Contents

ArgTraitsHelper struct

AsyncStatusInternal enum

BoolStruct struct

ComPtrRef class

ComPtrRefBase class

CreatorMap struct

DerefHelper struct

DontUseNewUseMake class

EnableIf struct

EventTargetArray class

FactoryCache struct

ImplementsBase struct

ImplementsHelper struct

InterfaceList struct

InterfaceListHelper struct

InterfaceTraits struct

InvokeHelper struct

IsBaseOfStrict struct

IsSame struct

MakeAllocator class

MakeAndInitialize function

ModuleBase class

Move function

Nil struct

RaiseException function

RemoveIUnknown class

RemoveReference struct

RuntimeClassBase struct

RuntimeClassBaseT struct

Swap function (WRL)

TerminateMap function

VerifyInheritanceHelper struct

Universal Windows Apps (C++)

9/21/2022 • 2 minutes to read • Edit Online

The Universal Windows Platform (UWP) is the modern programming interface for Windows. With UWP you write an application or component once and deploy it on any Windows 10 or later device. You can write a component in C++ and applications written in any other UWP-compatible language can use it.

Most of the UWP documentation is in the Windows content tree at Universal Windows Platform documentation. There you will find beginning tutorials as well as reference documentation.

For new UWP apps and components, we recommend that you use C++/WinRT, a new standard C++17 language projection for Windows Runtime APIs. C++/WinRT is available in the Windows SDK from version 1803 (10.0.17134.0) onward. C++/WinRT is implemented entirely in header files, and is designed to provide you with first-class access to the modern Windows API. Unlike the C++/CX implementation, C++/WinRT doesn't use non-standard syntax or Microsoft language extensions, and it takes full advantage of the C++ compiler to create highly-optimized output. For more information, see Introduction to C++/WinRT.

You can use the Desktop Bridge app converter to package your existing desktop application for deployment through the Microsoft Store. For more information, see Using Visual C++ Runtime in Centennial project and Desktop Bridge.

UWP apps that use C++/CX

C++/CX language reference
Describes the set of extensions that simplify C++ consumption of Windows Runtime APIs and enable error handling that's based on exceptions.

Building apps and libraries (C++/CX)
Describes how to create DLLs and static libraries that can be accessed from a C++/CX app or component.

Tutorial: Create a UWP "Hello, World" app in C++/CX
A walkthrough that introduces the basic concepts of UWP app development in C++/CX.

Creating Windows Runtime Components in C++/CX
Describes how to create DLLs that other UWP apps and components can consume.

UWP game programming
Describes how to use DirectX and C++/CX to create games.

UWP Apps that Use the Windows Runtime C++ Template Library (WRL)

The Windows Runtime C++ Template Library provides the low-level COM interfaces by which ISO C++ code can access the Windows Runtime in an exception-free environment. In most cases, we recommend that you use C++/WinRT or C++/CX instead of the Windows Runtime C++ Template Library for UWP app development. For information about the Windows Runtime C++ Template Library, see Windows Runtime C++ Template Library (WRL).

See also

C++ in Visual Studio
Overview of Windows Programming in C++

C++/CX Language Reference

9/21/2022 • 2 minutes to read • Edit Online

C++/CX is a set of extensions to the C++ language that enable the creation of Windows apps and Windows Runtime components in an idiom that is as close as possible to modern C++. Use C++/CX to write Windows apps and components in native code that easily interact with Visual C#, Visual Basic, and JavaScript, and other languages that support the Windows Runtime. In those rare cases that require direct access to the raw COM interfaces, or non-exceptional code, you can use the Windows Runtime C++ Template Library (WRL).

> **NOTE**
>
> **C++/WinRT is the recommended alternative to C++/CX**. It is a new, standard C++17 language projection for Windows Runtime APIs, available in the latest Windows SDK from version 1803 (10.0.17134.0) onward. C++/WinRT is implemented entirely in header files, and designed to provide you with first-class access to the modern Windows API.
>
> With C++/WinRT, you can both consume and author Windows Runtime APIs using any standards-conformant C++17 compiler. C++/WinRT typically performs better and produces smaller binaries than any other language option for the Windows Runtime. We will continue to support C++/CX and WRL, but highly recommend that new applications use C++/WinRT. For more information, see C++/WinRT.

By using C++/CX, you can create:

- C++ Universal Windows Platform (UWP) apps that use XAML to define the user interface and use the native stack. For more information, see Create a "hello world" app in C++ (UWP).
- C++ Windows Runtime components that can be consumed by JavaScript-based Windows apps. For more information, see Creating Windows Runtime Components in C++.
- Windows DirectX games and graphics-intensive apps. For more information, see Create a simple UWP Game with DirectX.

Related articles

LINK	DESCRIPTION
Quick Reference	Table of keywords and operators for C++/CX.
Type System	Describes basic C++/CX types and programming constructs, and how to utilize C++/CX to consume and create Windows Runtime types.
Building apps and libraries	Discusses how to use the IDE to build apps and link to static libraries and DLLs.
Interoperating with Other Languages	Discusses how components that are written by using C++/CX can be used with components that are written in JavaScript, any managed language, or the Windows Runtime C++ Template Library.
Threading and Marshaling	Discusses how to specify the threading and marshaling behavior of components that you create.

LINK	DESCRIPTION
Namespaces Reference	Reference documentation for the default namespace, the Platform namespace, Platform::Collections, and related namespaces.
CRT functions not supported in Universal Windows Platform apps	Lists the CRT functions that are not available for use in Windows Runtime apps.
Get started with Windows apps	Provides high-level guidance about Windows UWP apps and links to more information.
C++/CX Part 0 of [n]: An Introduction C++/CX Part 1 of [n]: A Simple Class C++/CX Part 2 of [n]: Types That Wear Hats C++/CX Part 3 of [n]: Under Construction C++/CX Part 4 of [n]: Static Member Functions	An introductory blog series on C++/CX.

Quick Reference (C++/CX)

9/21/2022 • 4 minutes to read • Edit Online

The Windows Runtime supports Universal Windows Platform (UWP) apps. These apps execute only in a trustworthy operating system environment, use authorized functions, data types, and devices, and are distributed through the Microsoft Store. The C++/CX simplify the writing of apps for the Windows Runtime. This article is a quick reference; for more complete documentation, see Type system.

When you build on the command line, use the `/ZW` compiler option to build a UWP app or Windows Runtime component. To access Windows Runtime declarations, which are defined in the Windows Runtime metadata (.winmd) files, specify the `#using` directive or the `/FU` compiler option. When you create a project for a UWP app, Visual Studio by default sets these options and adds references to all Windows Runtime libraries.

Quick reference

CONCEPT	STANDARD C++	C++/CX	REMARKS
Fundamental types	C++ fundamental types.	C++/CX fundamental types that implement fundamental types that are defined in the Windows Runtime.	The `default` namespace contains C++/CX built-in, fundamental types. The compiler implicitly maps C++/CX fundamental types to standard C++ types. The `Platform` family of namespaces contains types that implement fundamental Windows Runtime types.
	`bool`	`bool`	An 8-bit Boolean value.
	`wchar_t`, `char16_t`	`char16`	A 16-bit nonnumeric value that represents a Unicode (UTF-16) code point.

	`short`
	
	`unsigned short`
		`int16`
		
		`uint16`
			A 16-bit signed integer.
			
			A 16-bit unsigned integer.

	`int`
	
	`unsigned int`
		`int`
		
		`uint32`
			A 32-bit signed integer.
			
			A 32-bit unsigned integer.

	`long long` -or- `__int64`
	
	`unsigned long long`
		`int64`
		
		`uint64`
			A 64-bit signed integer.
			
			A 64-bit unsigned integer.

	`float`, `double`	`float32`, `float64`	A 32-bit or 64-bit IEEE 754 floating-point number.

CONCEPT	STANDARD C++	C++/CX	REMARKS

	`enum`	`enum class`
		
		-or-
		
		`enum struct`
			A 32-bit enumeration.

	(Doesn't apply)	`Platform::Guid`	A 128-bit nonnumeric value (a GUID) in the `Platform` namespace.
	`std::time_get`	`Windows::Foundation::DateTime`	A date-time structure.
	(Doesn't apply)	`Windows::Foundation::TimeSpan`	A timespan structure.
	(Doesn't apply)	`Platform::Object^`	The reference-counted base object in the C++ view of the Windows Runtime type system.

	`std::wstring`
	
	`L"..."`
		`Platform::String^`	`Platform::String^` is a reference-counted, immutable, sequence of Unicode characters that represent text.

Pointer	Pointer to object (`*`): `std::shared_ptr`	Handle-to-object (`^` , pronounced "hat"): `T^ identifier`	All Windows Runtime classes are declared by using the handle-to-object modifier. Members of the object are accessed by using the arrow (`->`) class-member-access operator. The hat modifier means "pointer to a Windows Runtime object that is automatically reference counted." More precisely, handle-to-object declares that the compiler should insert code to manage the object's reference count automatically, and delete the object if the reference count goes to zero

CONCEPT	STANDARD C++	C++/CX	REMARKS
Reference	Reference to an object (`&`): `T& identifier`	Tracking reference (`%`): `T% identifier`	Only Windows Runtime types can be declared by using the tracking reference modifier. Members of the object are accessed by using the dot (`.`) class-member-access operator. The tracking reference means "a reference to a Windows Runtime object that is automatically reference counted." More precisely, a tracking reference declares that the compiler should insert code to manage the object's reference count automatically. The code deletes the object if the reference count goes to zero.
Dynamic type declaration	`new`	`ref new`	Allocates a Windows Runtime object and then returns a handle to that object.

Object lifetime management	`delete identifier`
	
	`delete[] identifier`
		(Invokes the destructor.)	Lifetime is determined by reference counting. A call to `delete` invokes the destructor but itself doesn't free memory.

Array declaration	`T identifier[]`
	
	`std::array identifier`
		`Array<T^>^ identifier( size )`
		
		-or-
		
		`WriteOnlyArray<T^> identifier( size )`
			Declares a one-dimensional modifiable or write-only array of type `T^`. The array itself is also a reference-counted object that must be declared by using the handle-to-object modifier.
			
			(Array declarations use a template header class that is in the `Platform` namespace.)

Class declaration	`class identifier {}`
	
	`struct identifier {}`
		`ref class identifier {}`
		
		`ref struct identifier {}`
			Declares a runtime class that has default `private` accessibility.
			
			Declares a runtime class that has default `public` accessibility.

CONCEPT	STANDARD C++	C++/CX	REMARKS

Structure declaration	`struct identifier {}`
	
	(that is, a Plain Old Data structure (POD))
		`value class identifier {}`
		
		`value struct identifier {}`
			Declares a POD struct that has default `private` accessibility.
			
			A `value class` can be represented in Windows metadata, but a standard C++ `class` can't be.
			
			Declares a POD struct that has default `public` accessibility.
			
			A `value struct` can be represented in Windows metadata, but a standard C++ `struct` can't be.

Interface declaration	abstract class that contains only pure virtual functions.	`interface class identifier {}`
		
		`interface struct identifier {}`
			Declares an interface that has default `private` accessibility.
			
			Declares an interface that has default `public` accessibility.

Delegate	`std::function`	`public delegate return-type delegate-type-identifier (` *[parameters]* `);`	Declares an object that can be invoked like a function call.

CONCEPT	STANDARD C++	C++/CX	REMARKS

Event	(Doesn't apply)	`event delegate-type-identifier event-identifier;`
		
		`delegate-type-identifier delegate-identifier = ref new delegate-type-identifier( this` *[, parameters]* `);`
		
		`event-identifier += *delegate-identifier;`
		
		-or-
		
		`EventRegistrationToken token-identifier = object.event-identifier += delegate-identifier;`
		
		-or-
		
		`auto token-identifier = object.event-identifier::add( delegate-identifier );`
		
		`object.event-identifier -= token-identifier;`
		
		-or-
		
		`object.event-identifier::remove( token-identifier );`
			Declares an `event` object, which stores a collection of event handlers (delegates) that are called when an event occurs.
			
			Creates an event handler.
			
			Adds an event handler.
			
			Adding an event handler returns an event token (*`token-identifier`*). If you intend to explicitly remove the event handler, you must save the event token for later use.
			
			Removes an event handler.
			
			To remove an event handler, you must specify the event token that you saved when the event handler was added.

Property	(Doesn't apply)	`property T identifier;`
		
		`property T identifier[ index ];`
		
		`property T default[ index ];`
			Declares that a class or object member function is accessed by using the same syntax that's used to access a data member or indexed array element.
			
			Declares a property on a class or object member function.
			
			Declares an indexed property on an object member function.
			
			Declares an indexed property on a class member function.

CONCEPT	STANDARD C++	C++/CX	REMARKS

Parameterized types	templates	`generic <typename T> interface class identifier {}`
		
		`generic <typename T > delegate` *[return-type]* `delegate-identifier() {}`
			Declares a parameterized interface class.
			
			Declares a parameterized delegate.

Nullable value types	`std::optional<T>`	`Platform::IBox <T>`	Enables variables of scalar types and `value` structs to have a value of `nullptr`.

See also

C++/CX language reference

Type system overview (C++/CX)

9/21/2022 • 9 minutes to read • Edit Online

By using the Windows Runtime architecture, you can use C++/WinRT, C++/CX, Visual Basic, Visual C#, and JavaScript to write apps and components. They can directly access the Windows API and interoperate with other Windows Runtime apps and components. Universal Windows Platform (UWP) apps that are written in C++ compile to native code that executes directly in the CPU. UWP apps that are written in C# or Visual Basic compile to Microsoft intermediate language (MSIL) and execute in the common language runtime (CLR). UWP apps that are written in JavaScript execute in a JavaScript run-time environment. The Windows Runtime operating system components themselves are written in C++ and run as native code. All of these components and UWP apps communicate directly through the Windows Runtime application binary interface (ABI).

To enable support for the Windows Runtime in a modern C++ idiom, Microsoft created the C++/CX language extension. C++/CX provides built-in base types and implementations of fundamental Windows Runtime types. These types let C++ apps and components communicate across the ABI with apps written in other languages. C++/CX apps can consume any Windows Runtime type. They may also create classes, structs, interfaces, and other user-defined types that other UWP apps and components can consume. A UWP app that's written in C++/CX can also use regular C++ classes and structs, as long as they don't have public accessibility.

For an in-depth discussion of the C++/CX language projection and how it works under the covers, see these blog posts:

- C++/CX Part 0 of [n]: An Introduction
- C++/CX Part 1 of [n]: A Simple Class
- C++/CX Part 2 of [n]: Types That Wear Hats
- C++/CX Part 3 of [n]: Under Construction
- C++/CX Part 4 of [n]: Static Member Functions

> **NOTE**
> While C++/CX is still supported, we recommend you use C++/WinRT for new Windows Runtime apps and components instead. It's designed to provide you with first-class access to the modern Windows API. Despite the name, C++/WinRT uses only standard C++17 with no extensions. It uses a header-only library to implement a C++ language projection for Windows Runtime APIs. C++/WinRT is available in the Windows SDK from version 1803 (10.0.17134.0) onward.

Windows metadata (.winmd) files

When you compile a UWP app that's written in C++, the compiler generates the executable in native machine code, and also generates a separate Windows metadata (`.winmd`) file that contains descriptions of the public Windows Runtime types, which include classes, structs, enumerations, interfaces, parameterized interfaces, and delegates. The format of the metadata resembles the format that's used in .NET Framework assemblies. In a C++ component, the `.winmd` file contains only metadata; the executable code is in a separate file. The Windows Runtime components included with Windows use this arrangement. A `.winmd` file name must match or be a prefix of the root namespace in the source code. (For .NET Framework languages, the `.winmd` file contains both the code and the metadata, just like a .NET Framework assembly.)

The metadata in the `.winmd` file represents the published surface of your code. Published types are visible to other UWP apps no matter what language those other apps are written in. The metadata, or your published

code, can only contain types specified by the Windows Runtime type system. You can't publish C++-specific language constructs such as regular classes, arrays, templates, or C++ Standard Library (STL) containers. A JavaScript or C# client app wouldn't know what to do with them.

Whether a type or method is visible in metadata depends on what accessibility modifiers are applied to it. To be visible, a type must be declared in a namespace and must be declared as `public`. A non-`public` `ref class` is permitted as an internal helper type in your code; it just isn't visible in the metadata. Even in a `public ref class`, not all members are necessarily visible. The following table lists the relationship between C++ access specifiers in a `public ref class`, and Windows Runtime metadata visibility:

PUBLISHED IN METADATA	NOT PUBLISHED IN METADATA
`public`	`private`
`protected`	`internal`
`public protected`	`private protected`

You can use the **Object Browser** to view the contents of `.winmd` files. The Windows Runtime components included with Windows are found in the `Windows.winmd` file. The `default.winmd` file contains the fundamental types that are used in C++/CX, and `platform.winmd` contains types from the `Platform` namespace. By default, these three `.winmd` files are included in every C++ project for UWP apps.

> **TIP**
> The types in the `Platform::Collections` namespace don't appear in the `.winmd` file because they're not public. They're private C++-specific implementations of the interfaces that are defined in `Windows::Foundation::Collections`. A Windows Runtime app that's written in JavaScript or C# doesn't know what a `Platform::Collections::Vector` class is, but it can consume a `Windows::Foundation::Collections::IVector`. The `Platform::Collections` types are defined in `collection.h`.

Windows Runtime type system in C++/CX

The following sections describe the major features of the Windows Runtime type system and how they're supported in C++/CX.

Namespaces

All Windows Runtime types must be declared within a namespace; the Windows API itself is organized by namespace. A `.winmd` file must have the same name that the root namespace has. For example, a class that's named `A.B.C.MyClass` can be instantiated only if it's defined in a metadata file that's named `A.winmd`, `A.B.winmd`, or `A.B.C.winmd`. The name of the DLL isn't required to match the `.winmd` file name.

The Windows API itself has been reinvented as a well-factored class library that's organized by namespaces. All Windows Runtime components are declared in the `Windows.*` namespaces.

For more information, see Namespaces and type visibility.

Fundamental types

The Windows Runtime defines the following fundamental types: `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `Char16`, `Boolean`, and `String`. C++/CX supports the fundamental numeric types in its default namespace as `uint16`, `uint32`, `uint64`, `int16`, `int32`, `int64`, `float32`, `float64`, and `char16`. `Boolean` and `String` are also defined in the `Platform` namespace.

C++/CX also defines `uint8`, equivalent to `unsigned char`, which isn't supported in the Windows Runtime and can't be used in public APIs.

A fundamental type may be made nullable by wrapping it in a `Platform::IBox` interface. For more information, see Value classes and structs.

For more information about fundamental types, see Fundamental types

Strings

A Windows Runtime string is an immutable sequence of 16-bit UNICODE characters. A Windows Runtime string is projected as `Platform::String^`. This class provides methods for string construction, manipulation, and conversion to and from `wchar_t`.

For more information, see Strings.

Arrays

The Windows Runtime supports one-dimensional arrays of any type. Arrays of arrays aren't supported. In C++/CX, Windows Runtime arrays are projected as the `Platform::Array` class.

For more information, see `Array` and `WriteOnlyArray`.

`ref class` and `ref struct` types

A Windows Runtime class is projected in C++/CX as a `ref class` or `ref struct` type, because it's copied by reference. Memory management for `ref class` and `ref struct` objects is handled transparently through reference counting. When the last reference to an object goes out of scope, the object is destroyed. A `ref class` or `ref struct` type can:

- Contain as members constructors, methods, properties, and events. These members can have `public`, `private`, `protected`, or `internal` accessibility.
- Can contain private nested `enum`, `struct`, or `class` definitions.
- Can directly inherit from one base class and can implement any number of interfaces. All `ref class` objects are implicitly convertible to the `Platform::Object` class and can override its virtual methods—for example, `Object::ToString`.

