provides a number of classes that allow threads to synchronize access to objects to maintain thread safety. Usage of these classes is described in [Multithreading: How to Use the Synchronization Classes](#) and [Multithreading: When to Use the Synchronization Classes](#). For more information about these objects, see [Synchronization](#) in the Windows SDK.

## See also

[Multithreading with C++ and MFC](#)

# Multithreading and Locales

3/4/2019 • 9 minutes to read • [Edit Online](#)

Both the C Runtime Library and the C++ Standard Library provide support for changing the locale of your program. This topic discusses issues that arise when using the locale functionality of both libraries in a multithreaded application.

## Remarks

With the C Runtime Library, you can create multithreaded applications using the `_beginthread` and `_beginthreadex` functions. This topic only covers multithreaded applications created using these functions. For more information, see [\\_beginthread](#), [\\_beginthreadex](#).

To change the locale using the C Runtime Library, use the [setlocale](#) function. In previous versions of Visual C++, this function would always modify the locale throughout the entire application. There is now support for setting the locale on a per-thread basis. This is done using the [\\_configthreadlocale](#) function. To specify that [setlocale](#) should only change the locale in the current thread, call `_configthreadlocale(_ENABLE_PER_THREAD_LOCALE)` in that thread. Conversely, calling `_configthreadlocale(_DISABLE_PER_THREAD_LOCALE)` will cause that thread to use the global locale, and any call to [setlocale](#) in that thread will change the locale in all threads that have not explicitly enabled per-thread locale.

To change the locale using the C++ Runtime Library, use the [locale Class](#). By calling the `locale::global` method, you change the locale in every thread that has not explicitly enabled per-thread locale. To change the locale in a single thread or portion of an application, simply create an instance of a `locale` object in that thread or portion of code.

### NOTE

Calling `locale::global` changes the locale for both the C++ Standard Library and the C Runtime Library. However, calling [setlocale](#) only changes the locale for the C Runtime Library; the C++ Standard Library is not affected.

The following examples show how to use the [setlocale](#) function, the [locale Class](#), and the [\\_configthreadlocale](#) function to change the locale of an application in several different scenarios.

## Example

In this example, the main thread spawns two child threads. The first thread, Thread A, enables per-thread locale by calling `_configthreadlocale(_ENABLE_PER_THREAD_LOCALE)`. The second thread, Thread B, as well as the main thread, do not enable per-thread locale. Thread A then proceeds to change the locale using the [setlocale](#) function of the C Runtime Library.

Since Thread A has per-thread locale enabled, only the C Runtime Library functions in Thread A start using the "french" locale. The C Runtime Library functions in Thread B and in the main thread continue to use the "C" locale. Also, since [setlocale](#) does not affect the C++ Standard Library locale, all C++ Standard Library objects continue to use the "C" locale.

```
// multithread_locale_1.cpp
// compile with: /EHsc /MD
#include <locale>
#include <stdio>
#include <locale>
#include <process.h>
```

```

#include <windows.h>

#define NUM_THREADS 2
using namespace std;

unsigned __stdcall RunThreadA(void *params);
unsigned __stdcall RunThreadB(void *params);

BOOL localeSet = FALSE;
HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);

int main()
{
    HANDLE threads[NUM_THREADS];

    unsigned aID;
    threads[0] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadA, NULL, 0, &aID);

    unsigned bID;
    threads[1] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadB, NULL, 0, &bID);

    WaitForMultipleObjects(2, threads, TRUE, INFINITE);

    printf_s("[Thread main] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread main] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread main] locale::global is set to \"%s\"\n",
        locale().name().c_str());

    CloseHandle(threads[0]);
    CloseHandle(threads[1]);
    CloseHandle(printMutex);

    return 0;
}

unsigned __stdcall RunThreadA(void *params)
{
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);
    setlocale(LC_ALL, "french");
    localeSet = TRUE;

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread A] Per-thread locale is enabled.\n");
    printf_s("[Thread A] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread A] locale::global is set to \"%s\"\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}

unsigned __stdcall RunThreadB(void *params)
{
    while (!localeSet)
        Sleep(100);

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread B] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread B] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread B] locale::global is set to \"%s\"\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}

```

```
}
```

```
[Thread A] Per-thread locale is enabled.  
[Thread A] CRT locale is set to "French_France.1252"  
[Thread A] locale::global is set to "C"  
  
[Thread B] Per-thread locale is NOT enabled.  
[Thread B] CRT locale is set to "C"  
[Thread B] locale::global is set to "C"  
  
[Thread main] Per-thread locale is NOT enabled.  
[Thread main] CRT locale is set to "C"  
[Thread main] locale::global is set to "C"
```

## Example

In this example, the main thread spawns two child threads. The first thread, Thread A, enables per-thread locale by calling `_configthreadlocale(_ENABLE_PER_THREAD_LOCALE)`. The second thread, Thread B, as well as the main thread, do not enable per-thread locale. Thread A then proceeds to change the locale using the `locale::global` method of the C++ Standard Library.

Since Thread A has per-thread locale enabled, only the C Runtime Library functions in Thread A start using the "french" locale. The C Runtime Library functions in Thread B and in the main thread continue to use the "C" locale. However, since the `locale::global` method changes the locale "globally", all C++ Standard Library objects in all threads start using the "french" locale.

```
// multithread_locale_2.cpp  
// compile with: /EHsc /MD  
#include <locale>  
#include <cstdio>  
#include <locale>  
#include <process.h>  
#include <windows.h>  
  
#define NUM_THREADS 2  
using namespace std;  
  
unsigned __stdcall RunThreadA(void *params);  
unsigned __stdcall RunThreadB(void *params);  
  
BOOL localeSet = FALSE;  
HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);  
  
int main()  
{  
    HANDLE threads[NUM_THREADS];  
  
    unsigned aID;  
    threads[0] = (HANDLE)_beginthreadex(  
        NULL, 0, RunThreadA, NULL, 0, &aID);  
  
    unsigned bID;  
    threads[1] = (HANDLE)_beginthreadex(  
        NULL, 0, RunThreadB, NULL, 0, &bID);  
  
    WaitForMultipleObjects(2, threads, TRUE, INFINITE);  
  
    printf_s("[Thread main] Per-thread locale is NOT enabled.\n");  
    printf_s("[Thread main] CRT locale is set to \"%s\"\n",  
        setlocale(LC_ALL, NULL));  
    printf_s("[Thread main] locale::global is set to \"%s\"\n",  
        locale().name().c_str());  
}
```

```

    CloseHandle(threads[0]);
    CloseHandle(threads[1]);
    CloseHandle(printMutex);

    return 0;
}

unsigned __stdcall RunThreadA(void *params)
{
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);
    locale::global(locale("french"));
    localeSet = TRUE;

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread A] Per-thread locale is enabled.\n");
    printf_s("[Thread A] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread A] locale::global is set to \"%s\"\n\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}

unsigned __stdcall RunThreadB(void *params)
{
    while (!localeSet)
        Sleep(100);

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread B] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread B] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread B] locale::global is set to \"%s\"\n\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}

```

```

[Thread A] Per-thread locale is enabled.
[Thread A] CRT locale is set to "French_France.1252"
[Thread A] locale::global is set to "French_France.1252"

[Thread B] Per-thread locale is NOT enabled.
[Thread B] CRT locale is set to "C"
[Thread B] locale::global is set to "French_France.1252"

[Thread main] Per-thread locale is NOT enabled.
[Thread main] CRT locale is set to "C"
[Thread main] locale::global is set to "French_France.1252"

```

## Example

In this example, the main thread spawns two child threads. The first thread, Thread A, enables per-thread locale by calling `_configthreadlocale(_ENABLE_PER_THREAD_LOCALE)`. The second thread, Thread B, as well as the main thread, do not enable per-thread locale. Thread B then proceeds to change the locale using the [setlocale](#) function of the C Runtime Library.

Since Thread B does not have per-thread locale enabled, the C Runtime Library functions in Thread B and in the main thread start using the "french" locale. The C Runtime Library functions in Thread A continue to use the "C" locale because Thread A has per-thread locale enabled. Also, since [setlocale](#) does not affect the C++ Standard

Library locale, all C++ Standard Library objects continue to use the "C" locale.

```
// multithread_locale_3.cpp
// compile with: /EHsc /MD
#include <locale>
#include <stdio>
#include <locale>
#include <process.h>
#include <windows.h>

#define NUM_THREADS 2
using namespace std;

unsigned __stdcall RunThreadA(void *params);
unsigned __stdcall RunThreadB(void *params);

BOOL localeSet = FALSE;
BOOL configThreadLocaleCalled = FALSE;
HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);

int main()
{
    HANDLE threads[NUM_THREADS];

    unsigned aID;
    threads[0] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadA, NULL, 0, &aID);

    unsigned bID;
    threads[1] = (HANDLE)_beginthreadex(
        NULL, 0, RunThreadB, NULL, 0, &bID);

    WaitForMultipleObjects(2, threads, TRUE, INFINITE);

    printf_s("[Thread main] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread main] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread main] locale::global is set to \"%s\"\n",
        locale().name().c_str());

    CloseHandle(threads[0]);
    CloseHandle(threads[1]);
    CloseHandle(printMutex);

    return 0;
}

unsigned __stdcall RunThreadA(void *params)
{
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);
    configThreadLocaleCalled = TRUE;
    while (!localeSet)
        Sleep(100);

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread A] Per-thread locale is enabled.\n");
    printf_s("[Thread A] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread A] locale::global is set to \"%s\"\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}

unsigned __stdcall RunThreadB(void *params)
{
    while (!configThreadLocaleCalled)
```

```

        Sleep(100);
        setlocale(LC_ALL, "french");
        localeSet = TRUE;

        WaitForSingleObject(printMutex, INFINITE);
        printf_s("[Thread B] Per-thread locale is NOT enabled.\n");
        printf_s("[Thread B] CRT locale is set to \"%s\"\n",
            setlocale(LC_ALL, NULL));
        printf_s("[Thread B] locale::global is set to \"%s\"\n\n",
            locale().name().c_str());
        ReleaseMutex(printMutex);

        return 1;
    }