A `ref class` that has a public constructor must be declared as `sealed`, to prevent further derivation.

For more information, see Ref classes and structs

`value class` and `value struct` types

A `value class` or `value struct` represents a basic data structure and contains only fields, which may be `value class` types, `value struct` types, or type `Platform::String^`. `value struct` and `value class` objects are copied by value.

A `value struct` can be made nullable by wrapping it in an `IBox` interface.

For more information, see Value classes and structs.

Partial classes

The partial class feature enables one class to be defined over multiple files. It lets code-generation tools such as the XAML editor modify one file without touching another file that you edit.

For more information, see Partial classes

Properties

A property is a public data member of any Windows Runtime type. It's declared and defined by using the

`property` keyword. A property is implemented as a `get` / `set` method pair. Client code accesses a property as if it were a public field. A property that requires no custom `get` or `set` code is known as a *trivial property* and can be declared without explicit `get` or `set` methods.

For more information, see Properties.

Windows Runtime collections in C++/CX

The Windows Runtime defines a set of interfaces for collection types that each language implements in its own way. C++/CX provides implementations in the `Platform::Collections::Vector` class, `Platform::Collections::Map` class, and other related concrete collection types, which are compatible with their C++ Standard Library counterparts.

For more information, see Collections.

Template `ref class` types

`private` and `internal` access `ref class` types can be templated and specialized.

For more information, see Template ref classes.

Interfaces

A Windows Runtime interface defines a set of public properties, methods, and events that a `ref class` or `ref struct` type must implement if it inherits from the interface.

For more information, see Interfaces.

Enums

An `enum class` type in the Windows Runtime resembles a scoped `enum` in C++. The underlying type is `int32`, unless the [Flags] attribute is applied—in that case, the underlying type is `uint32`.

For more information, see Enums.

Delegates

A delegate in the Windows Runtime is analogous to a `std::function` object in C++. It's a special `ref class` type that's used to invoke client-provided functions that have compatible signatures. Delegates are most commonly used in the Windows Runtime as the type of an event.

For more information, see Delegates.

Exceptions

In C++/CX, you can catch custom exception types, `std::exception` types, and `Platform::Exception` types.

For more information, see Exceptions.

Events

An event is a public member in a `ref class` or `ref struct` whose type is a delegate type. An event can only be invoked—that is, fired—by the owning class. However, client code can provide its own event handler functions, which are invoked when the owning class fires the event.

For more information, see Events.

Casting

C++/CX supports the standard C++ cast operators `static_cast`, `dynamic_cast`, and `reinterpret_cast`, and also the `safe_cast` operator that's specific to C++/CX.

For more information, see Casting.

Boxing

A boxed variable is a value type that's wrapped in a reference type. Use boxed variables in situations where reference semantics are required.

For more information, see Boxing.

Attributes

An attribute is a metadata value that can be applied to any Windows Runtime type or type member. Attributes can be inspected at run time. The Windows Runtime defines a set of common attributes in the `Windows::Foundation::Metadata` namespace. User-defined attributes on public interfaces are not supported by Windows Runtime in this release.

API deprecation

You can mark public APIs as deprecated by using the same attribute that's used by the Windows Runtime system types.

For more information, see Deprecating types and members.

See also

C++/CX language reference

Namespaces and Type Visibility (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

A namespace is a standard C++ construct for grouping types that have related functionality and for preventing name collisions in libraries. The Windows Runtime type system requires that all public Windows Runtime types, including those in your own code, must be declared in a namespace at namespace scope. Public types that are declared at global scope or nested inside another class will cause a compile-time error.

A .winmd file must have the same name that the root namespace has. For example, a class that's named A.B.C.MyClass can be instantiated only if it's defined in a metadata file that's named A.winmd or A.B.winmd or A.B.C.winmd. The name of the executable is not required to match the .winmd file name.

Type visibility

In a namespace, Windows Runtime types—unlike standard C++ types—have either private or public accessibility. By default, the accessibility is private. Only a public type is visible to metadata and is therefore consumable from apps and components that might be written in languages other than C++. In general, the rules for visible types are more restrictive than the rules for non-visible types because visible types cannot expose C++-specific concepts that are not supported in .NET languages or JavaScript.

> **NOTE**
> Metadata is only consumed at run time by .NET languages and JavaScript. When a C++ app or component is talking to another C++ app or component—this includes Windows components ,which are all written in C++—then no run-time consumption of metadata is required.

Member accessibility and visibility

In a private ref class, interface, or delegate, no members are emitted to metadata, even if they have public accessibility. In public ref classes, you can control the visibility of members in metadata independently of their accessibility in your source code. As in standard C++, apply the principle of least privilege; don't make your members visible in metadata unless they absolutely must be.

Use the following access modifiers to control both metadata visibility and source code accessibility.

MODIFIER	MEANING	EMITTED TO METADATA?
`private`	The default accessibility. Same meaning as in standard C++.	No
`protected`	Same meaning as in standard C++, both within the app or component and in metadata.	Yes
`public`	Same meaning as in standard C++.	Yes
`public protected` -or- `protected public`	Protected accessibility in metadata, public within the app or component.	Yes

MODIFIER	MEANING	EMITTED TO METADATA?
`protected private` or `private protected`	Not visible in metadata; protected accessibility within the app or component.	
`internal` or `private public`	The member is public within the app or component, but is not visible in metadata.	No

Windows Runtime namespaces

The Windows API consists of types that are declared in the Windows::* namespaces. These namespaces are reserved for Windows, and types cannot be added to them. In the **Object Browser**, you can view these namespaces in the windows.winmd file. For documentation about these namespaces, see Windows API.

C++/CX namespaces

The C++/CX define certain types in these namespaces as part of the projection of the Windows Runtime type system.

NAMESPACE	DESCRIPTION
default	Contains the built-in numeric and char16 types. These types are in scope in every namespace and a `using` statement is never required.
`Platform`	Contains primarily public types that correspond to Windows Runtime types such as `Array<T>`, `String`, `Guid`, and `Boolean`. Also includes specialized helper types such as `Platform::Agile<T>` and `Platform::Box<T>`.
`Platform::Collections`	Contains the concrete collection classes that implement the Windows Runtime collection interfaces `IVector`, `IMap`, and so on. These types are defined in a header file, collection.h, not in platform.winmd.
`Platform::Details`	Contains types that are used by the compiler and are not meant for public consumption.

See also

Type System (C++/CX)

Fundamental types (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

In addition to the standard C++ built-in types, C++/CX supports the type system that's defined by the Windows Runtime architecture by providing typedefs for the Windows Runtime fundamental types that map to standard C++ types.. C++/CX implements Boolean, character, and numeric fundamental types. These typedefs are defined in the `default` namespace, which never needs to be specified explicitly. In addition, C++/CX provides wrappers and concrete implementations for certain Windows Runtime types and interfaces.

Boolean and Character Types

The following table lists the built-in Boolean and character types, and their standard C++ equivalents.

NAMESPACE	C++/CX NAME	DEFINITION	STANDARD C++ NAME	RANGE OF VALUES
Platform	Boolean	An 8-bit Boolean value.	bool	`true` (nonzero) and `false` (zero)
default	char16	A 16-bit non-numeric value that represents a Unicode (UTF-16) code point.	wchar_t -or- L'c'	(Specified by the Unicode standard)

Numeric types

The following table lists the built-in numeric types. The numeric types are declared in the `default` namespace and are typedefs for the corresponding C++ built-in type. Not all C++ built-in types (long, for example) are supported in the Windows Runtime. For consistency and clarity, we recommend that you use the C++/CX name.

C++/CX NAME	DEFINITION	STANDARD C++ NAME	RANGE OF VALUES
int8	An 8-bit signed numeric value.	signed char	-128 through 127
uint8	An 8-bit unsigned numeric value.	unsigned char	0 through 255
int16	A 16-bit signed integer.	short	-32,768 through 32,767
uint16	A 16-bit unsigned integer.	unsigned short	0 through 65,535
int32	A 32-bit signed integer.	int	-2,147,483,648 through 2,147,483,647
uint32	A 32-bit unsigned integer.	unsigned int	0 through 4,294,967,295
int64	A 64-bit signed integer.	long long -or- __int64	-9,223,372,036,854, 775,808 through 9,223,372,036,854,775,807

C++/CX NAME	DEFINITION	STANDARD C++ NAME	RANGE OF VALUES
uint64	A 64-bit unsigned integer.	unsigned long long -or- unsigned __int64	0 through 18,446,744,073,709,551,615
float32	A 32-bit IEEE 754 floating-point number.	float	3.4E +/- 38 (7 digits)
float64	A 64-bit IEEE 754 floating-point number.	double	1.7E +/- 308 (15 digits)

Windows Runtime Types

The following table lists some additional types that are defined by the Windows Runtime architecture and are built into C++/CX. Object and String are reference types. The others are value types. All of these types are declared in the `Platform` namespace. For a full list, see Platform namespace.

NAME	DEFINITION
Object	Represents any Windows Runtime type.
String	A series of characters that represent text.
Rect	A set of four floating-point numbers that represent the location and size of a rectangle.
SizeT	An ordered pair of floating-point numbers that specify a height and width.
Point	An ordered pair of floating-point x-coordinates and y-coordinates that define a point in a two-dimensional plane.
Guid	A 128-bit non-numeric value that is used as a unique identifier.
UIntPtr	(For internal use only.) An unsigned 64-bit value that is used as a pointer.
IntPtr	(For internal use only.) A signed 64-bit value that is used as a pointer.

See also

Type System

Strings (C++/CX)

9/21/2022 • 5 minutes to read • Edit Online

Text in the Windows Runtime is represented in C++/CX by the Platform::String Class. Use the `Platform::String Class` when you pass strings back and forth to methods in Windows Runtime classes, or when you are interacting with other Windows Runtime components across the application binary interface (ABI) boundary. The `Platform::String Class` provides methods for several common string operations, but it's not designed to be a full-featured string class. In your C++ module, use standard C++ string types such as wstring for any significant text processing, and then convert the final result to Platform::String^ before you pass it to or from a public interface. It's easy and efficient to convert between `wstring` or `wchar_t*` and `Platform::String`.

Fast pass

In some cases, the compiler can verify that it can safely construct a `Platform::String` or pass a `String` to a function without copying the underlying string data. Such operations are known as *fast pass* and they occur transparently.

String construction

The value of a `String` object is an immutable (read-only) sequence of `char16` (16-bit Unicode) characters. Because a `String` object is immutable, assignment of a new string literal to a `String` variable actually replaces the original `String` object with a new `String` object. Concatenation operations involve the destruction of the original `String` object and the creation of a new object.

Literals

A *literal character* is a character that's enclosed in single quotation marks, and a *literal string* is a sequence of characters that's enclosed in double quotation marks. If you use a literal to initialize a String^ variable, the compiler assumes that the literal consists of `char16` characters. That is, you don't have to precede the literal with the 'L' string modifier or enclose the literal in a **_T()** or **TEXT()** macro. For more information about C++ support for Unicode, see Unicode Programming Summary.

The following example shows various ways to construct `String` objects.

```
// Initializing a String^ by using string literals
String^ str1 = "Test"; // ok for ANSI text only. uses current code page
String^ str2("Test");
String^ str3 = L"Test";
String^ str4(L"Test");


//Initialize a String^ by using another String^
String^ str6(str1);
auto str7 = str2;

// Initialize a String from wchar_t* and wstring
wchar_t msg[] = L"Test";
String^ str8 = ref new String(msg);
std::wstring wstr1(L"Test");
String^ str9 = ref new String(wstr1.c_str());
String^ str10 = ref new String(wstr1.c_str(), wstr1.length());
```

String handling operations

The `String` class provides methods and operators for concatenating, comparing strings, and other basic string operations. To perform more extensive string manipulations, use the `String::Data()` member function to retrieve the value of the `String^` object as a `const wchar_t*`. Then use that value to initialize a `std::wstring`, which provides rich string handling functions.

```
// Concatenation
auto str1 = "Hello" + " World";
auto str2 = str1 + " from C++/CX!";
auto str3 = String::Concat(str2, " and the String class");

// Comparison
if (str1 == str2) { /* ... */ }
if (str1->Equals(str2)) { /* ... */ }
if (str1 != str2) { /* ... */ }
if (str1 < str2 || str1 > str2) { /* ... */};
int result = String::CompareOrdinal(str1, str2);

if(str1 == nullptr) { /* ...*/};
if(str1->IsEmpty()) { /* ...*/};

// Accessing individual characters in a String^
auto it = str1->Begin();
char16 ch = it[0];
```

String conversions

A `Platform::String` can contain only `char16` characters, or the `NULL` character. If your application has to work with 8-bit characters, use the String::Data to extract the text as a `const wchar_t*`. You can then use the appropriate Windows functions or Standard Library functions to operate on the data and convert it back to a `wchar_t*` or wstring, which you can use to construct a new `Platform::String`.

The following code fragment shows how to convert a `String^` variable to and from a `wstring` variable. For more information about the string manipulation that's used in this example, see basic_string::replace.

```
// Create a String^ variable statically or dynamically from a literal string.
String^ str1 = "AAAAAAAA";

// Use the value of str1 to create the ws1 wstring variable.
std::wstring ws1( str1->Data() );
// The value of ws1 is L"AAAAAAAA".

// Manipulate the wstring value.
std::wstring replacement( L"BBB" );
ws1 = ws1.replace ( 1, 3, replacement );
// The value of ws1 is L"ABBBAAAA".

// Assign the modified wstring back to str1.
str1 = ref new String( ws1.c_str() );
```

String length and embedded NULL values

The String::Length returns the number of characters in the string, not the number of bytes. The terminating NULL character is not counted unless you explicitly specify it when you use stack semantics to construct a string.

A `Platform::String` can contain embedded NULL values, but only when the NULL is a result of a concatenation operation. Embedded NULLs are not supported in string literals; therefore, you cannot use embedded NULLs in that manner to initialize a `Platform::String`. Embedded NULL values in a `Platform::String` are ignored when

the string is displayed, for example, when it is assigned to a `TextBlock::Text` property. Embedded NULLs are removed when the string value is returned by the `Data` property.

StringReference

In some cases your code (a) receives a std::wstring, or wchar_t string or L"" string literal and just passes it on to another method that takes a String^ as input parameter. As long as the original string buffer itself remains valid and does not mutate before the function returns, you can convert the `wchar_t*` string or string literal to a Platform::StringReference, and pass in that instead of a `Platform::String^`. This is allowed because `StringReference` has a user-defined conversion to `Platform::String^`. By using `StringReference` you can avoid making an extra copy of the string data. In loops where you are passing large numbers of strings, or when passing very large strings, you can potentially achieve a significant performance improvement by using `StringReference`. But because `StringReference` essentially borrows the original string buffer, you must use extreme care to avoid memory corruption. You should not pass a `StringReference` to an asynchronous method unless the original string is guaranteed to be in scope when that method returns. A String^ that is initialized from a StringReference will force an allocation and copy of the string data if a second assignment operation occurs. In this case, you will lose the performance benefit of `StringReference`.

Note that `StringReference` is a standard C++ class type, not a ref class, you cannot use it in the public interface of ref classes that you define.

The following example shows how to use StringReference:

```
void GetDecodedStrings(std::vector<std::wstring> strings)
{
    using namespace Windows::Security::Cryptography;
    using namespace Windows::Storage::Streams;

    for (auto&& s : strings)
    {
        // Method signature is IBuffer^ CryptographicBuffer::DecodeFromBase64String (Platform::String^)
        // Call using StringReference:
        IBuffer^ buffer = CryptographicBuffer::DecodeFromBase64String(StringReference(s.c_str()));

        //...do something with buffer
    }
}
```

Array and WriteOnlyArray (C++/CX)

9/21/2022 • 5 minutes to read • Edit Online

You can freely use regular C-style arrays or `std::array` in a C++/CX program (although `std::vector` is often a better choice), but in any API that is published in metadata, you must convert a C-style array or vector to a `Platform::Array` or `Platform::WriteOnlyArray` type depending on how it is being used. The `Platform::Array` type is neither as efficient nor as powerful as `std::vector`, so as a general guideline you should avoid its use in internal code that performs lots of operations on the array elements.

The following array types can be passed across the ABI:

1. `const Platform::Array^`
2. `Platform::Array^*`
3. `Platform::WriteOnlyArray`
4. return value of `Platform::Array^`

You use these array types to implement the three kinds of array patterns that are defined by the Windows Runtime.

PassArray
Used when the caller passes an array to a method. The C++ input parameter type is `const` `Platform::Array` <T>.

FillArray
Used when the caller passes an array for the method to fill. The C++ input parameter type is `Platform::WriteOnlyArray` <T>.

ReceiveArray
Used when the caller receives an array that the method allocates. In C++/CX you can return the array in the return value as an Array^ or you can return it as an out parameter as type Array^*.

PassArray pattern

When client code passes an array to a C++ method and the method does not modify it, the method accepts the array as a `const Array^`. At the Windows Runtime application binary interface (ABI) level, this is known as a *PassArray*. The next example shows how to pass an array that's allocated in JavaScript to a C++ function that reads from it.

```
//JavaScript
function button2_click() {
    var obj = new JS-Array.Class1();
    var a = new Array(100);
    for (i = 0; i < 100; i++) {
        a[i] = i;
    }
    // Notice that method names are camelCased in JavaScript.
    var sum = obj.passArrayForReading(a);
    document.getElementById('results').innerText
        = "The sum of all the numbers is " + sum;
}
```

The following snippet shows the C++ method:

```
double Class1::PassArrayForReading(const Array<double>^ arr)
{
    double sum = 0;
    for(unsigned int i = 0 ; i < arr->Length; i++)
    {
        sum += arr[i];
    }
    return sum;
}
```

ReceiveArray pattern

In the *ReceiveArray* pattern, client code declares an array and passes it to a method which allocates the memory for it and initializes it. The C++ input parameter type is a pointer-to-hat: `Array<T>^*`. The following example shows how to declare an array object in JavaScript, and pass it to a C++ function that allocates the memory, initializes the elements, and returns it to JavaScript. JavaScript treats the allocated array as a return value, but the C++ function treats it as an out parameter.

```
//JavaScript
function button3_click() {
    var obj = new JS-Array.Class1();

    // Remember to use camelCase for the function name.
    var array2 = obj.calleeAllocatedDemo2();
    for (j = 0; j < array2.length; j++) {
        document.getElementById('results').innerText += array2[j] + " ";
    }
}
```

The following snippet shows two ways to implement the C++ method:

```
// Return array as out parameter...
void Class1::CalleeAllocatedDemo(Array<int>^* arr)
{
    auto temp = ref new Array<int>(10);
    for(unsigned int i = 0; i < temp->Length; i++)
    {
        temp[i] = i;
    }

    *arr = temp;
}

// ...or return array as return value:
Array<int>^ Class1::CalleeAllocatedDemo2()
{
    auto temp = ref new Array<int>(10);
    for(unsigned int i = 0; i < temp->Length; i++)
    {
        temp[i] = i;
    }

    return temp;
}
```

Fill arrays

When you want to allocate an array in the caller, and initialize or modify it in the callee, use `WriteOnlyArray`. The next example shows how to implement a C++ function that uses `WriteOnlyArray` and call it from JavaScript.

```
// JavaScript
function button4_click() {
    var obj = new JS-Array.Class1();
    //Allocate the array.
    var a = new Array(10);

    //Pass the array to C++.
    obj.callerAllocatedDemo(a);

    var results = document.getElementById('results');
    // Display the modified contents.
    for (i = 0; i < 10; i++) {
        document.getElementById('results').innerText += a[i] + " ";
    }
}
```

The following snippet shows how to implement the C++ method:

```
void Class1::CallerAllocatedDemo(Platform::WriteOnlyArray<int>^ arr)
{
    // You can write to the elements directly.
    for(unsigned int i = 0; i < arr->Length; i++)
    {
        arr[i] = i;
    }
}
```

Array conversions

This example shows how to use a `Platform::Array` to construct other kinds of collections:

```
#include <vector>
#include <collection.h>
using namespace Platform;
using namespace std;
using namespace Platform::Collections;

void ArrayConversions(const Array<int>^ arr)
{
    // Construct an Array from another Array.
    Platform::Array<int>^ newArr = ref new Platform::Array<int>(arr);

    // Construct a Vector from an Array
    auto v = ref new Platform::Collections::Vector<int>(arr);

    // Construct a std::vector. Two options.
    vector<int> v1(begin(arr), end(arr));
    vector<int> v2(arr->begin(), arr->end());

    // Initialize a vector one element at a time.
    // using a range for loop. Not as efficient as using begin/end.
    vector<int> v3;
    for(int i : arr)
    {
        v3.push_back(i);
    }
}
```

The next example shows how to construct a `Platform::Array` from a C-style array and return it from a public method.

```
Array<int>^ GetNums()
{
    int nums[] = {0,1,2,3,4};
    //Use nums internally....

    // Convert to Platform::Array and return to caller.
    return ref new Array<int>(nums, 5);
}
```

Jagged arrays

The Windows Runtime type system does not support the concept of jagged arrays and therefore you cannot use `IVector<Platform::Array<T>>` as a return value or method parameter in a public method. To pass a jagged array or a sequence of sequences across the ABI, use `IVector<IVector<T>^>`.

Use ArrayReference to avoid copying data

In some scenarios where data is being passed across the ABI into a `Platform::Array`, and you ultimately want to process that data in a C-style array for efficiency, you can use Platform::ArrayReference to avoid the extra copy operation. When you pass a `Platform::ArrayReference` as an argument to a parameter that takes a `Platform::Array`, the `ArrayReference` will store the data directly into a C-style array that you specify. Just be aware that `ArrayReference` has no lock on the source data, so if it that data is modified or deleted on another thread before the call completes, the results will be undefined.

The following code snippet shows how to copy the results of a `DataReader` operation into a `Platform::Array` (the usual pattern), and then how to substitute `ArrayReference` to copy the data directly into a C-style array:

```
public ref class TestReferenceArray sealed
{
public:

    // Assume dr is already initialized with a stream
    void GetArray(Windows::Storage::Streams::DataReader^ dr, int numBytesRemaining)
    {
        // Copy into Platform::Array
        auto bytes = ref new Platform::Array<unsigned char>(numBytesRemaining);

        // Fill an Array.
        dr->ReadBytes(bytes);

        // Fill a C-style array
        uint8 data[1024];
        dr->ReadBytes( Platform::ArrayReference<uint8>(data, 1024) );
    }
};
```

Avoid exposing an Array as a property

In general, you should avoid exposing a `Platform::Array` type as a property in a ref class because the entire array is returned even when client code is only attempting to access a single element. When you need to expose a sequence container as a property in a public ref class, `Windows::Foundation::IVector` is a better choice. In private or internal APIs (which are not published to metadata), consider using a standard C++ container such as `std::vector`.

See also

Type System
C++/CX Language Reference
Namespaces Reference

Ref classes and structs (C++/CX)

9/21/2022 • 8 minutes to read • Edit Online

The C++/CX supports user-defined *ref classes* and *ref structs*, and user-defined *value classes* and *value structs*. These data structures are the primary containers by which C++/CX supports the Windows Runtime type system. Their contents are emitted to metadata according to certain specific rules, and this enables them to be passed between Windows Runtime components and Universal Windows Platform apps that are written in C++ or other languages.

A ref class or ref struct has these essential features:

- It must be declared within a namespace, at namespace scope, and in that namespace it may have public or private accessibility. Only public types are emitted to metadata. Nested public class definitions are not permitted, including nested public enum classes. For more information, see Namespaces and Type Visibility.
- It may contain as members C++/CX including ref classes, value classes, ref structs, value structs, or nullable value structs. It may also contain scalar types such as `float64`, `bool`, and so on. It may also contain standard C++ types such as `std::vector` or a custom class, as long as they are not public. C++/CX constructs may have `public`, `protected`, `internal`, `private`, or `protected private` accessibility. All `public` or `protected` members are emitted to metadata. Standard C++ types must have `private`, `internal`, or `protected private` accessibility, which prevents them from being emitted to metadata.
- It may implement one or more *interface classes* or *interface structs*.
- It may inherit from one base class, and base classes themselves have additional restrictions. Inheritance in public ref class hierarchies has more restrictions than inheritance in private ref classes.
- It may not be declared as generic. If it has private accessibility, it may be a template.
- Its lifetime is managed by automatic reference counting.

Declaration

The following code fragment declares the `Person` ref class. Notice that the standard C++ `std::map` type is used in the private members, and the Windows Runtime `IMapView` interface is used in the public interface. Also notice that the "^" is appended to declarations of reference types.

```
// #include <map>
namespace WFC = Windows::Foundation::Collections;
namespace WFM = Windows::Foundation::Metadata;

[WFM::WebHostHidden]
ref class Person sealed
{
public:
    Person(Platform::String^ name);
    void AddPhoneNumber(Platform::String^ type, Platform::String^ number);
    property WFC::IMapView<Platform::String^, Platform::String^>^ PhoneNumbers
    {
        WFC::IMapView<Platform::String^, Platform::String^>^ get();
    }
private:
    Platform::String^ m_name;
    std::map<Platform::String^, Platform::String^> m_numbers;
};
```

Implementation

This code example shows an implementation of the `Person` ref class:

```
#include <collection.h>
using namespace Windows::Foundation::Collections;
using namespace Platform;
using namespace Platform::Collections;

Person::Person(String^ name): m_name(name) { }
void Person::AddPhoneNumber(String^ type, String^ number)
{
    m_numbers[type] = number;
}
IMapView< String^, String^>^ Person::PhoneNumbers::get()
{
    // Simple implementation.
    return ref new MapView< String^, String^>(m_numbers);
}
```

Usage

The next code example shows how client code uses the `Person` ref class.

```
using namespace Platform;

Person^ p = ref new Person("Clark Kent");
p->AddPhoneNumber("Home", "425-555-4567");
p->AddPhoneNumber("Work", "206-555-9999");
String^ workphone = p->PhoneNumbers->Lookup("Work");
```

You can also use stack semantics to declare a local ref class variable. Such an object behaves like a stack-based variable even though the memory is still allocated dynamically. One important difference is that you cannot assign a tracking reference (%) to a variable that is declared by using stack semantics; this guarantees that the reference count is decremented to zero when the function exits. This example shows a basic ref class `Uri`, and a function that uses it with stack semantics:

```
void DoSomething()
{
    Windows::Foundation::Uri docs("http://docs.microsoft.com");
    Windows::Foundation::Uri^ devCenter = docs.CombineUri("/windows/");
    // ...
} // both variables cleaned up here.
```

Memory management

You allocate a ref class in dynamic memory by using the `ref new` keyword.

```
MyRefClass^ myClass = ref new MyRefClass();
```

The handle-to-object operator `^` is known as a *hat* and is fundamentally a C++ smart pointer. The memory it points to is automatically destroyed when the last hat goes out of scope or is explicitly set to `nullptr`.

By definition, a ref class has reference semantics. When you assign a ref class variable, it's the handle that's copied, not the object itself. In the next example, after assignment, both `myClass` and `myClass2` point to the same memory location.

```
MyRefClass^ myClass = ref new MyRefClass();
MyRefClass^ myClass2 = myClass;
```

When a C++/CX ref class is instantiated, its memory is zero-initialized before its constructor is called; therefore it is not necessary to zero-initialize individual members, including properties. If the C++/CX class derives from a Windows Runtime C++ Library (WRL) class, only the C++/CX derived class portion is zero-initialized.

Members

A ref class can contain `public`, `protected`, and `private` function members; only `public` and `protected` members are emitted into metadata. Nested classes and ref classes are permitted but cannot be `public`. Public fields are not allowed; public data members must be declared as properties. Private or protected internal data members may be fields. By default in a ref class, the accessibility of all members is `private`.

A ref struct is the same as a ref class, except that by default its members have `public` accessibility.

A `public` ref class or ref struct is emitted in metadata, but to be usable from other Universal Windows Platform apps and Windows Runtime components it must have at least one public or protected constructor. A public ref class that has a public constructor must also be declared as `sealed` to prevent further derivation through the application binary interface (ABI).

Public members may not be declared as `const` because the Windows Runtime type system does not support const. You can use a static property to declare a public data member with a constant value.

When you define a public ref class or struct, the compiler applies the required attributes to the class and stores that information in the .winmd file of the app. However, when you define a public unsealed ref class, manually apply the `Windows::Foundation::Metadata::WebHostHidden` attribute to ensure that the class is not visible to Universal Windows Platform apps that are written in JavaScript.

A ref class can have standard C++ types, including `const` types, in any `private`, `internal`, or `protected private` members.

Public ref classes that have type parameters are not permitted. User-defined generic ref classes are not permitted. A private, internal, or protected private ref class may be a template.

Destructors

In C++/CX, calling `delete` on a public destructor invokes the destructor regardless of the object's reference count. This behavior enables you to define a destructor that performs custom cleanup of non-RAII resources in a deterministic manner. However, even in this case, the object itself is not deleted from memory. The memory for the object is only freed when the reference count reaches zero.

If a class's destructor is not public, then it is only invoked when the reference count reaches zero. If you call `delete` on an object that has a private destructor, the compiler raises warning C4493, which says "delete expression has no effect as the destructor of <type name> does not have 'public' accessibility."

Ref class destructors can only be declared as follows:

- public and virtual (allowed on sealed or unsealed types)
- protected private and non-virtual (only allowed on unsealed types)
- private and non-virtual (allowed only on sealed types)

No other combination of accessibility, virtualness, and sealedness is allowed. If you do not explicitly declare a destructor, the compiler generates a public virtual destructor if the type's base class or any member has a public destructor. Otherwise, the compiler generates a protected private non-virtual destructor for unsealed types, or a private non-virtual destructor for sealed types.

The behavior is undefined if you try to access members of a class that has already had its destructor run; it will most likely cause the program to crash. Calling `delete t` on a type that has no public destructor has no effect. Calling `delete this` on a type or base class that has a known `private` or `protected private` destructor from within its type hierarchy also has no effect.

When you declare a public destructor, the compiler generates the code so that the ref class implements `Platform::IDisposable` and the destructor implements the `Dispose` method. `Platform::IDisposable` is the C++/CX projection of `Windows::Foundation::IClosable`. Never explicitly implement these interfaces.

Inheritance

Platform::Object is the universal base class for all ref classes. All ref classes are implicitly convertible to Platform::Object and can override Object::ToString. However, the Windows Runtime inheritance model not intended as a general inheritance model; in C++/CX this means that a user-defined public ref class cannot serve as a base class.

If you are creating a XAML user control, and the object participates in the dependency property system, then you can use `Windows::UI::Xaml::DependencyObject` as a base class.

After you have defined an unsealed class `MyBase` that inherits from `DependencyObject`, other public or private ref classes in your component or app may inherit from `MyBase`. Inheritance in public ref classes should only be done to support overrides of virtual methods, polymorphic identity, and encapsulation.

A private base ref class is not required to derive from an existing unsealed class. If you require an object hierarchy to model your own program structure or to enable code reuse, then use private or internal ref classes, or better yet, standard C++ classes. You can expose the functionality of the private object hierarchy through a public sealed ref class wrapper.

A ref class that has a public or protected constructor in C++/CX must be declared as sealed. This restriction means that there is no way for classes that are written in other languages such as C# or Visual Basic to inherit from types that you declare in a Windows Runtime component that's written in C++/CX.

Here are the basic rules for inheritance in C++/CX:

- Ref classes can inherit directly from at most one base ref class, but can implement any number of interfaces.
- If a class has a public constructor, it must be declared as sealed to prevent further derivation.
- You can create public unsealed base classes that have internal or protected private constructors, provided that the base class derives directly or indirectly from an existing unsealed base class such as `Windows::UI::Xaml::DependencyObject`. Inheritance of user-defined ref classes across .winmd files is not supported; however, a ref class can inherit from an interface that's defined in another .winmd file. You can create derived classes from a user-defined base ref class only within the same Windows Runtime component or Universal Windows Platform app.
- For ref classes, only public inheritance is supported.

```
ref class C{};
public ref class D : private C //Error C3628
{};
```

The following example shows how to expose a public ref class that derives from other ref classes in an inheritance hierarchy.

```
namespace InheritanceTest2
{
    namespace WFM = Windows::Foundation::Metadata;

    // Base class. No public constructor.
    [WFM::WebHostHidden]
    public ref class Base : Windows::UI::Xaml::DependencyObject
    {
    internal:
        Base(){}
    protected:
        virtual void DoSomething (){}
        property Windows::UI::Xaml::DependencyProperty^ WidthProperty;
    };

    // Class intended for use by client code across ABI.
    // Declared as sealed with public constructor.
    public ref class MyPublicClass sealed : Base
    {
    public:
        MyPublicClass(){}
        //...
    };
}
```

See also

Type System
Value classes and structs
C++/CX Language Reference
Namespaces Reference

Value classes and structs (C++/CX)

9/21/2022 • 4 minutes to read • Edit Online

A *value struct* or *value class* is a Windows Runtime-compatible POD ("plain old data structure"). It has a fixed size and consists of fields only; unlike a ref class, it has no properties.

The following examples show how to declare and initialize value structs.

```
// in mainpage.xaml.h:
    value struct TestStruct
    {
        Platform::String^ str;
        int i;
    };

    value struct TestStruct2
    {
        TestStruct ts;
        Platform::String^ str;
        int i;
    };

// in mainpage.cpp:
    // Initialize a value struct with an int and String
    TestStruct ts = {"I am a TestStruct", 1};

    // Initialize a value struct that contains
    // another value struct, an int and a String
    TestStruct2 ts2 = {{"I am a TestStruct", 1}, "I am a TestStruct2", 2};

    // Initialize value struct members individually.
    TestStruct ts3;
    ts3.i = 108;
    ts3.str = "Another way to init a value struct.";
```

When a variable of a value type is assigned to another variable, the value is copied, so that each of the two variables has its own copy of the data. A *value struct* is a fixed-size structure that contains only public data fields and is declared by using the `value struct` keyword.

A *value class* is just like a `value struct` except that its fields must be explicitly given public accessibility. It's declared by using the `value class` keyword.

A value struct or value class can contain as fields only fundamental numeric types, enum classes, `Platform::String^`, or Platform::IBox <T> ^ where T is a numeric type or enum class or value class or struct. An `IBox<T>^` field can have a value of `nullptr`—this is how C++ implements the concept of *nullable value types*.

A value class or value struct that contains a `Platform::String^` or `IBox<T>^` type as a member is not `memcpy`-able.

Because all members of a `value class` or `value struct` are public and are emitted into metadata, standard C++ types are not allowed as members. This is different from ref classes, which may contain `private` or `internal` standard C++ types..

The following code fragment declares the `Coordinates` and `City` types as value structs. Notice that one of the `City` data members is a `GeoCoordinates` type. A `value struct` can contain other value structs as members.

```
public enum class Continent
{
    Africa,
    Asia,
    Australia,
    Europe,
    NorthAmerica,
    SouthAmerica,
    Antarctica
};

value struct GeoCoordinates
{
    double Latitude; //or float64 if you prefer
    double Longitude;
};

value struct City
{
    Platform::String^ Name;
    int Population;
    double AverageTemperature;
    GeoCoordinates Coordinates;
    Continent continent;
};
```

Parameter passing for value types

If you have a value type as a function or method parameter, it is normally passed by value. For larger objects, this can cause a performance problem. In Visual Studio 2013 and earlier, value types in C++/CX were always passed by value. In Visual Studio 2015 and later, you can pass value types by reference or by value.

To declare a parameter that passes a value type by value, use code like the following:

```
void Method1(MyValueType obj);
```

To declare a parameter that passes a value type by reference, use the reference symbol (&), as in the following:

```
void Method2(MyValueType& obj);
```

The type inside Method2 is a reference to MyValueType and works the same way as a reference type in standard C++.

When you call Method1 from another language, like C#, you do not need to use the `ref` or `out` keyword. When you call Method2, use the `ref` keyword.

```
Method2(ref obj);
```

You can also use a pointer symbol (*) to pass a value type by reference. The behavior with respect to callers in other languages is the same (callers in C# use the `ref` keyword), but in the method, the type is a pointer to the value type.

Nullable value types

As mentioned earlier, a value class or value struct can have a field of type Platform::IBox<T>^—for example, `IBox<int>^`. Such a field can have any numeric value that is valid for the `int` type, or it can have a value of

`nullptr`. You can pass a nullable field as an argument to a method whose parameter is declared as optional, or anywhere else that a value type is not required to have a value.

The following example shows how to initialize a struct that has a nullable field.

```
public value struct Student
{
    Platform::String^ Name;
    int EnrollmentYear;
    Platform::IBox<int>^ GraduationYear; // Null if not yet graduated.
};
//To create a Student struct, one must populate the nullable type.
MainPage::MainPage()
{
    InitializeComponent();

    Student A;
    A.Name = "Alice";
    A.EnrollmentYear = 2008;
    A.GraduationYear = ref new Platform::Box<int>(2012);

    Student B;
    B.Name = "Bob";
    B.EnrollmentYear = 2011;
    B.GraduationYear = nullptr;

    IsCurrentlyEnrolled(A);
    IsCurrentlyEnrolled(B);
}
bool MainPage::IsCurrentlyEnrolled(Student s)
{
    if (s.GraduationYear == nullptr)
    {
        return true;
    }
    return false;
}
```

A value struct itself may be made nullable in the same way, as shown here:

```
public value struct MyStruct
{
public:
    int i;
    Platform::String^ s;
};

public ref class MyClass sealed
{
public:
    property Platform::IBox<MyStruct>^ myNullableStruct;
};
```

See also

Type System (C++/CX)
C++/CX Language Reference
Namespaces Reference
Ref classes and structs (C++/CX)

Partial classes (C++/CX)

9/21/2022 • 5 minutes to read • Edit Online

A partial class is a construct that supports scenarios in which you are modifying one part of a class definition, and automatic code-generating software—for example, the XAML designer—is also modifying code in the same class. By using a partial class, you can prevent the designer from overwriting your code. In a Visual Studio project, the `partial` modifier is applied automatically to the generated file.

Syntax

To define a partial class, use the `partial` keyword immediately before the class-key of what would otherwise be a normal class definition. A keyword such as `partial ref class` is a contextual keyword that contains whitespace characters. Partial definitions are supported in the following constructs.

- `class` or `struct`
- `ref class` or `ref struct`
- `value class` or `value struct`
- `enum` or `enum class`
- `ref interface`, `interface class`, `interface struct`, or `__interface`
- `union`

This example demonstrates a partial `ref class`:

```
partial ref class MyClass {/* ... */};
```

Contents

A partial class definition can contain anything that the full class definition can contain if the `partial` keyword had been omitted. With one exception, this includes any valid construct such as base classes, data members, member functions, enums, friend declarations, and attributes. And inline definitions of static data members are permitted.

The one exception is class accessibility. For example, the statement `public partial class MyInvalidClass {/* ... */};` is an error. Any access specifiers that are used in a partial class definition for MyInvalidClass don't affect the default accessibility in a subsequent partial or full class definition for MyInvalidClass.

The following code fragment demonstrates accessibility. In the first partial class, `Method1` is public because its accessibility is public. In the second partial class, `Method2` is private because the default class accessibility is private.

```
partial ref class N
{
public:
    int Method1(); // Method1 is public.

};
ref class N
{
    void Method2(); // Method2 is private.
};
```

Declaration

A partial definition of a class such as `MyClass` is only a declaration of MyClass. That is, it only introduces the name `MyClass`. `MyClass` can't be used in a way that requires a class definition, for example, knowing the size of `MyClass` or using a base or member of `MyClass`. `MyClass` is considered to be defined only when the compiler encounters a non-partial definition of `MyClass`.