```

```

[Thread B] Per-thread locale is NOT enabled.
[Thread B] CRT locale is set to "French_France.1252"
[Thread B] locale::global is set to "C"

[Thread A] Per-thread locale is enabled.
[Thread A] CRT locale is set to "C"
[Thread A] locale::global is set to "C"

[Thread main] Per-thread locale is NOT enabled.
[Thread main] CRT locale is set to "French_France.1252"
[Thread main] locale::global is set to "C"

```

## Example

In this example, the main thread spawns two child threads. The first thread, Thread A, enables per-thread locale by calling `_configthreadlocale(_ENABLE_PER_THREAD_LOCALE)`. The second thread, Thread B, as well as the main thread, do not enable per-thread locale. Thread B then proceeds to change the locale using the `locale::global` method of the C++ Standard Library.

Since Thread B does not have per-thread locale enabled, the C Runtime Library functions in Thread B and in the main thread start using the "french" locale. The C Runtime Library functions in Thread A continue to use the "C" locale because Thread A has per-thread locale enabled. However, since the `locale::global` method changes the locale "globally", all C++ Standard Library objects in all threads start using the "french" locale.

```

// multithread_locale_4.cpp
// compile with: /EHsc /MD
#include <clocale>
#include <cstdio>
#include <locale>
#include <process.h>
#include <windows.h>

#define NUM_THREADS 2
using namespace std;

unsigned __stdcall RunThreadA(void *params);
unsigned __stdcall RunThreadB(void *params);

BOOL localeSet = FALSE;
BOOL configThreadLocaleCalled = FALSE;
HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);

int main()
{
    HANDLE threads[NUM_THREADS];

    unsigned aID;

```

```

threads[0] = (HANDLE)_beginthreadex(
    NULL, 0, RunThreadA, NULL, 0, &aID);

unsigned bID;
threads[1] = (HANDLE)_beginthreadex(
    NULL, 0, RunThreadB, NULL, 0, &bID);

WaitForMultipleObjects(2, threads, TRUE, INFINITE);

printf_s("[Thread main] Per-thread locale is NOT enabled.\n");
printf_s("[Thread main] CRT locale is set to \"%s\"\n",
    setlocale(LC_ALL, NULL));
printf_s("[Thread main] locale::global is set to \"%s\"\n",
    locale().name().c_str());

CloseHandle(threads[0]);
CloseHandle(threads[1]);
CloseHandle(printMutex);

return 0;
}

unsigned __stdcall RunThreadA(void *params)
{
    _configthreadlocale(_ENABLE_PER_THREAD_LOCALE);
    configThreadLocaleCalled = TRUE;
    while (!localeSet)
        Sleep(100);

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread A] Per-thread locale is enabled.\n");
    printf_s("[Thread A] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread A] locale::global is set to \"%s\"\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}

unsigned __stdcall RunThreadB(void *params)
{
    while (!configThreadLocaleCalled)
        Sleep(100);
    locale::global(locale("french"));
    localeSet = TRUE;

    WaitForSingleObject(printMutex, INFINITE);
    printf_s("[Thread B] Per-thread locale is NOT enabled.\n");
    printf_s("[Thread B] CRT locale is set to \"%s\"\n",
        setlocale(LC_ALL, NULL));
    printf_s("[Thread B] locale::global is set to \"%s\"\n",
        locale().name().c_str());
    ReleaseMutex(printMutex);

    return 1;
}

```

```
[Thread B] Per-thread locale is NOT enabled.  
[Thread B] CRT locale is set to "French_France.1252"  
[Thread B] locale::global is set to "French_France.1252"  
  
[Thread A] Per-thread locale is enabled.  
[Thread A] CRT locale is set to "C"  
[Thread A] locale::global is set to "French_France.1252"  
  
[Thread main] Per-thread locale is NOT enabled.  
[Thread main] CRT locale is set to "French_France.1252"  
[Thread main] locale::global is set to "French_France.1252"
```

## See also

[Multithreading Support for Older Code \(Visual C++\)](#)

[\\_beginthread, \\_beginthreadex](#)

[\\_configthreadlocale](#)

[setlocale](#)

[Internationalization](#)

[Locale](#)

[<locale>](#)

[<locale>](#)

[locale Class](#)