The following example demonstrates the declaration behavior of a partial class. After declaration #1, `MyClass` can be used as if it were written as the forward declaration, `ref class MyClass;`. Declaration #2 is equivalent to declaration #1. Declaration #3 is valid because it's a forward declaration to a class. But declaration #4 is invalid because

`MyClass` is not fully defined.

Declaration #5 does not use the `partial` keyword, and the declaration fully defines `MyClass`. Consequently, declaration #6 is valid.

```
// Declaration #1
partial ref class MyClass {};

// Declaration #2
partial ref class MyClass;

// Declaration #3
MyClass^ pMc; // OK, forward declaration.

// Declaration #4
MyClass mc; // Error, MyClass is not defined.

// Declaration #5
ref class MyClass { };

// Declaration #6
MyClass mc; // OK, now MyClass is defined.
```

Number and ordering

There can be zero or more partial class definitions for every full definition of a class.

Every partial class definition of a class must lexically precede the one full definition of that class, but doesn't have to precede forward declarations of the class. If there's no full definition of the class, then the partial class declarations can only be forward declarations.

All class-keys such as `class` and `struct` must match. For example, it's an error to code `partial class X {}; struct X {};`.

The following example demonstrates number and ordering. The last partial declaration fails because the class is

already defined.

```
ref class MyClass;  // OK
partial ref class MyClass{};  //OK
partial ref class MyClass{}; // OK
partial ref class MyClass{}; // OK
ref class MyClass{}; // OK
partial ref class MyClass{}; // C3971, partial definition cannot appear after full definition.
```

Full definition

At the point of the full definition of the class X, the behavior is the same as if the definition of X had declared all base classes, members, and so on, in the order in which they were encountered and defined in the partial classes. That is, the contents of the partial classes are treated as though they were written at the point of full definition of the class, and name lookup and other language rules are applied at the point of the full definition of the class as if the contents of the partial classes were written in place

The following two code examples have identical meaning and effect. The first example uses a partial class and the second example doesn't.

```
ref class Base1 { public: property int m_num; int GetNumBase();};
interface class Base2 { int GetNum(); };
interface class Base3{ int GetNum2();};

partial ref class N : public Base1
{
public:
    /*...*/

};

partial ref class N : public Base2
{
public:
    virtual int GetNum();
    // OK, as long as OtherClass is
    //declared before the full definition of N
    void Method2( OtherClass^ oc );
};

ref class OtherClass;

ref class N : public Base3
{
public:
    virtual int GetNum2();
};
```

```
ref class OtherClass;
ref class N : public Base1, public Base2, public Base3
{
public:
    virtual int GetNum();
    virtual int GetNum2();
private:
    void Method2(OtherClass^ oc);

};
```

Templates

A partial class can't be a template.

Restrictions

A partial class can't span beyond one translation unit.

The `partial` keyword is supported only in combination with the `ref class` keyword or the `value class` keyword.

Examples

The following example defines the `Address` class across two code files. The designer modifies `Address.details.h` and you modify `Address.h`. Only the class definition in the first file uses the `partial` keyword.

```
// Address.Details.h
partial ref class Address
{
private:
  Platform::String^ street_;
  Platform::String^ city_;
  Platform::String^ state_;
  Platform::String^ zip_;
  Platform::String^ country_;
  void ValidateAddress(bool normalize = true);
};
```

```
// Address.h
#include "Address.details.h"
ref class Address
{
public:
  Address(Platform::String^ street, Platform::String^ city, Platform::String^ state,
    Platform::String^ zip, Platform::String^ country);
  property Platform::String^ Street { Platform::String^ get(); }
  property Platform::String^ City { Platform::String^ get(); }
  property Platform::String^ State { Platform::String^ get(); }
  property Platform::String^ Zip { Platform::String^ get(); }
  property Platform::String^ Country { Platform::String^ get(); }
};
```

See also

Type System
C++/CX Language Reference
Namespaces Reference

Properties (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

Windows Runtime types expose public data as properties. Client code accesses the property like a public datamember. Internally, the property is implemented as a block that contains a get accessor method, a set accessor method, or both. By using the accessor methods, you can perform additional actions before or after you retrieve the value, for example, you could fire an event or perform validation checks.

Remarks

The value of a property is contained in a private variable—known as the *backing store*—which is the same type as the property. A property can contain both a set accessor, which assigns a value to the backing store, and a get accessor that retrieves the value of the backing store. The property is read-only if it provides only a get accessor, write-only if it provides only a set accessor, and read/write (modifiable) if it provides both accessors.

A *trivial* property is a read/write property for which the compiler automatically implements the accessors and backing store. You don't have access to the compiler's implementation. However, you can declare a custom property and explicitly declare its accessors and backing store. Within an accessor, you can perform any logic that you require, such as validating the input to the set accessor, calculating a value from the property value, accessing a database, or firing an event when the property changes.

When a C++/CX ref class is instantiated, its memory is zero-initialized before its constructor is called; therefore all properties are assigned a default value of zero or nullptr at the point of declaration.

Examples

The following code example shows how to declare and access a property. The first property, `Name`, is known as a *trivial* property because the compiler automatically generates a `set` accessor, `get` accessor, and a backing store.

The second property, `Doctor`, is a read-only property because it specifies a *property block* that explicitly declares only a `get` accessor. Because the property block is declared, you must explicitly declare a backing store; that is, the private String^ variable, `doctor_`. Typically, a read-only property just returns the value of the backing store. Only the class itself can set the value of the backing store, typically in the constructor.

The third property, `Quantity`, is a read-write property because it declares a property block that declares both a `set` accessor and a `get` accessor.

The `set` accessor performs a user-defined validity test on the assigned value. And unlike C#, here the name *value* is just the identifier for the parameter in the `set` accessor; it's not a keyword. If *value* isn't greater than zero, Platform::InvalidArgumentException is thrown. Otherwise, the backing store, `quantity_`, is updated with the assigned value.

Note that a property cannot be initialized in a member list. You can of course initialize backing store variables in a member list.

```
public ref class Prescription sealed
{
private:
    Platform::String^ m_doctor;
    int quantity;
public:
    Prescription(Platform::String^ name, Platform::String^ d) : m_doctor(d)
    {
        // Trivial properties can't be initialized in member list.
        Name = name;
    }

    // Trivial property
    property Platform::String^ Name;

    // Read-only property
    property Platform::String^ Doctor
    {
        Platform::String^ get() { return m_doctor; }
    }

    // Read-write property
    property int Quantity
    {
        int get() { return quantity; }
        void set(int value)
        {
            if (value <= 0)
            {
                throw ref new Platform::InvalidArgumentException();
            }
            quantity = value;
        }
    }
};

public ref class PropertyConsumer sealed
{
private:
    void GetPrescriptions()
    {
        Prescription^ p = ref new Prescription("Louis", "Dr. Who");
        p->Quantity = 5;
        Platform::String^ s = p->Doctor;
        int32 i = p->Quantity;

        Prescription p2("JR", "Dr. Dat");
        p2.Quantity = 10;
    }
};
```

See also

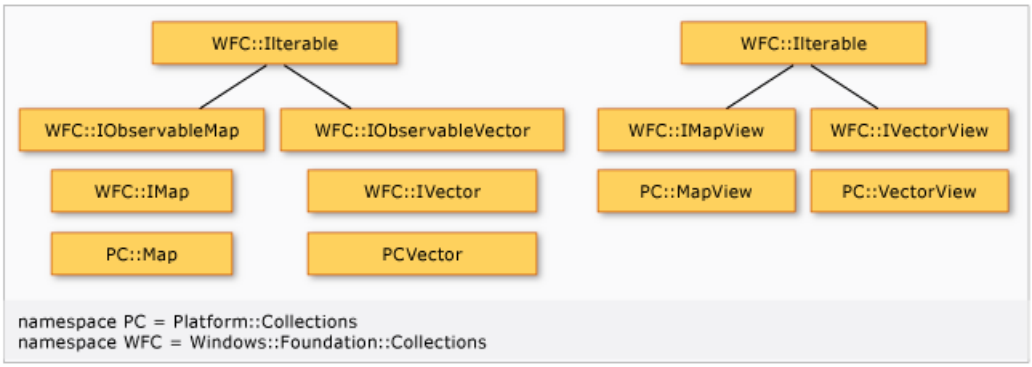
Type System
C++/CX Language Reference
Namespaces Reference

Collections (C++/CX)

9/21/2022 • 10 minutes to read • Edit Online

In a C++/CX program, you can make free use of Standard Template Library (STL) containers, or any other user-defined collection type. However, when you pass collections back and forth across the Windows Runtime application binary interface (ABI)—for example, to a XAML control or to a JavaScript client—you must use Windows Runtime collection types.

The Windows Runtime defines the interfaces for collections and related types, and C++/CX provides the concrete C++ implementations in the collection.h header file. This illustration shows the relationships between the collection types:



- The Platform::Collections::Vector class resembles the std::vector class.
- The Platform::Collections::Map Class class resembles the std::map class.
- Platform::Collections::VectorView Class andPlatform::Collections::MapView Class are read-only versions of `Vector` and `Map`.
- Iterators are defined in the Platform::Collections Namespace. These iterators satisfy the requirements for STL iterators and enable the use of std::find, std::count_if, and other STL algorithms on any Windows::Foundation::Collections interface type or Platform::Collections concrete type. For example, this means that you can iterate a collection in a Windows Runtime component that's created in C# and apply an STL algorithm to it.

  > **IMPORTANT**
  > Proxy iterators `VectorIterator` and `VectorViewIterator` utilize proxy objects `VectoryProxy<T>` and `ArrowProxy<T>` to enable usage with STL containers. For more information, see "VectorProxy elements" later in this article.

- The C++/CX collection types support the same thread safety guarantees that STL containers support.
- Windows::Foundation::Collections::IObservableVector and Windows::Foundation::Collections::IObservableMap define events that are fired when the collection changes in various ways. By implementing these interfaces, Platform::Collections::Map and Platform::Collections::Vector support databinding with XAML collections. For example, if you have a `Vector` that is data-bound to a `Grid`, when you add an item to a collection, the change is reflected in the Grid UI.

Vector usage

When your class has to pass a sequence container to another Windows Runtime component, use Windows::Foundation::Collections:: IVector<T> as the parameter or return type, and Platform::Collections::Vector<T> as the concrete implementation. If you attempt to use a `Vector` type in a public return value or parameter, compiler error C3986 will be raised. You can fix the error by changing the `Vector` to an `IVector`.

> **IMPORTANT**
>
> If you are passing a sequence within your own program, then use either `Vector` or `std::vector` because they are more efficient than `IVector`. Use `IVector` only when you pass the container across the ABI.
>
> The Windows Runtime type system does not support the concept of jagged arrays and therefore you cannot pass an `IVector<Platform::Array<T>>` as a return value or method parameter. To pass a jagged array or a sequence of sequences across the ABI, use `IVector<IVector<T>^>`.

`Vector<T>` provides the methods that are required for adding, removing, and accessing items in the collection, and it is implicitly convertible to `IVector<T>`. You can also use STL algorithms on instances of `Vector<T>`. The following example demonstrates some basic usage. The begin function and end function here are from the `Platform::Collections` namespace, not the `std` namespace.

```
#include <collection.h>
#include <algorithm>
using namespace Platform;
using namespace Platform::Collections;
using namespace Windows::Foundation::Collections;


void Class1::Test()
{
    Vector<int>^ vec = ref new Vector<int>();
    vec->Append(1);
    vec->Append(2);
    vec->Append(3);
    vec->Append(4);
    vec->Append(5);


    auto it =
        std::find(begin(vec), end(vec), 3);

    int j = *it; //j = 3
    int k = *(it + 1); //or it[1]

    // Find a specified value.
    unsigned int n;
    bool found = vec->IndexOf(4, &n); //n = 3

    // Get the value at the specified index.
    n = vec->GetAt(4); // n = 3

    // Insert an item.
    // vec = 0, 1, 2, 3, 4, 5
    vec->InsertAt(0, 0);

    // Modify an item.
    // vec = 0, 1, 2, 12, 4, 5,
    vec->SetAt(3, 12);

    // Remove an item.
    //vec = 1, 2, 12, 4, 5
    vec->RemoveAt(0);

    // vec = 1, 2, 12, 4
    vec->RemoveAtEnd();

    // Get a read-only view into the vector.
    IVectorView<int>^ view = vec->GetView();
}
```

If you have existing code that uses `std::vector` and you want to reuse it in a Windows Runtime component, just use one of the `Vector` constructors that takes a `std::vector` or a pair of iterators to construct a `Vector` at the point where you pass the collection across the ABI. The following example shows how to use the `Vector` move constructor for efficient initialization from a `std::vector`. After the move operation, the original `vec` variable is no longer valid.

```
//#include <collection.h>
//#include <vector>
//#include <utility> //for std::move
//using namespace Platform::Collections;
//using namespace Windows::Foundation::Collections;
//using namespace std;
IVector<int>^ Class1::GetInts()
{
    vector<int> vec;
    for(int i = 0; i < 10; i++)
    {
        vec.push_back(i);
    }
    // Implicit conversion to IVector
    return ref new Vector<int>(std::move(vec));
}
```

If you have a vector of strings that you must pass across the ABI at some future point, you must decide whether to create the strings initially as `std::wstring` types or as `Platform::String^` types. If you have to do a lot of processing on the strings, then use `wstring`. Otherwise, create the strings as `Platform::String^` types and avoid the cost of converting them later. You must also decide whether to put these strings into a `std:vector` or `Platform::Collections::Vector` internally. As a general practice, use `std::vector` and then create a `Platform::Vector` from it only when you pass the container across the ABI.

Value types in Vector

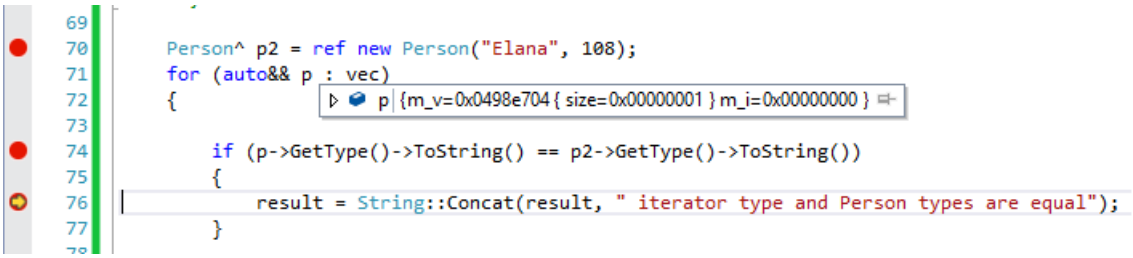
Any element to be stored in a Platform::Collections::Vector must support equality comparison, either implicitly or by using a custom std::equal_to comparator that you provide. All reference types and all scalar types implicitly support equality comparisons. For non-scalar value types such as Windows::Foundation::DateTime, or for custom comparisons—for example, `objA->UniqueID == objB->UniqueID`—you must provide a custom function object.

VectorProxy elements

Platform::Collections::VectorIterator and Platform::Collections::VectorViewIterator enable the use of `range for` loops and algorithms like std::sort with an IVector<T> container. But `IVector` elements cannot be accessed through C++ pointer dereference; they can be accessed only through GetAt and SetAt methods. Therefore, these iterators use the proxy classes `Platform::Details::VectorProxy<T>` and `Platform::Details::ArrowProxy<T>` to provide access to the individual elements through *, ->, and [] operators, as required by the Standard Library. Strictly speaking, given an `IVector<Person^> vec`, the type of `*begin(vec)` is `VectorProxy<Person^>`. However, the proxy object is almost always transparent to your code. These proxy objects are not documented because they are only for internal use by the iterators, but it is useful to know how the mechanism works.

When you use a range-based `for` loop over `IVector` containers, use `auto&&` to enable the iterator variable to bind correctly to the `VectorProxy` elements. If you use `auto&`, compiler warning C4239 is raised and `VectoryProxy` is mentioned in the warning text.

The following illustration shows a `range for` loop over an `IVector<Person^>`. Notice that execution is stopped on the breakpoint on line 64. The **QuickWatch** window shows that the iterator variable `p` is in fact a `VectorProxy<Person^>` that has `m_v` and `m_i` member variables. However, when you call `GetType` on this variable, it returns the identical type to the `Person` instance `p2`. The takeaway is that although `VectorProxy` and `ArrowProxy` might appear in **QuickWatch**, the debugger certain compiler errors, or other places, you typically don't have to explicitly code for them.



One scenario in which you have to code around the proxy object is when you have to perform a `dynamic_cast` on the elements—for example, when you are looking for XAML objects of a particular type in a `UIElement` element collection. In this case, you must first cast the element to Platform::Object^ and then perform the dynamic cast:

```
void FindButton(UIElementCollection^ col)
{
    // Use auto&& to avoid warning C4239
    for (auto&& elem : col)
    {
        Button^ temp = dynamic_cast<Button^>(static_cast<Object^>(elem));
        if (nullptr != temp)
        {
            // Use temp...
        }
    }
}
```

Map usage

This example shows how to insert items and look them up in a Platform::Collections::Map, and then return the `Map` as a read-only Windows::Foundation::Collections::IMapView type.

```
//#include <collection.h>
//using namespace Platform::Collections;
//using namespace Windows::Foundation::Collections;
IMapView<String^, int>^ Class1::MapTest()
{
    Map<String^, int>^ m = ref new Map<String^, int >();
    m->Insert("Mike", 0);
    m->Insert("Dave", 1);
    m->Insert("Doug", 2);
    m->Insert("Nikki", 3);
    m->Insert("Kayley", 4);
    m->Insert("Alex", 5);
    m->Insert("Spencer", 6);

   // PC::Map does not support [] operator
   int i = m->Lookup("Doug");

   return m->GetView();

}
```

In general, for internal map functionality, prefer the `std::map` type for performance reasons. If you have to pass the container across the ABI, construct a Platform::Collections::Map from the std::map and return the `Map` as an Windows::Foundation::Collections::IMap. If you attempt to use a `Map` type in a public return value or parameter, compiler error C3986 will be raised. You can fix the error by changing the `Map` to an `IMap`. In some cases—for example, if you are not making a large number of lookups or insertions, and you are passing the collection across the ABI frequently—it might be less expensive to use `Platform::Collections::Map` from the beginning and avoid the cost of converting the `std::map`. In any case, avoid lookup and insert operations on an `IMap`

because these are the least performant of the three types. Convert to `IMap` only at the point that you pass the container across the ABI.

Value types in Map

Elements in a Platform::Collections::Map are ordered. Any element to be stored in a `Map` must support less-than comparison with strict weak ordering, either implicitly or by using a custom stl::less comparator that you provide. Scalar types support the comparison implicitly. For non-scalar value types such as `Windows::Foundation::DateTime`, or for custom comparisons—for example, `objA->UniqueID < objB->UniqueID`—you must provide a custom comparator.

Collection types

Collections fall into four categories: modifiable versions and read-only versions of sequence collections and associative collections. In addition, C++/CX enhances collections by providing three iterator classes that simplify the accessing of collections.

Elements of a modifiable collection can be changed, but elements of a read-only collection, which is known as a *view*, can only be read. Elements of a Platform::Collections::Vector orPlatform::Collections::VectorView collection can be accessed by using an iterator or the collection's Vector::GetAt and an index. Elements of an associative collection can be accessed by using the collection's Map::Lookup and a key.

Platform::Collections::Map Class
A modifiable, associative collection. Map elements are key-value pairs. Looking up a key to retrieve its associated value, and iterating through all key-value pairs, are both supported.

`Map` and `MapView` are templated on `<K, V, C = std::less<K>>`; therefore, you can customize the comparator. Additionally, `Vector` and `VectorView` are templated on `<T, E = std::equal_to<T>>` so that you can customize the behavior of `IndexOf()`. This is important mostly for `Vector` and `VectorView` of value structs. For example, to create a Vector<Windows::Foundation::DateTime>, you must provide a custom comparator because DateTime does not overload the == operator.

Platform::Collections::MapView Class
A read-only version of a `Map`.

Platform::Collections::Vector Class
A modifiable sequence collection. `Vector<T>` supports constant-time random access and amortized-constant-time Append operations..

Platform::Collections::VectorView Class
A read-only version of a `Vector`.

Platform::Collections::InputIterator Class
An STL iterator that satisfies the requirements of an STL input iterator.

Platform::Collections::VectorIterator Class
An STL iterator that satisfies the requirements of an STL mutable random-access iterator.

Platform::Collections::VectorViewIterator Class
An STL iterator that satisfies the requirements of an STL `const` random-access iterator.

begin() and end() functions

To simplify the use of the STL to process `Vector`, `VectorView`, `Map`, `MapView`, and arbitrary `Windows::Foundation::Collections` objects, C++/CX supports overloads of the begin Function and end Function non-member functions.

The following table lists the available iterators and functions.

ITERATORS	FUNCTIONS
Platform::Collections::VectorIterator<T> (Internally stores Windows::Foundation::Collections::IVector<T> and int.)	begin/ end(Windows::Foundation::Collections:: IVector<T>)
Platform::Collections::VectorViewIterator<T> (Internally stores IVectorView<T>^ and int.)	begin/ end (IVectorView<T>^)
Platform::Collections::InputIterator<T> (Internally stores IIterator<T>^ and T.)	begin/ end (IIterable<T>)
Platform::Collections::InputIterator<IKeyValuePair<K, V>^> (Internally stores IIterator<T>^ and T.)	begin/ end (IMap<K,V>.
Platform::Collections::InputIterator<IKeyValuePair<K, V>^> (Internally stores IIterator<T>^ and T.)	begin/ end (Windows::Foundation::Collections::IMapView)

Collection change events

`Vector` and `Map` support databinding in XAML collections by implementing events that occur when a collection object is changed or reset, or when any element of a collection is inserted, removed, or changed. You can write your own types that support databinding, although you cannot inherit from `Map` or `Vector` because those types are sealed.

The Windows::Foundation::Collections::VectorChangedEventHandler and Windows::Foundation::Collections::MapChangedEventHandler delegates specify the signatures for event handlers for collection change events. The Windows::Foundation::Collections::CollectionChange public enum class, and `Platform::Collection::Details::MapChangedEventArgs` and `Platform::Collections::Details::VectorChangedEventArgs` ref classes, store the event arguments to determine what caused the event. The `*EventArgs` types are defined in the `Details` namespace because you don't have to construct or consume them explicitly when you use `Map` or `Vector`.

See also

Type System
C++/CX Language Reference
Namespaces Reference

Template ref classes (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

C++ templates are not published to metadata and therefore cannot have public or protected accessibility in your program. You can, of course, use standard C++ templates internally in your program. In addition, you can define a private ref class as a template and you can declare an explicitly specialized template ref class as a private member in a public ref class.

Authoring ref class templates

The following example shows how to declare a private ref class as a template, and also how to declare a standard C++ template and how declare them both as members in a public ref class. Note that the standard C++ template can be specialized by a Windows Runtime type, in this case a Platform::String^.

```
namespace TemplateDemo
{
    // A private ref class template
    template <typename T>
    ref class MyRefTemplate
    {
    internal:
        MyRefTemplate(T d) : data(d){}
    public:
        T Get(){ return data; }
    private:
        T data;
    };

    // Specialization of ref class template
    template<>
    ref class MyRefTemplate<Platform::String^>
    {
    internal:
        //...
    };

    // A private derived ref class that inherits
    // from a ref class template specialization
    ref class MyDerivedSpecialized sealed : public MyRefTemplate<int>
    {
    internal:
        MyDerivedSpecialized() : MyRefTemplate<int>(5){}
    };

    // A private derived template ref class
    // that inherits from a ref class template
    template <typename T>
    ref class MyDerived : public MyRefTemplate<T>
    {
    internal:
        MyDerived(){}
    };

    // A standard C++ template
    template <typename T>
    class MyStandardTemplate
    {
    public:
        MyStandardTemplate(){}
        T Get() { return data; }
    private:
        T data;

    };

    // A public ref class with private
    // members that are specializations of
    // ref class templates and standard C++ templates.
    public ref class MySpecializeBoth sealed
    {
    public:
        MySpecializeBoth(){}
    private:
        MyDerivedSpecialized^ g;
        MyStandardTemplate<Platform::String^>* n;
    };
}
```

See also

Type System (C++/CX)
C++/CX Language Reference
Namespaces Reference

Interfaces (C++/CX)

9/21/2022 • 6 minutes to read • Edit Online

Although a ref class can inherit from at most one concrete base class, it can implement any number of interface classes. An interface class (or interface struct) itself can inherit (or require) multiple interface classes, can overload its member functions, and can have type parameters.

Characteristics

An interface has these characteristics:

- An interface class (or struct) must be declared within a namespace and may have public or private accessibility. Only public interfaces are emitted to metadata.
- The members of an interface can include properties, methods, and events.
- All interface members are implicitly public and virtual.
- Fields and static members are not permitted.
- Types that are used as properties, method parameters, or return values can only be Windows Runtime types; this includes the fundamental types and enum class types.

Declaration and usage

The following example shows how to declare an interface. Notice that an interface can be declared as either a class or struct type.

```
namespace InterfacesTest
{
    public enum class PlayState {Playing, Paused, Stopped, Forward, Reverse};

    public ref struct MediaPlayerEventArgs sealed
    {
        property PlayState oldState;
        property PlayState newState;
    };

    public delegate void OnStateChanged(Platform::Object^ sender, MediaPlayerEventArgs^ a);
    public interface class IMediaPlayer // or public interface struct IMediaPlayer
    {
        event OnStateChanged^ StateChanged;
        property Platform::String^ CurrentTitle;
        property PlayState CurrentState;
        void Play();
        void Pause();
        void Stop();
        void Back(float speed);
        void Forward(float speed);
    };
}
```

To implement an interface, a ref class or ref struct declares and implements virtual methods and properties. The interface and the implementing ref class must use the same method parameter names, as shown in this example:

```
public ref class MyMediaPlayer sealed : public IMediaPlayer
{
public:
    //IMediaPlayer
    virtual event OnStateChanged^ StateChanged;
    virtual property Platform::String^ CurrentTitle;
    virtual property PlayState CurrentState;
    virtual void Play()
    {
        // ...
        auto args = ref new MediaPlayerEventArgs();
        args->newState = PlayState::Playing;
        args->oldState = PlayState::Stopped;
        StateChanged(this, args);
    }
    virtual void Pause(){/*...*/}
    virtual void Stop(){/*...*/}
    virtual void Forward(float speed){/*...*/}
    virtual void Back(float speed){/*...*/}
private:
    //...
};
```

Interface inheritance hierarchies

An interface can inherit from one or more interfaces. But unlike a ref class or struct, an interface doesn't declare the inherited interface members. If interface B inherits from interface A, and ref class C inherits from B, C must implement both A and B. This is shown in the next example.

```
public interface struct A { void DoSomething(); };
public interface struct B : A { void DoSomethingMore();};

public ref struct C sealed : B
{
    virtual void DoSomething(){}
    virtual void DoSomethingMore(){}
};
```

Implementing interface properties and events

As shown in the previous example, you can use trivial virtual properties to implement interface properties. You can also provide custom getters and setters in the implementing class. Both the getter and the setter must be public in an interface property.

```
//Alternate implementation in MediaPlayer class of IMediaPlayer::CurrentTitle
virtual property Platform::String^ CurrentTitle
{
    Platform::String^ get() {return "Now playing: " + _title;}
    void set(Platform::String^ t) {_title = t; }
}
```

If an interface declares a get-only or set-only property, then the implementing class should explicitly provide a getter or setter.

```
public interface class IMediaPlayer
{
    //...
    property Platform::String^ CurrentTitle
    {
        Platform::String^ get();
    }
};

public ref class MyMediaPlayer3 sealed : public IMediaPlayer
{
public:
    //...
    virtual property Platform::String^ CurrentTitle
    {
        Platform::String^ get() {return "Now playing: " + _title;}
    }
private:
    Platform::String^ _title;
};
```

You can also implement custom add and remove methods for events in the implementing class.

Explicit interface implementation

When a ref class implements multiple interfaces, and those interfaces have methods whose names and signatures are identical to the compiler, you can use the following syntax to explicitly indicate the interface method that a class method is implementing.

```
public interface class IArtist
{
    Platform::String^ Draw();
};

public interface class ICowboy
{
    Platform::String^ Draw();
};

public ref class MyClass sealed : public IArtist, ICowboy
{
public:
    MyClass(){}
    virtual  Platform::String^ ArtistDraw() = IArtist::Draw {return L"Artist";}
    virtual  Platform::String^ CowboyDraw() = ICowboy::Draw {return L"Cowboy";}
};
```

Generic interfaces

In C++/CX, the `generic` keyword is used to represent a Windows Runtime parameterized type. A parameterized type is emitted in metadata and can be consumed by code that's written in any language that supports type parameters. The Windows Runtime defines some generic interfaces—for example, Windows::Foundation::Collections::IVector<T>—but it doesn't support the creation of public user-defined generic interfaces in C++/CX. However, you can create private generic interfaces.

Here's how Windows Runtime types can be used to author a generic interface:

- A generic user-defined `interface class` in a component is not allowed to be emitted into its Windows metadata file; therefore, it can't have public accessibility, and client code in other .winmd files can't implement it. It can be implemented by non-public ref classes in the same component. A public ref class

can have a generic interface type as a private member.

The following code snippet shows how to declare a generic `interface class` and then implement it in a private ref class and use the ref class as a private member in a public ref class.

```
public ref class MediaFile sealed {};

generic <typename T>
private interface class  IFileCollection
{
    property Windows::Foundation::Collections::IVector<T>^ Files;
    Platform::String^  GetFileInfoAsString(T file);
};

private ref class MediaFileCollection : IFileCollection<MediaFile^>
{
public:
    virtual property Windows::Foundation::Collections::IVector<MediaFile^>^ Files;
    virtual Platform::String^  GetFileInfoAsString(MediaFile^ file){return "";}
};

public interface class ILibraryClient
{
    bool FindTitle(Platform::String^ title);
    //...
};

public ref class MediaPlayer sealed : public IMediaPlayer, public ILibraryClient
{
public:
    //IMediaPlayer
    virtual event OnStateChanged^ StateChanged;
    virtual property Platform::String^ CurrentTitle;
    virtual property PlayState CurrentState;
    virtual void Play()
    {
        auto args = ref new MediaPlayerEventArgs();
        args->newState = PlayState::Playing;
        args->oldState = PlayState::Stopped;
        StateChanged(this, args);
    }
    virtual void Pause(){/*...*/}
    virtual void Stop(){/*...*/}
    virtual void Forward(float speed){/*...*/}
    virtual void Back(float speed){/*...*/}

    //ILibraryClient
    virtual bool FindTitle(Platform::String^ title){/*...*/ return true;}

private:
    MediaFileCollection^ fileCollection;

};
```

- A generic interface must follow the standard interface rules that govern accessibility, members, *requires* relationships, base classes, and so on.
- A generic interface can take one or more generic type parameters that are preceded by `typename` or `class`. Non-type parameters are not supported.
- A type parameter can be any Windows Runtime type. That is, the type parameter can be a reference type, a value type, an interface class, a delegate, a fundamental type, or a public enum class.
- A *closed generic interface* is an interface that inherits from a generic interface and specifies concrete type arguments for all type parameters. It can be used anywhere that a non-generic private interface can be

used.

- An *open generic interface* is an interface that has one or more type parameters for which no concrete type is yet provided. It can be used anywhere that a type can be used, including as a type argument of another generic interface.
- You can parameterize only an entire interface, not individual methods.
- Type parameters cannot be constrained.
- A closed generic interface has an implicitly generated UUID. A user cannot specify the UUID.
- In the interface, any reference to the current interface—in a method parameter, return value, or property—is assumed to refer to the current instantiation. For example, *IMyIntf* means *IMyIntf<T>*.
- When the type of a method parameter is a type parameter, the declaration of that parameter or variable uses the type parameter's name without any pointer, native reference, or handle declarators. In other words, you never write "T^".
- Templated ref classes must be private. They can implement generic interfaces, and can pass template parameter *T* to generic argument *T*. Each instantiation of a templated ref class is itself a ref class.

See also

Type System
C++/CX Language Reference
Namespaces Reference

Enums (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

C++/CX supports the `public enum class` keyword, which is analogous to a standard C++ `scoped enum`. When you use an enumerator that's declared by using the `public enum class` keyword, you must use the enumeration identifier to scope each enumerator value.

Remarks

A `public enum class` that doesn't have an access specifier, such as `public`, is treated as a standard C++ scoped enum.

A `public enum class` or `public enum struct` declaration can have an underlying type of any integral type although the Windows Runtime itself requires that the type be int32, or uint32 for a flags enum. The following syntax describes the parts of an `public enum class` or `public enum struct`.

This example shows how to define a public enum class:

```
// Define the enum
public enum class TrafficLight : int { Red, Yellow, Green };
// ...
```

This next example shows how to consume it:

```
// Consume the enum:
TrafficLight myLight = TrafficLight::Red;
if (myLight == TrafficLight::Green)
{
    //...
}
```

Examples

The next examples show how to declare an enum,

```
// Underlying type is int32
public enum class Enum1
{
    Zero,
    One,
    Two,
    Three
};

public enum class Enum2
{
    None = 0,
    First,      // First == 1
    Some = 5,
    Many = 10
};

// Underlying type is unsigned int
// for Flags. Must be explicitly specified
using namespace Platform::Metadata;
[Flags]
public enum class BitField : unsigned int
{
    Mask0 = 0x0,
    Mask2 = 0x2,
    Mask4 = 0x4,
    Mask8 = 0x8
};

Enum1 e1 = Enum1::One;
int v1 = static_cast<int>(e1);
int v2 = static_cast<int>(Enum2::First);
```

The next example shows how to cast to numeric equivalents, and perform comparisons. Notice that the use of enumerator `One` is scoped by the `Enum1` enumeration identifier, and enumerator `First` is scoped by `Enum2`.

```
if (e1 == Enum1::One) { /* ... */ }
//if (e1 == Enum2::First) { /* ... */ } // yields compile error C3063

static_assert(sizeof(Enum1) == 4, "sizeof(Enum1) should be 4");

BitField x = BitField::Mask0 | BitField::Mask2 | BitField::Mask4;
if ((x & BitField::Mask2) == BitField::Mask2) { /*   */ }
```

See also

Type System
C++/CX Language Reference
Namespaces Reference

Delegates (C++/CX)

9/21/2022 • 9 minutes to read • Edit Online

The `delegate` keyword is used to declare a reference type that is the Windows Runtime equivalent of a function object in standard C++. A delegate declaration similar to a function signature; it specifies the return type and parameter types that its wrapped function must have. This is a user-defined delegate declaration:

```
public delegate void PrimeFoundHandler(int result);
```

Delegates are most commonly used in conjunction with events. An event has a delegate type, in much the same way that a class can have an interface type. The delegate represents a contract that event handlers much fulfill. Here's an event class member whose type is the previously-defined delegate:

```
event PrimeFoundHandler^ primeFoundEvent;
```

When declaring delegates that will be exposed to clients across the Windows Runtime application binary interface, use Windows::Foundation::TypedEventHandler<TSender, TResult>. This delegate has predefined proxy and stub binaries that enable it to be consumed by Javascript clients.

Consuming delegates

When you create a Universal Windows Platform app, you often work with a delegate as the type of an event that a Windows Runtime class exposes. To subscribe to an event, create an instance of its delegate type by specifying a function—or lambda—that matches the delegate signature. Then use the `+=` operator to pass the delegate object to the event member on the class. This is known as subscribing to the event. When the class instance "fires" the event, your function is called, along with any other handlers that were added by your object or other objects.

> **TIP**
> Visual Studio does a lot of work for you when you create an event handler. For example, if you specify an event handler in XAML markup, a tool tip appears. If you choose the tool tip, Visual Studio automatically creates the event handler method and associates it with the event on the publishing class.

The following example shows the basic pattern. `Windows::Foundation::TypedEventHandler` is the delegate type. The handler function is created by using a named function.

In app.h:

```
[Windows::Foundation::Metadata::WebHostHiddenAttribute]
ref class App sealed
{
    void InitializeSensor();
    void SensorReadingEventHandler(Windows::Devices::Sensors::LightSensor^ sender,
        Windows::Devices::Sensors::LightSensorReadingChangedEventArgs^ args);

    float m_oldReading;
    Windows::Devices::Sensors::LightSensor^ m_sensor;

};
```

In app.cpp:

```
void App::InitializeSensor()
{
    // using namespace Windows::Devices::Sensors;
    // using namespace Windows::Foundation;
    m_sensor = LightSensor::GetDefault();

    // Create the event handler delegate and add
    // it  to the object's  event handler list.
    m_sensor->ReadingChanged += ref new  TypedEventHandler<LightSensor^,
        LightSensorReadingChangedEventArgs^>( this,
        &App::SensorReadingEventHandler);

}

void App::SensorReadingEventHandler(LightSensor^ sender,
                                    LightSensorReadingChangedEventArgs^ args)
{
    LightSensorReading^ reading = args->Reading;
    if (reading->IlluminanceInLux > m_oldReading)
    {/*...*/}

}
```

WARNING

In general, for an event handler, it's better to use a named function instead of a lambda unless you take great care to avoid circular references. A named function captures the "this" pointer by weak reference, but a lambda captures it by strong reference and creates a circular reference. For more information, see Weak references and breaking cycles.

By convention, event-handler delegate names that are defined by the Windows Runtime have the form *EventHandler—for example, RoutedEventHandler, SizeChangedEventHandler, or SuspendingEventHandler. Also by convention, event handler delegates have two parameters and return void. In a delegate that doesn't have type parameters, the first parameter is of type Platform::Object^; it holds a reference to the sender, which is the object that fired the event. You have to cast back to the original type before you use the argument in the event handler method. In an event handler delegate that has type parameters, the first type parameter specifies the type of the sender, and the second parameter is a handle to a ref class that holds information about the event. By convention, that class is named *EventArgs. For example, a RoutedEventHandler delegate has a second parameter of type RoutedEventArgs^, and DragEventHander has a second parameter of type DragEventArgs^.

By convention, delegates that wrap the code that executes when an asynchronous operation completes are named *CompletedHandler. These delegates are defined as properties on the class, not as events. Therefore, you don't use the `+=` operator to subscribe to them; you just assign a delegate object to the property.

TIP

C++ IntelliSense doesn't show the full delegate signature; therefore, it doesn't help you determine the specific type of the EventArgs parameter. To find the type, you can go to the **Object Browser** and look at the `Invoke` method for the delegate.

Creating custom delegates

You can define your own delegates, to define event handlers or to enable consumers to pass in custom functionality to your Windows Runtime component. Like any other Windows Runtime type, a public delegate cannot be declared as generic.

Declaration

The declaration of a delegate resembles a function declaration except that the delegate is a type. Typically, you declare a delegate at namespace scope, although you can also nest a delegate declaration in a class declaration. The following delegate encapsulates any function that takes a `ContactInfo^` as input and returns a `Platform::String^`.

```
public delegate Platform::String^ CustomStringDelegate(ContactInfo^ ci);
```

After you declare a delegate type, you can declare class members of that type or methods that take objects of that type as parameters. A method or function can also return a delegate type. In the following example, the `ToCustomString` method takes the delegate as an input parameter. The method enables client code to provide a custom function that constructs a string from some or all of the public properties of a `ContactInfo` object.

```
public ref class ContactInfo sealed
{
public:
    ContactInfo(){}
    ContactInfo(Platform::String^ saluation, Platform::String^ last, Platform::String^ first,
Platform::String^ address1);
    property Platform::String^ Salutation;
    property Platform::String^ LastName;
    property Platform::String^ FirstName;
    property Platform::String^ Address1;
    //...other properties

    Platform::String^ ToCustomString(CustomStringDelegate^ func)
    {
        return func(this);
    }
};
```

> **NOTE**
>
> You use the "^" symbol when you refer to the delegate type, just as you do with any Windows Runtime reference type.

An event declaration always has a delegate type. This example shows a typical delegate type signature in the Windows Runtime:

```
public delegate void RoutedEventHandler(
    Platform::Object^ sender,
    Windows::UI::Xaml::RoutedEventArgs^ e
    );
```

The `Click` event in the `Windows:: UI::Xaml::Controls::Primitives::ButtonBase` class is of type `RoutedEventHandler`. For more information, see Events.

Client code first constructs the delegate instance by using `ref new` and providing a lambda that's compatible with the delegate signature and defines the custom behavior.

```
CustomStringDelegate^ func = ref new CustomStringDelegate([] (ContactInfo^ c)
{
    return c->FirstName + " " + c->LastName;
});
```

It then calls the member function and passes the delegate. Assume that `ci` is a `ContactInfo^` instance and

`textBlock` is a XAML `TextBlock^`.

```
textBlock->Text = ci->ToCustomString( func );
```

In the next example, a client app passes a custom delegate to a public method in a Windows Runtime component that executes the delegate against each item in a `Vector`:

```
//Client app
obj = ref new DelegatesEvents::Class1();

CustomStringDelegate^ myDel = ref new CustomStringDelegate([] (ContactInfo^ c)
{
    return c->Salutation + " " + c->LastName;
});
IVector<String^>^ mycontacts = obj->GetCustomContactStrings(myDel);
std::for_each(begin(mycontacts), end(mycontacts), [this] (String^ s)
{
    this->ContactString->Text += s + " ";
});
```

```
// Public method in WinRT component.
IVector<String^>^ Class1::GetCustomContactStrings(CustomStringDelegate^ del)
{
    namespace WFC = Windows::Foundation::Collections;

    Vector<String^>^ contacts = ref new Vector<String^>();
    VectorIterator<ContactInfo^> i = WFC::begin(m_contacts);
    std::for_each( i ,WFC::end(m_contacts), [contacts, del](ContactInfo^ ci)
    {
        contacts->Append(del(ci));
    });

    return contacts;
}
```

Construction

You can construct a delegate from any of these objects:

- lambda
- static function
- pointer-to-member
- std::function

The following example shows how to construct a delegate from each of these objects. You consume the delegate in exactly the same way regardless of the type of object that's used to construct it.

```
ContactInfo^ ci = ref new ContactInfo("Mr.", "Michael", "Jurek", "1234 Compiler Way");

// Lambda. (Avoid capturing "this" or class members.)
CustomStringDelegate^ func = ref new CustomStringDelegate([] (ContactInfo^ c)
{
    return c->Salutation + " " + c->FirstName + " " + c->LastName;
});

// Static function.
// static Platform::String^ GetFirstAndLast(ContactInfo^ info);
CustomStringDelegate^ func2 = ref new CustomStringDelegate(Class1::GetFirstAndLast);


// Pointer to member.
// Platform::String^ GetSalutationAndLast(ContactInfo^ info)
CustomStringDelegate^ func3 = ref new CustomStringDelegate(this,
&DelegatesEvents::Class1::GetSalutationAndLast);

// std::function
std::function<String^ (ContactInfo^)> f = Class1::GetFirstAndLast;
CustomStringDelegate^ func4 = ref new CustomStringDelegate(f);


// Consume the delegates. Output depends on the
// implementation of the functions you provide.
textBlock->Text  = func(ci);
textBlock2->Text = func2(ci);
textBlock3->Text = func3(ci);
textBlock4->Text = func4(ci);
```

WARNING

If you use a lambda that captures the "this" pointer, be sure to use the `-=` operator to explicitly un-register from the event before you exit the lambda. For more information, see Events.

Generic delegates

Generic delegates in C++/CX have restrictions similar to declarations of generic classes. They cannot be declared as public. You can declare a private or internal generic delegate and consume it from C++, but .NET or JavaScript clients can't consume it because it is not emitted into the .winmd metadata. This example declares a generic delegate that can only be consumed by C++:

```
generic <typename T>
delegate void  MyEventHandler(T p1, T p2);
```

The next example declares a specialized instance of the delegate inside a class definition:

```
MyEventHandler<float>^ myDelegate;
```

Delegates and threads

A delegate, just like a function object, contains code that will execute at some time in the future. If the code that creates and passes the delegate, and the function that accepts and executes the delegate, are running on the same thread, then things are relatively simple. If that thread is the UI thread, then the delegate can directly manipulate user interface objects such as XAML controls.

If a client app loads a Windows Runtime component that runs in a threaded apartment, and provides a delegate to that component, then by default the delegate is invoked directly on the STA thread. Most Windows Runtime components can run in either STA or MTA.

If the code that executes the delegate is running on a different thread—for example, within the context of a concurrency::task object—then you are responsible for synchronizing access to shared data. For example, if your delegate contains a reference to a Vector, and a XAML control has a reference to that same Vector, you must take steps to avoid deadlocks or race conditions that might occur when both the delegate and XAML control attempt to access the Vector at the same time. You must also take care that the delegate doesn't attempt to capture by reference local variables that might go out of scope before the delegate is invoked.

If you want your created delegate to be called back on the same thread that it was created on—for example, if you pass it to a component that runs in an MTA apartment—and you want it to be invoked on the same thread as the creator, then use the delegate constructor overload that takes a second `CallbackContext` parameter. Only use this overload on delegates that have a registered proxy/stub; not all of the delegates that are defined in Windows.winmd are registered.

If you are familiar with event handlers in .NET, you know that the recommended practice is to make a local copy of an event before you fire it. This avoids race conditions in which an event handler might be removed just before the event is invoked. It isn't necessary to do this in C++/CX because when event handlers are added or removed a new handler list is created. Because a C++ object increments the reference count on the handler list before invoking an event, it is guaranteed that all handlers will be valid. However, this also means that if you remove an event handler on the consuming thread, that handler might still get invoked if the publishing object is still operating on its copy of the list, which is now out-of-date. The publishing object will not get the updated list until the next time it fires the event.

See also

Type System
C++/CX Language Reference
Namespaces Reference

Exceptions (C++/CX)

9/21/2022 • 5 minutes to read • Edit Online

Error handling in C++/CX is based on exceptions. At the most fundamental level, Windows Runtime components report errors as HRESULT values. In C++/CX, these values are converted to strongly typed exceptions that contain an HRESULT value and a string description that you can access programmatically. Exceptions are implemented as a `ref class` that derives from `Platform::Exception`. The `Platform` namespace defines distinct exception classes for the most common HRESULT values; all other values are reported through the `Platform::COMException` class. All exception classes have an Exception::HResult field that you can use to retrieve the original HRESULT. You can also examine call-stack information for user code in the debugger that can help pinpoint the original source of the exception, even if it originated in code that was written in a language other than C++.

Exceptions

In your C++ program, you can throw and catch an exception that comes from a Windows Runtime operation, an exception that's derived from `std::exception`, or a user-defined type. You have to throw a Windows Runtime exception only when it will cross the application binary interface (ABI) boundary, for example, when the code that catches your exception is written in JavaScript. When a non-Windows Runtime C++ exception reaches the ABI boundary, the exception is translated into a `Platform::FailureException` exception, which represents an E_FAIL HRESULT. For more information about the ABI, see Creating Windows Runtime Components in C++.

You can declare a Platform::Exception by using one of two constructors that take either an HRESULT parameter, or an HRESULT parameter and a Platform::String^ parameter that can be passed across the ABI to any Windows Runtime app that handles it. Or you can declare an exception by using one of two Exception::CreateException method overloads that take either an HRESULT parameter, or an HRESULT parameter and a `Platform::String^` parameter.

Standard exceptions

C++/CX supports a set of standard exceptions that represent typical HRESULT errors. Each standard exception derives from Platform::COMException, which in turn derives from `Platform::Exception`. When you throw an exception across the ABI boundary, you must throw one of the standard exceptions.

You can't derive your own exception type from `Platform::Exception`. To throw a custom exception, use a user-defined HRESULT to construct a `COMException` object.

The following table lists the standard exceptions.

NAME	UNDERLYING HRESULT	DESCRIPTION
COMException	*user-defined hresult*	Thrown when an unrecognized HRESULT is returned from a COM method call.
AccessDeniedException	E_ACCESSDENIED	Thrown when access is denied to a resource or feature.

NAME	UNDERLYING HRESULT	DESCRIPTION
ChangedStateException	E_CHANGED_STATE	Thrown when methods of a collection iterator or a collection view are called after the parent collection has changed, thereby invalidating the results of the method.
ClassNotRegisteredException	REGDB_E_CLASSNOTREG	Thrown when a COM class has not been registered.
DisconnectedException	RPC_E_DISCONNECTED	Thrown when an object is disconnected from its clients.
FailureException	E_FAIL	Thrown when an operation fails.
InvalidArgumentException	E_INVALIDARG	Thrown when one of the arguments that are provided to a method is not valid.
InvalidCastException	E_NOINTERFACE	Thrown when a type can't be cast to another type.
NotImplementedException	E_NOTIMPL	Thrown if an interface method hasn't been implemented on a class.
NullReferenceException	E_POINTER	Thrown when there is an attempt to de-reference a null object reference.
ObjectDisposedException	RO_E_CLOSED	Thrown when an operation is performed on a disposed object.
OperationCanceledException	E_ABORT	Thrown when an operation is aborted.
OutOfBoundsException	E_BOUNDS	Thrown when an operation attempts to access data outside the valid range.
OutOfMemoryException	E_OUTOFMEMORY	Thrown when there's insufficient memory to complete the operation.
WrongThreadException	RPC_E_WRONG_THREAD	Thrown when a thread calls via an interface pointer which is for a proxy object that does not belong to the thread's apartment.

HResult and Message properties

All exceptions have an HResult property and a Message property. The Exception::HResult property gets the exception's underlying numeric HRESULT value. The Exception::Message property gets the system-supplied string that describes the exception. In Windows 8, the message is available only in the debugger and is read-only. This means that you cannot change it when you rethrow the exception. In Windows 8.1, you can access the message string programmatically and provide a new message if you rethrow the exception. Better callstack information is also available in the debugger, including callstacks for asynchronous method calls.

Examples

This example shows how to throw a Windows Runtime exception for synchronous operations:

```
String^ Class1::MyMethod(String^ argument)
{

    if (argument->Length() == 0)
    {
        auto e = ref new Exception(-1, "I'm Zork bringing you this message from across the ABI.");
        //throw ref new InvalidArgumentException();
        throw e;
    }

    return MyMethodInternal(argument);
}
```

The next example shows how to catch the exception.

```
void Class2::ProcessString(String^ input)
{
    String^ result = nullptr;
    auto obj = ref new Class1();

    try
    {
        result = obj->MyMethod(input);
    }

    catch (/*InvalidArgument*/Exception^ e)
    {
        // Handle the exception in a way that's appropriate
        // for your particular scenario. Assume
        // here that this string enables graceful
        // recover-and-continue. Why not?
        result = ref new String(L"forty two");

        // You can use Exception data for logging purposes.
        Windows::Globalization::Calendar calendar;
        LogMyErrors(calendar.GetDateTime(), e->HResult, e->Message);
    }

    // Execution continues here in both cases.
    //#include <string>
    std::wstring ws(result->Data());
    //...
}
```

To catch exceptions that are thrown during an asynchronous operation, use the task class and add an error-handling continuation. The error-handling continuation marshals exceptions that are thrown on other threads back to the calling thread so that you can handle all potential exceptions at just one point in your code. For more information, see Asynchronous Programming in C++.

UnhandledErrorDetected event

In Windows 8.1 you can subscribe to the Windows::ApplicationModel::Core::CoreApplication::UnhandledErrorDetected static event, which provides access to unhandled errors that are about to bring down the process. Regardless of where the error originated, it reaches this handler as a Windows::ApplicationModel::Core::UnhandledError object that's passed in with the event args. When you call `Propagate` on the object, it creates and throws a `Platform::*Exception` of the type that corresponds to the error code. In the catch blocks, you can save user state if necessary and then either allow the process to terminate by calling `throw`, or do something to get the program back into a known state. The following example shows the basic pattern:

In app.xaml.h:

```
void OnUnhandledException(Platform::Object^ sender,
Windows::ApplicationModel::Core::UnhandledErrorDetectedEventArgs^ e);
```

In app.xaml.cpp:

```
// Subscribe to the event, for example in the app class constructor:
Windows::ApplicationModel::Core::CoreApplication::UnhandledErrorDetected += ref new
EventHandler<UnhandledErrorDetectedEventArgs^>(this, &App::OnUnhandledException);

// Event handler implementation:
void App::OnUnhandledException(Platform::Object^ sender,
Windows::ApplicationModel::Core::UnhandledErrorDetectedEventArgs^ e)
{
    auto err = e->UnhandledError;

    if (!err->Handled) //Propagate has not been called on it yet.
{
    try
    {
        err->Propagate();
    }
    // Catch any specific exception types if you know how to handle them
    catch (AccessDeniedException^ ex)
    {
        // TODO: Log error and either take action to recover
        // or else re-throw exception to continue fail-fast
    }
}
```

Remarks

C++/CX does not use the `finally` clause.

See also

C++/CX Language Reference

Namespaces Reference

Events (C++/CX)

9/21/2022 • 6 minutes to read • Edit Online

A Windows Runtime type can declare (that is, publish) events, and client code in the same component or in other components can subscribe to those events by associating methods called *event handlers* with the event. Multiple event handlers can be associated with a single event. When the publishing object raises the event, it causes all event handlers to be invoked. In this way, a subscribing class can perform whatever custom action is appropriate when the publisher raises the event. An event has a delegate type that specifies the signature that all event handlers must have in order to subscribe to the event.

Consuming events in Windows components

Many components in the Windows Runtime expose events. For example, a LightSensor object fires a ReadingChanged event when the sensor reports a new luminescence value. When you use a LightSensor object in your program, you can define a method that will be called when the ReadingChanged event is fired. The method can do whatever you want it to do; the only requirement is that its signature must match the signature of the delegate that is invoked. For more information about how to create an delegate event handler and subscribe to an event, see Delegates.

Creating custom events

Declaration

You can declare an event in a ref class or an interface, and it can have public, internal (public/private), public protected, protected, private protected, or private accessibility. When you declare an event, internally the compiler creates an object that exposes two accessor methods: add and remove. When subscribing objects register event handlers, the event object stores them in a collection. When an event is fired, the event object invokes all the handlers in its list in turn. A trivial event—like the one in the following example—has an implicit backing store as well as implicit `add` and `remove` accessor methods. You can also specify your own accessors, in the same way that you can specify custom `get` and `set` accessors on a property. The implementing class cannot manually cycle through the event subscriber list in a trivial event.

The following example shows how to declare and fire an event. Notice that the event has a delegate type and is declared by using the "^" symbol.

```
namespace EventTest
{
    ref class Class1;
    public delegate void SomethingHappenedEventHandler(Class1^ sender, Platform::String^ s);

    public ref class Class1 sealed
    {
    public:
        Class1(){}
        event SomethingHappenedEventHandler^ SomethingHappened;
        void DoSomething()
        {
            //Do something....

            // ...then fire the event:
            SomethingHappened(this, L"Something happened.");
        }
    };
}
```

Usage

The following example shows how a subscribing class uses the `+=` operator to subscribe to the event, and provide an event handler to be invoked when the event is fired. Notice that the function that's provided matches the signature of the delegate that's defined on the publisher side in the `EventTest` namespace.

```
namespace EventClient
{
    using namespace EventTest;
    namespace PC = Platform::Collections; //#include <collection.h>

    public ref class Subscriber sealed
    {
    public:
        Subscriber() : eventCount(0)
        {
            // Instantiate the class that publishes the event.
            publisher= ref new EventTest::Class1();

            // Subscribe to the event and provide a handler function.
            publisher->SomethingHappened +=
                ref new EventTest::SomethingHappenedEventHandler(
                this,
                &Subscriber::MyEventHandler);
            eventLog = ref new PC::Map<int, Platform::String^>();
        }
        void SomeMethod()
        {
            publisher->DoSomething();
        }

        void MyEventHandler(EventTest::Class1^ mc, Platform::String^ msg)
        {
            // Our custom action: log the event.
            eventLog->Insert(eventCount, msg);
            eventCount++;
        }

    private:
        PC::Map<int, Platform::String^>^ eventLog;
        int eventCount;
        EventTest::Class1^ publisher;
    };
}
```

WARNING

In general, it's better to use a named function, rather than a lambda, for an event handler unless you take great care to avoid circular references. A named function captures the "this" pointer by weak reference, whereas a lambda captures it by strong reference and creates a circular reference. For more information, see Weak references and breaking cycles (C++/CX).

Custom add and remove methods

Internally, an event has an add method, a remove method, and a raise method. When client code subscribes to an event, the add method is called and the delegate that's passed in is added to the event's invocation list. The publishing class invokes the event, it causes the raise() method to be called, and each delegate in the list is invoked in turn. A subscriber can remove itself from the delegate list, which causes the event's remove method to be called. The compiler provides default versions of these methods if you don't define them in your code; these are known as trivial events. In many cases, a trivial event is all that's required.

You can specify custom add, remove, and raise methods for an event if you have to perform custom logic in response to the addition or removal of subscribers. For example, if you have an expensive object that is only required for event reporting, you can lazily defer the creation of the object until a client actually subscribes to the event.

The next example shows how to add custom add, remove, and raise methods to an event:

```
namespace EventTest2
{
    ref class Class1;
    public delegate void SomethingHappenedEventHandler(Class1^ sender, Platform::String^ msg);

    public ref class Class1 sealed
    {
    public:
        Class1(){}
        event SomethingHappenedEventHandler^ SomethingHappened;
        void DoSomething(){/*...*/}
        void MethodThatFires()
        {
            // Fire before doing something...
            BeforeSomethingHappens(this, "Something's going to happen.");

            DoSomething();

            // ...then fire after doing something...
            SomethingHappened(this, L"Something happened.");
        }

        event SomethingHappenedEventHandler^ _InternalHandler;

        event SomethingHappenedEventHandler^ BeforeSomethingHappens
        {
            Windows::Foundation::EventRegistrationToken add(SomethingHappenedEventHandler^ handler)
            {
                // Add custom logic here:
                //....
                return _InternalHandler += handler;
            }

            void remove(Windows::Foundation::EventRegistrationToken token)
            {
                // Add custom logic here:
                //....
                _InternalHandler -= token;
            }

            void raise(Class1^ sender, Platform::String^ str)
            {

                // Add custom logic here:
                //....
                return _InternalHandler(sender, str);
            }
        }
    };
}
```

Removing an event handler from the subscriber side

In some rare cases, you may want to remove an event handler for an event that you previously subscribed to. For example, you may want to replace it with another event handler or you may want to delete some resources that are held by it. To remove a handler, you must store the EventRegistrationToken that's returned from the `+=` operation. You can then use the `-=` operator on the token to remove an event handler. However, the original handler could still be invoked even after it's removed. For example, a race condition may arise when the event source gets a list of handlers and starts to invoke them. If an event handler gets removed while this happens, the list becomes out of date. So, if you intend to remove an event handler, create a member flag. Set it if the event is removed, and then in the event handler, check the flag, and return immediately if it's set. The next example shows the basic pattern.

```
namespace EventClient2
{
    using namespace EventTest2;

    ref class Subscriber2 sealed
    {
    private:
        bool handlerIsActive;
        Platform::String^ lastMessage;

        void TestMethod()
        {
            Class1^ c1 = ref new Class1();
            handlerIsActive = true;
            Windows::Foundation::EventRegistrationToken cookie =
                c1->SomethingHappened +=
                ref new EventTest2::SomethingHappenedEventHandler(this, &Subscriber2::MyEventHandler);
            c1->DoSomething();

            // Do some other work..then remove the event handler and set the flag.
            handlerIsActive = false;
            c1->SomethingHappened -= cookie;
        }

        void MyEventHandler(Class1^ mc, Platform::String^ msg)
        {
            if (!handlerIsActive)
                return;
            lastMessage = msg;
        }
    };
}
```

Remarks

Multiple handlers may be associated with the same event. The event source sequentially calls into all event handlers from the same thread. If an event receiver blocks within the event handler method, it blocks the event source from invoking other event handlers for this event.

The order in which the event source invokes event handlers on event receivers is not guaranteed and may differ from call to call.

See also

Type System
Delegates
C++/CX Language Reference
Namespaces Reference

Casting (C++/CX)

9/21/2022 • 5 minutes to read • Edit Online

Four different cast operators apply to Windows Runtime types: static_cast Operator, dynamic_cast Operator, **safe_cast Operator**, and reinterpret_cast Operator. **safe_cast** and `static_cast` throw an exception when the conversion can't be performed; static_cast Operator also performs compile-time type checking. `dynamic_cast` returns `nullptr` if it fails to convert the type. Although `reinterpret_cast` returns a non-null value, it might be invalid. For this reason, we recommend that you not use `reinterpret_cast` unless you know that the cast will succeed. In addition, we recommend that you not use C-style casts in your C++/CX code because they are identical to `reinterpret_cast`.

The compiler and runtime also perform implicit casts—for example, in boxing operations when a value type or built-in type are passed as arguments to a method whose parameter type is `Object^`. In theory, an implicit cast should never cause an exception at run time; if the compiler can't perform an implicit conversion, it raises an error at compile time.

Windows Runtime is an abstraction over COM, which uses HRESULT error codes instead of exceptions. In general, the Platform::InvalidCastException indicates a low-level COM error of E_NOINTERFACE.

static_cast

A `static_cast` is checked at compile time to determine whether there is an inheritance relationship between the two types. The cast causes a compiler error if the types are not related.

A `static_cast` on a ref class also causes a run-time check to be performed. A `static_cast` on a ref class can pass compile time verification but still fail at run time; in this case a `Platform::InvalidCastException` is thrown. In general, you don't have to handle these exceptions because almost always they indicate programming errors that you can eliminate during development and testing.

Use `static_cast` if the code explicitly declares a relationship between the two types, and you therefore are sure that the cast should work.

```
    interface class A{};
    public ref class Class1 sealed : A { };
    // ...
    A^ obj = ref new Class1(); // Class1 is an A
    // You know obj is a Class1. The compiler verifies that this is possible, and in C++/CX a run-time check
is also performed.
    Class1^ c = static_cast<Class1^>(obj);
```

safe_cast

The **safe_cast** operator is part of Windows Runtime. It performs a run-time type check and throws a `Platform::InvalidCastException` if the conversion fails. Use **safe_cast** when a run-time failure indicates an exceptional condition. The primary purpose of **safe_cast** is to help identify programming errors during the development and testing phases at the point where they occur. You don't have to handle the exception because the unhandled exception itself identifies the point of failure.

Use safe_cast if the code does not declare the relationship but you are sure that the cast should work.

```
    // A and B are not related
    interface class A{};
    interface class B{};
    public ref class Class1 sealed : A, B { };
    // ...
    A^ obj = ref new Class1();

    // You know that obj's backing type implements A and B, but
    // the compiler can't tell this by comparing A and B. The run-time type check succeeds.
    B^ obj2 = safe_cast<B^>(obj);
```

dynamic_cast

Use `dynamic_cast` when you cast an object (more specifically, a hat ^) to a more derived type, you expect either that the target object might sometimes be `nullptr` or that the cast might fail, and you want to handle that condition as a regular code path instead of an exception. For example, in the **Blank App (Universal Windows)** project template, the `OnLaunched` method in app.xaml.cpp uses `dynamic_cast` to test whether the app window has content. It's not an error if it doesn't have content; it is an expected condition. `Windows::Current::Content` is a `Windows::UI::XAML::UIElement` and the conversion is to a `Windows::UI.XAML::Controls::Frame`, which is a more derived type in the inheritance hierarchy.

```
void App::OnLaunched(Windows::ApplicationModel::Activation::LaunchActivatedEventArgs^ args)
{
    auto rootFrame = dynamic_cast<Frame^>(Window::Current->Content);

    // Do not repeat app initialization when the window already has content,
    // just ensure that the window is active
    if (rootFrame == nullptr)
    {
        // Create a Frame to act as the navigation context and associate it with
        // a SuspensionManager key
        rootFrame = ref new Frame();
        // ...
    }
}
```

Another use of `dynamic_cast` is to probe an `Object^` to determine whether it contains a boxed value type. In this case, you attempt a `dynamic_cast<Platform::Box>` or a `dynamic_cast<Platform::IBox>`.

dynamic_cast and tracking references (%)

You can also apply a `dynamic_cast` to a tracking reference, but in this case the cast behaves like **safe_cast**. It throws `Platform::InvalidCastException` on failure because a tracking reference cannot have a value of `nullptr`.

reinterpret_cast

We recommend that you not use reinterpret_cast because neither a compile-time check nor a run-time check is performed. In the worst case, a `reinterpret_cast` makes it possible for programming errors to go undetected at development time and cause subtle or catastrophic errors in your program's behavior. Therefore, we recommend that you use `reinterpret_cast` only in those rare cases when you must cast between unrelated types and you know that the cast will succeed. An example of a rare use is to convert a Windows Runtime type to its underlying ABI type—this means that you are taking control of the reference counting for the object. To do this, we recommend that you use the ComPtr Class smart pointer. Otherwise, you must specifically call Release on the interface. The following example shows how a ref class can be cast to an `IInspectable*`.

```
#include <wrl.h>
using namespace Microsoft::WRL;
auto winRtObject = ref new SomeWinRTType();
ComPtr<IInspectable> inspectable = reinterpret_cast<IInspectable*>(winRtObject);
// ...
```

If you use `reinterpret_cast` to convert from one Windows Runtime interface to another, you cause the object to be released twice. Therefore, only use this cast when you are converting to a non-C++ component extensions interface.

ABI types

- ABI types live in headers in the Windows SDK. Conveniently, the headers are named after the namespaces—for example, `windows.storage.h`.
- ABI types live in a special namespace ABI—for example, `ABI::Windows::Storage::Streams::IBuffer*`.
- Conversions between a Windows Runtime interface type and its equivalent ABI type are always safe—that is, `IBuffer^` to `ABI::IBuffer*`.
- A Windows Runtime runtime class should always be converted to `IInspectable*` or its default interface, if that is known.
- After you convert to ABI types, you own the lifetime of the type and must follow the COM rules. We recommend that you use `WRL::ComPtr` to simplify lifetime management of ABI pointers.

The following table summarizes the cases in which it is safe to use `reinterpret_cast`. In every case, the cast is safe in both directions.

CAST FROM, CAST TO	CAST TO, CAST FROM
`HSTRING`	`String^`
`HSTRING*`	`String^*`
`IInspectable*`	`Object^`
`IInspectable**`	`Object^*`
`IInspectable-derived-type*`	`same-interface-from-winmd^`
`IInspectable-derived-type**`	`same-interface-from-winmd^*`
`IDefault-interface-of-RuntimeClass*`	`same-RefClass-from-winmd^`
`IDefault-interface-of-RuntimeClass**`	`same-RefClass-from-winmd^*`

See also

- Type System
- C++/CX Language Reference
- Namespaces Reference

Boxing (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

Boxing is wrapping a value type variable such as Windows::Foundation::DateTime—or a fundamental scalar type such as `int`—in a ref class when the variable is passed to a method that takes Platform::Object^ as its input type.

Passing a value type to an Object^ parameter

Although you don't have to explicitly box a variable to pass it to a method parameter of type Platform::Object^, you do have to explicitly cast back to the original type when you retrieve values that have been previously boxed.

```
Object^ obj = 5; //scalar value is implicitly boxed
int i = safe_cast<int>(obj); //unboxed with explicit cast.
```

Using Platform::IBox<T> to support nullable value types

C# and Visual Basic support the concept of nullable value types. In C++/CX, you can use the `Platform::IBox<T>` type to expose public methods that support nullable value type parameters. The following example shows a C++/CX public method that returns null when a C# caller passes null for one of the arguments.

```
// A WinRT Component DLL
namespace BoxingDemo
{
    public ref class Class1 sealed
    {
    public:
        Class1(){}
        Platform::IBox<int>^ Multiply(Platform::IBox<int>^ a, Platform::IBox<int>^ b)
        {
            if(a == nullptr || b == nullptr)
                return nullptr;
            else
                return ref new Platform::Box<int>(a->Value * b->Value);
        }
    };
```

In a C# XAML client, you can consume it like this:

```
// C# client code
BoxingDemo.Class1 obj = new BoxingDemo.Class1();
int? a = null;
int? b = 5;
var result = obj.Multiply(a, b); //result = null
```

See also

Type System (C++/CX)
Casting (C++/CX)
C++/CX Language Reference
Namespaces Reference

Attributes (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

An attribute is a special kind of ref class that can be prepended in square brackets to Windows Runtime types and methods to specify certain behaviors in metadata creation. Several predefined attributes—for example, Windows::Foundation::Metadata::WebHostHidden—are commonly used in C++/CX code. This example shows how the attribute is applied to a class:

```
[Windows::Foundation::Metadata::WebHostHidden]
public ref class MyClass : Windows::UI::Xaml::DependencyObject {};
```

Custom attributes

You can also define custom attributes. Custom attributes must conform to these Windows Runtime rules:

- Custom attributes can contain only public fields.
- Custom attribute fields can be initialized when the attribute is applied to a class.
- A field may be one of these types:
  - int32 (int)
  - uint32 (unsigned int)
  - bool
  - Platform::String^
  - Windows::Foundation::HResult
  - Platform::Type^
  - public enum class (includes user-defined enums)

The next example shows how to define a custom attribute and then initialize it when you use it.

```
[Windows::Foundation::Metadata::WebHostHiddenAttribute]
public ref class MyCustomAttribute sealed : Platform::Metadata::Attribute {
public:
    int Num;
    Platform::String^ Msg;
};

[MyCustomAttribute(Num=5, Msg="Hello")]
public ref class Class1 sealed
{
public:
    Class1();
};
```

See also

Type System (C++/CX)
C++/CX Language Reference

Deprecating types and members (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

C++/CX supports deprecation of Windows Runtime types and members for producers and consumers by using the `Deprecated` attribute. If you consume an API that has this attribute, you get a compile-time warning message. It indicates that the API is deprecated and also recommends an alternative API to use. In your own public types and methods, you can apply this attribute and supply your own custom message.

Caution

The `Deprecated` attribute is for use only with Windows Runtime types. For standard C++ classes and members, use `[[deprecated]]` (C++14 and later) or `__declspec(deprecated)`.

Example

The following example shows how to deprecate your own public APIs—for example, in a Windows Runtime component. The second parameter, of type `Windows:Foundation::Metadata::DeprecationType` specifies whether the API is being deprecated or removed. Currently only the `DeprecationType::Deprecated` value is supported. The third parameter in the attribute specifies the `Windows::Foundation::Metadata::Platform` to which the attribute applies.

```
namespace wfm = Windows::Foundation::Metadata;

public ref class Bicycle sealed
{

public:
    property double Speed;

    [wfm::Deprecated("Use the Speed property to compute the angular speed of the wheel",
wfm::DeprecationType::Deprecate, 0x0)]
    double ComputeAngularVelocity();
};
```

Supported targets

The following table lists the constructs to which the `Deprecated` attribute may be applied:

class
enum
event
method
property
struct field

delegate
enum field
interface
parameterized constructor
struct
XAML control

See also

Type system (C++/CX)
C++/CX language reference
Namespaces reference

Building apps and libraries (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

The topics in this section provide a few details about using the build system to produce a Universal Windows Platform app or Windows Runtime component.

In this section

- Compiler and Linker options
- Static libraries
- DLLs

> **NOTE**
> Visual Studio does not support profile guided optimizations for Universal Windows Platform. If you attempt to build a project with these options set in the IDE, a build error will result. Console applications are also not supported.

Compiler and Linker options (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

An environment variable, C++/CX compiler options, and linker options support the building of apps for the Windows Runtime.

Library path

The %LIBPATH% environment variable specifies the default path to search for .winmd files.

Compiler options

OPTION	DESCRIPTION
/ZW /ZW:nostdlib	Enables Windows Runtime language extensions. The `nostdlib` parameter prevents the compiler from using the standard, predefined search path to find assembly and .winmd files. The **/ZW** compiler option implicitly specifies the following compiler options: - **/FI** vccorlib.h, which forces inclusion of the vccorlib.h header file that defines many types that are required by the compiler. - /FU Windows.winmd, which forces inclusion of the Windows.winmd metadata file that's provided by the operating system and defines many types in the Windows Runtime. - **/FU** Platform.winmd, which forces inclusion of the Platform.winmd metadata file that's provided by the compiler and defines most types in the Platform family of namespaces.
/AI *dir*	Adds a directory, which is specified by the *dir* parameter, to the search path that the compiler uses to find assembly and .winmd files.
/FU *file*	Forces the inclusion of the specified module, or .winmd file. That is, you don't have to specify `#using` *file* in your source code. The compiler automatically forces the inclusion of its own Windows metadata file, Platform.winmd.
/D "WINAPI_FAMILY=2"	Creates a definition that enables the use of a subset of the Win32 SDK that's compatible with the Windows Runtime.

Linker options

OPTION	DESCRIPTION
/APPCONTAINER[:NO]	Marks the executable as runnable in the appcontainer (only).

OPTION	DESCRIPTION
/WINMD[:{NO|ONLY}]	Emits a .winmd file and an associated binary file. This option must be passed to the linker for a .winmd to be emitted. **NO**—Doesn't emit a .winmd file, but does emit a binary file. **ONLY**—Emits a .winmd file, but doesn't emit a binary file.
/WINMDFILE:*filename*	The name of the .winmd file to emit, instead of the default .winmd file name. If multiple file names are specified on the command line, the last name is used.
/WINMDDELAYSIGN[:NO]	Partially signs the .winmd file and places the public key in the binary. **NO**—(Default) Doesn't sign the .winmd file. /WINMDDELAYSIGN has no effect unless /WINMDKEYFILE or /WINMDKEYCONTAINER is also specified.
/WINMDKEYCONTAINER:*name*	Specifies a key container to sign an assembly. The *name* parameter corresponds to the key container that's used to sign the metadata file.
/WINMDKEYFILE:*filename*	Specifies a key or a key pair to sign the assembly. The *filename* parameter corresponds to the key that's used to sign the metadata file.

Remarks

When you use **/ZW**, the compiler automatically links to the DLL version of the C Runtime (CRT). Linking to the static library version is not allowed, and any use of CRT functions that are not allowed in a Universal Windows Platform app will cause a compile-time error.

See also

Building apps and libraries

Static libraries (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

A static library that's used in a Universal Windows Platform (UWP) app can contain ISO-standard C++ code, including STL types, and also calls to Win32 APIs that are not excluded from the Windows Runtime app platform. A static library consumes Windows Runtime components and may create Windows Runtime components with certain restrictions.

Creating static libraries

Instructions for creating a new project vary depending on which version of Visual Studio you have installed. To see the documentation for your preferred version of Visual Studio, use the **Version** selector control. It's found at the top of the table of contents on this page.

To create a UWP static library in Visual Studio

1. On the menu bar, choose **File** > **New** > **Project** to open the **Create a New Project** dialog.
2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **UWP**.
3. From the filtered list of project types, choose **Static Library (Universal Windows - C++/CX)** then choose **Next**. In the next page, give the project a name, and specify the project location if desired.
4. Choose the **Create** button to create the project.

To create a UWP static library in Visual Studio 2017 or Visual Studio 2015

1. On the menu bar, choose **File** > **New** > **Project**. Under **Visual C++** > **Windows Universal** choose **Static Library (Universal Windows)**.
2. In **Solution Explorer**, open the shortcut menu for the project and then choose **Properties**. In the **Properties** dialog box, on the **Configuration Properties** > **C/C++** page, set **Consume Windows Runtime Extension** to **Yes (/ZW)**.

When you compile a new static library, if you make a call to a Win32 API that's excluded for UWP apps, the compiler will raise error C3861, "Identifier not found." To look for an alternative method that's supported for the Windows Runtime, see Alternatives to Windows APIs in UWP apps.

If you add a C++ static library project to a UWP app solution, you might have to update the library project's property settings so that the UWP support property is set to **Yes**. Without this setting, the code builds and links, but an error occurs when you attempt to verify the app for the Microsoft Store. The static lib should be compiled with the same compiler settings as the project that consumes it.

If you consume a static library that creates public `ref` classes, public interface classes, or public value classes, the linker raises this warning:

> **warning LNK4264:** archiving object file compiled with /ZW into a static library; note that when authoring Windows Runtime types it is not recommended to link with a static library that contains Windows Runtime metadata.

You can safely ignore the warning only if the static library is not producing Windows Runtime components that are consumed outside the library itself. If the library doesn't consume a component that it defines, then the linker can optimize away the implementation even though the public metadata contains the type information.

This means that public components in a static library will compile but will not activate at run time. For this reason, any Windows Runtime component that's intended for consumption by other components or apps must be implemented in a dynamic-link library (DLL).

See also

Threading and Marshaling

DLLs (C++/CX)

9/21/2022 • 3 minutes to read • Edit Online

You can use Visual Studio to create either a standard Win32 DLL or a Windows Runtime component DLL that can be consumed by Universal Windows Platform (UWP) apps. A standard DLL that was created by using a version of Visual Studio or the Microsoft C++ compiler that's earlier than Visual Studio 2012 may not load correctly in a UWP app and may not pass the app verification test in the Microsoft Store.

Windows Runtime component DLLs

In almost all cases, when you want to create a DLL for use in a UWP app, create it as a Windows Runtime component by using the project template of that name. You can create a Windows Runtime component project for DLLs that have public or private Windows Runtime types. A Windows Runtime component can be accessed from apps that are written in any Windows Runtime-compatible language. By default, the compiler settings for a Windows Runtime component project use the **/ZW** switch. A .winmd file must have the same name that the root namespace has. For example, a class that's named A.B.C.MyClass can be instantiated only if it's defined in a metadata file that's named A.winmd or A.B.winmd or A.B.C.winmd. The name of the DLL is not required to match the .winmd file name.

For more information, see Creating Windows Runtime Components in C++.

To reference a third-party Windows Runtime component binary in your project

1. Open the shortcut menu for the project that will use the DLL and then choose **Properties**. On the **Common Properties** page, choose the **Add New Reference** button.
2. A Windows Runtime component consists of a DLL file and a .winmd file that contains the metadata. Typically, these files are located in the same folder. In the left pane of the **Add Reference** dialog box, choose the **Browse** button and then navigate to the location of the DLL and its .winmd file. For more information, see Extension SDKs.

Standard DLLs

You can create a standard DLL for C++ code that doesn't consume or produce public Windows Runtime types and consume it from a UWP app. Use the Dynamic-Link Library (DLL) project type when you just want to migrate an existing DLL to compile in this version of Visual Studio but not convert the code to a Windows Runtime Component project. When you use the following steps, the DLL will be deployed alongside your app executable in the .appx package.

To create a standard DLL in Visual Studio

1. On the menu bar, choose **File**, **New**, **Project**, and then select the **Dynamic Link Library (DLL)** template.
2. Enter a name for the project, and then choose the **OK** button.
3. Add the code. Be sure to use `__declspec(dllexport)` for functions that you intend to export—for example, `__declspec(dllexport) Add(int I, in j);`
4. Add `#include winapifamily.h` to include that header file from the Windows SDK for UWP apps and set the macro `WINAPI_FAMILY=WINAPI_PARTITION_APP`.

To reference a standard DLL project from the same solution

1. Open the shortcut menu for the project that will use the DLL and then choose **Properties**. On the

Common Properties page, choose the **Add New Reference** button.

2. In the left pane, select **Solution**, and then select the appropriate check box in the right pane.
3. In your source code files, add a `#include` statement for the DLL header file, as needed.

To reference a standard DLL binary

1. Copy the DLL file, the .lib file, and the header file, and paste them in a known location—for example, in your current project folder.
2. Open the shortcut menu for the project that will use the DLL and then choose **Properties**. On the **Configuration Properties**, **Linker**, **Input** page, add the .lib file as a dependency.
3. In your source code files, add a `#include` statement for the DLL header file, as needed.

To migrate an existing Win32 DLL for UWP app compatibility

1. Create a project of the DLL (Universal Windows) type and add your existing source code to it.
2. Add `#include winapifamily.h` to include that header file from the Windows SDK for UWP apps and set the macro `WINAPI_FAMILY=WINAPI_PARTITION_APP`.
3. In your source code files, add a `#include` statement for the DLL header file, as needed.

Interoperating with Other Languages (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

This part of the documentation describes how to use C++/CX to author Windows Runtime components that can be consumed by other programming languages and libraries. You can also author components that can't be consumed by all languages. This section describes different aspects to consider when your C++/CX application interoperates with components that are written by using JavaScript, a .NET Framework managed language, or the Windows Runtime C++ Template Library.

Related topics

- JavaScript integration
- CLR integration
- WRL integration
- C++/CX Language Reference

JavaScript integration (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

For more information, see Creating Windows Runtime Components in C++.

See also

Interoperating with Other Languages

CLR integration (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

Some Windows Runtime types receive special handling in C++/CX and the languages that are based on the common language runtime (CLR). This article discusses how several types in one language map to another language. For example, the CLR maps Windows.Foundation.IVector to System.Collections.IList, Windows.Foundation.IMap to System.Collections.IDictionary, and so on. Similarly, C++/CX specially maps types such as Platform::Delegate and Platform::String.

Mapping the Windows Runtime to C++/CX

When C++/CX reads a Windows metadata (.winmd) file, the compiler automatically maps common Windows Runtime namespaces and types to C++/CX namespaces and types. For example, the numeric Windows Runtime type `UInt32` is automatically mapped to `default::uint32`.

C++/CX maps several other Windows Runtime types to the **Platform** namespace. For example, the **Windows::Foundation** HSTRING handle, which represents a read-only Unicode text string, is mapped to the C++/CX `Platform::String` class. When a Windows Runtime operation returns an error HRESULT, it's mapped to a C++/CX `Platform::Exception`.

The C++/CX also maps certain types in Windows Runtime namespaces to enhance the functionality of the type. For these types, C++/CX provides helper constructors and methods that are specific to C++ and are not available in the type's standard .winmd file.

The following lists show value structs that support new constructors and helper methods. If you have previously written code that uses struct initialization lists, change it to use the newly added constructors.

Windows::Foundation

- Point
- Rect
- Size

Windows::UI

- Color

Windows::UI::Xaml

- CornerRadius
- Duration
- GridLength
- Thickness

Windows::UI::Xaml::Interop

- TypeName

Windows::UI::Xaml::Media

- Matrix

Windows::UI::Xaml::Media::Animation

- KeyTime
- RepeatBehavior

Windows::UI::Xaml::Media::Media3D

- Matrix3D

Mapping the CLR to C++/CX

When the Microsoft C++ or C# compilers read a .winmd file, they automatically map certain types in the metadata file to appropriate C++/CX or CLR types. For example, in the CLR, the IVector<T> interface is mapped to IList<T>. But in C++/CX, the IVector<T> interface is not mapped to another type.

IReference<T> in the Windows Runtime maps to Nullable<T> in .NET.

See also

Interoperating with Other Languages

WRL integration (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

You freely can mix WRL code with Windows Runtime C++ Template Library (WRL) code. In the same translation unit, you can use objects declared with WRL handle-to-object (`^`) notation and WRL smart pointer (`ComPtr<T>`) notation. However, you must manually handle return values, and WRL HRESULT error codes and WRL exceptions.

WRL development

For more information about authoring and consuming WRL components, see Windows Runtime C++ Template Library (WRL).

Example

The following code snippet demonstrates using WRL and WRL to consume Windows Runtime classes and examine a metadata file.

The example is taken from a code snippet in the Building Microsoft Store apps forum. The author of this code snippet offers the following disclaimers and stipulations:

1. C++ doesn't provide specific APIs to reflect on Windows Runtime types, but Windows metadata files (.winmd) for a type are fully compatible with CLR metadata files. Windows provides the new metadata discovery APIs (RoGetMetaDataFile) to get to the .winmd file for a given type. However, these APIs are of limited use to C++ developers because you can't instantiate a class.
2. After the code is compiled, you'll also need to pass Runtimeobject.lib and Rometadata.lib to the Linker.
3. This snippet is presented as-is. While it is expected to work correctly, it possibly can contain errors.

```
#include <hstring.h>
#include <cor.h>
#include <rometadata.h>
#include <rometadataresolution.h>
#include <collection.h>

namespace ABI_Isolation_Workaround {
    #include <inspectable.h>
    #include <WeakReference.h>
}
using namespace ABI_Isolation_Workaround;
#include <wrl/client.h>

using namespace Microsoft::WRL;
using namespace Windows::Foundation::Collections;

IVector<String^>^ GetTypeMethods(Object^);

MainPage::MainPage()
{
    InitializeComponent();

    Windows::Foundation::Uri^ uri = ref new Windows::Foundation::Uri("http://buildwindows.com/");
    auto methods = GetTypeMethods(uri);

    std::wstring strMethods;
    std::for_each(begin(methods), end(methods), [&strMethods](String^ methodName) {
        strMethods += methodName->Data();
```

```
        strMethods += L"\n";
    });

    wprintf_s(L"%s\n", strMethods.c_str());
}

IVector<String^>^ GetTypeMethods(Object^ instance)
{
    HRESULT hr;
    HSTRING hStringClassName;
    hr = instance->__cli_GetRuntimeClassName(reinterpret_cast<__cli_HSTRING__**>(&hStringClassName)); //
internal method name subject to change post BUILD
    if (FAILED(hr))
        __cli_WinRTThrowError(hr); // internal method name subject to change post BUILD
    String^ className = reinterpret_cast<String^>(hStringClassName);

    ComPtr<IMetaDataDispenserEx> metadataDispenser; ComPtr<IMetaDataImport2> metadataImport; hr =
MetaDataGetDispenser(CLSID_CorMetaDataDispenser, IID_IMetaDataDispenser,
(LPVOID*)metadataDispenser.GetAddressOf());
    if (FAILED(hr))
        __cli_WinRTThrowError(hr); // internal method name subject to change post BUILD

    HSTRING hStringFileName;
    mdTypeDef typeDefToken;
    hr = RoGetMetaDataFile(hStringClassName, metadataDispenser.Get(), &hStringFileName, &metadataImport,
&typeDefToken);
    if (FAILED(hr))
        __cli_WinRTThrowError(hr); // internal method name subject to change post BUILD
    String^ fileName = reinterpret_cast<String^>(hStringFileName);

    HCORENUM hCorEnum = 0;
    mdMethodDef methodDefs[2048];
    ULONG countMethodDefs = sizeof(methodDefs);
    hr = metadataImport->EnumMethods(&hCorEnum, typeDefToken, methodDefs, countMethodDefs,
&countMethodDefs);
    if (FAILED(hr))
        __cli_WinRTThrowError(hr); // internal method name subject to change post BUILD

    wchar_t methodName[1024];
    ULONG countMethodName;
    std::wstring strMethods;
    Vector<String^>^ retVal = ref new Vector<String^>();

    for (int i = 0; i < countMethodDefs; ++i)
    {
        countMethodName = sizeof(methodName);
        hr = metadataImport->GetMethodProps(methodDefs[i], nullptr, methodName, countMethodName,
&countMethodName, nullptr, nullptr, nullptr, nullptr, nullptr);
        if (SUCCEEDED(hr))
        {
            methodName[ countMethodName ] = 0;
            retVal->Append(ref new String(methodName));
        }
    }
    return retVal;
}
```

See also

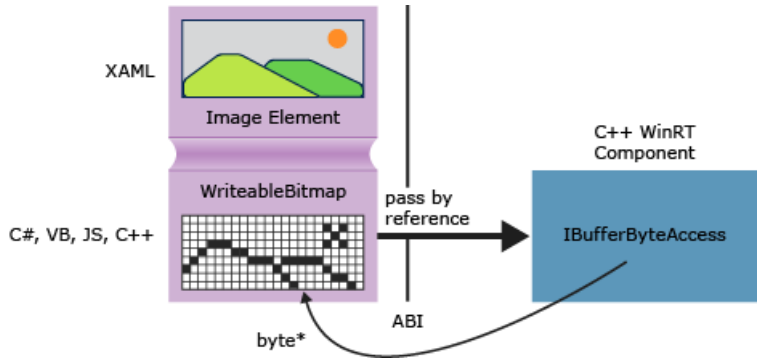
Interoperating with Other Languages

Obtaining pointers to data buffers (C++/CX)

9/21/2022 • 4 minutes to read • Edit Online

In the Windows Runtime the Windows::Storage::Streams::IBuffer interface provides a language-neutral, stream-based means to access data buffers. In C++ you can get a raw pointer to the underlying byte array by using the Windows Runtime Library IBufferByteAccess interface that is defined in robuffer.h. By using this approach you can modify the byte array in-place without making any unnecessary copies of the data.

The following diagram shows a XAML image element, whose source is a Windows::UI::Xaml::Media::Imaging WriteableBitmap. A client app that's written in any language can pass a reference to the `WriteableBitmap` to C++ code and then C++ can use the reference to get at the underlying buffer. In a Universal Windows Platform app that's written in C++, you can use the function in the following example directly in the source code without packaging it in a Windows Runtime component.



GetPointerToPixelData

The following method accepts an Windows::Storage::Streams::IBuffer and returns a raw pointer to the underlying byte array. To call the function, pass in a WriteableBitmap::PixelBuffer property.

```
#include <wrl.h>
#include <robuffer.h>
using namespace Windows::Storage::Streams;
using namespace Microsoft::WRL;
typedef uint8 byte;
// Retrieves the raw pixel data from the provided IBuffer object.
// Warning: The lifetime of the returned buffer is controlled by
// the lifetime of the buffer object that's passed to this method.
// When the buffer has been released, the pointer becomes invalid
// and must not be used.
byte* Class1::GetPointerToPixelData(IBuffer^ pixelBuffer, unsigned int *length)
{
    if (length != nullptr)
    {
        *length = pixelBuffer ->Length;
    }
    // Query the IBufferByteAccess interface.
    ComPtr<IBufferByteAccess> bufferByteAccess;
    reinterpret_cast<IInspectable*>( pixelBuffer)->QueryInterface(IID_PPV_ARGS(&bufferByteAccess));

    // Retrieve the buffer data.
    byte* pixels = nullptr;
    bufferByteAccess->Buffer(&pixels);
    return pixels;
}
```

Complete Example

The following steps show how to create a C# Universal Windows Platform app that passes a `WriteableBitmap` to a C++ Windows Runtime component DLL. The C++ code obtains a pointer to the pixel buffer and performs a simple in-place modification on the image. As an alternative, you can create the client app in Visual Basic, JavaScript, or C++ instead of C#. If you use C++, you don't need the component DLL; you can just add these methods directly to the MainPage class or some other class that you define.

Create the client

1. Use the Blank app project template to create a C# Universal Windows Platform app.
2. In MainPage.xaml
   - Use this XAML to replace the `Grid` element:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel HorizontalAlignment="Left" Margin="176,110,0,0" VerticalAlignment="Top"
Width="932">
        <Image x:Name="Pic"/>
        <Button Content="Process Image" HorizontalAlignment="Stretch"
VerticalAlignment="Stretch" Height="47" Click="Button_Click_1"/>
    </StackPanel>
</Grid>
```

3. In MainPage.xaml.cs
   a. Add these namespace declarations:

```
using Windows.Storage;
using Windows.Storage.FileProperties;
using Windows.UI.Xaml.Media.Imaging;
using Windows.Storage.Streams;
using Windows.Storage.Pickers;
```

   b. Add a `WriteableBitmap` member variable to the `MainPage` class and name it `m_bm`.

```
private WriteableBitmap m_bm;
```

   c. Use the following code to replace the `OnNavigatedTo` method stub. This opens the file picker when the app is started. (Notice that the `async` keyword is added to the function signature).

```
async protected override void OnNavigatedTo(NavigationEventArgs e)
{
    FileOpenPicker openPicker = new FileOpenPicker();
    openPicker.ViewMode = PickerViewMode.Thumbnail;
    openPicker.SuggestedStartLocation = PickerLocationId.PicturesLibrary;
    openPicker.FileTypeFilter.Add(".jpg");
    openPicker.FileTypeFilter.Add(".jpeg");
    openPicker.FileTypeFilter.Add(".png");

    StorageFile file = await openPicker.PickSingleFileAsync();
    if (file != null)
    {
        // Get the size of the image for the WriteableBitmap constructor.
        ImageProperties props = await file.Properties.GetImagePropertiesAsync();
        m_bm = new WriteableBitmap((int)props.Height, (int)props.Width);
        m_bm.SetSource(await file.OpenReadAsync());
        Pic.Source = m_bm;
    }
    else
    {
        //  Handle error...
    }
}
```

d. Add the event handler for the button click. (Because the `ImageManipCPP` namespace reference hasn't been created yet, it might have a wavy underline in the editor window.)

```
async private void Button_Click_1(object sender, RoutedEventArgs e)
{
    ImageManipCPP.Class1 obj = new ImageManipCPP.Class1();
    await obj.Negativize(m_bm);
    Pic.Source = m_bm;
}
```

Create the C++ component

1. Add a new C++ Windows Runtime component to the existing solution, and name it `ImageManipCPP`. Add a reference to it in the C# project by right-clicking on that project in **Solution Explorer** and choosing **Add**, **Reference**.

2. In Class1.h

   a. Add this `typedef` at the second line, just after `#pragma once`:

   ```
   typedef uint8 byte;
   ```

   b. Add the `WebHostHidden` attribute just above the beginning of the `Class1` declaration.

   ```
   [Windows::Foundation::Metadata::WebHostHidden]
   ```

   c. Add this public method signature to `Class1`:

   ```
   Windows::Foundation::IAsyncAction^
   Negativize(Windows::UI::Xaml::Media::Imaging::WriteableBitmap^ bm);
   ```

   d. Add the signature from the `GetPointerToPixelData` method that is shown in the earlier code snippet. Make sure that this method is private.

3. In Class1.cpp

   a. Add these `#include` directives and namespace declarations:

```
#include <ppltasks.h>
#include <wrl.h>
#include <robuffer.h>

using namespace Windows::Storage;
using namespace Windows::UI::Xaml::Media::Imaging;
using namespace Windows::Storage::Streams;
using namespace Microsoft::WRL;
```

   b. Add the implementation of `GetPointerToPixelData` from the earlier code snippet.

   c. Add the implementation of `Negativize`. This method creates an effect that resembles a film negative by reversing the value of each RGB value in the pixel. We make the method asynchronous because on larger images it might take a perceptible amount of time to complete.

```
IAsyncAction^ Class1::Negativize(WriteableBitmap^ bm)
{
    unsigned int length;
    byte* sourcePixels = GetPointerToPixelData(bm->PixelBuffer, &length);
    const unsigned int width = bm->PixelWidth;
    const unsigned int height = bm->PixelHeight;

    return create_async([this, width, height, sourcePixels]
    {
        byte* temp = sourcePixels;
        for(unsigned int k = 0; k < height; k++)
        {
            for (unsigned int i = 0; i < (width * 4); i += 4)
            {
                int pos = k * (width * 4) + (i);
                temp[pos] = ~temp[pos];
                temp[pos + 1] = ~temp[pos + 1] / 3;
                temp[pos + 2] = ~temp[pos + 2] / 2;
                temp[pos + 3] = ~temp[pos + 3];
            }
        }
    });

}
```

   NOTE

   This method might run faster if you use AMP or the Parallel Patterns Library to parallelize the operation.

4. Ensure that you have at least one picture in your pictures folder, and then press F5 to compile and run the program.

Threading and Marshaling (C++/CX)

9/21/2022 • 5 minutes to read • Edit Online

In the vast majority of cases, instances of Windows Runtime classes, like standard C++ objects, can be accessed from any thread. Such classes are referred to as "agile". However, a small number of Windows Runtime classes that ship with Windows are non-agile, and must be consumed more like COM objects than standard C++ objects. You don't need to be a COM expert to use non-agile classes, but you do need to take into consideration the class's threading model and its marshaling behavior. This article provides background and guidance for those rare scenarios in which you need to consume an instance of a non-agile class.

Threading model and marshaling behavior

A Windows Runtime class can support concurrent thread access in various ways, as indicated by two attributes that are applied to it:

- `ThreadingModel` attribute can have one of the values—STA, MTA, or Both, as defined by the `ThreadingModel` enumeration.
- `MarshallingBehavior` attribute can have one of the values—Agile, None, or Standard as defined by the `MarshallingType` enumeration.

The `ThreadingModel` attribute specifies where the class is loaded when activated: only in a user-interface thread (STA) context, only in a background thread (MTA) context, or in the context of the thread that creates the object (Both). The `MarshallingBehavior` attribute values refer to how the object behaves in the various threading contexts; in most cases, you don't have to understand these values in detail. Of the classes that are provided by the Windows API, about 90 percent have `ThreadingModel` =Both and `MarshallingType` =Agile. This means that they can handle low-level threading details transparently and efficiently. When you use `ref new` to create an "agile" class, you can call methods on it from your main app thread or from one or more worker threads. In other words, you can use an agile class—no matter whether it's provided by Windows or by a third party—from anywhere in your code. You don't have to be concerned with the class's threading model or marshaling behavior.

Consuming Windows Runtime components

When you create a Universal Windows Platform app, you might interact with both agile and non-agile components. When you interact with non-agile components, you may encounter the following warning.

Compiler warning C4451 when consuming non-agile classes

For various reasons, some classes can't be agile. If you are accessing instances of non-agile classes from both a user-interface thread and a background thread, then take extra care to ensure correct behavior at run time. The Microsoft C++ compiler issues warnings when you instantiate a non-agile run-time class in your app at global scope or declare a non-agile type as a class member in a ref class that itself is marked as agile.

Of the non-agile classes, the easiest to deal with are those that have `ThreadingModel` =Both and `MarshallingType` =Standard. You can make these classes agile just by using the `Agile<T>` helper class. The following example shows a declaration of a non-agile object of type `Windows::Security::Credentials::UI::CredentialPickerOptions^`, and the compiler warning that's issued as a result.

```
ref class MyOptions
    {
    public:
        property Windows::Security::Credentials::UI::CredentialPickerOptions^ Options

        {
            Windows::Security::Credentials::UI::CredentialPickerOptions^ get()
            {
                return _myOptions;
            }
        }
    private:
        Windows::Security::Credentials::UI::CredentialPickerOptions^ _myOptions;
    };
```

Here's the warning that's issued:

```
Warning 1 warning C4451: 'Platform::Agile<T>::_object' : Usage of ref class
'Windows::Security::Credentials::UI::CredentialPickerOptions' inside this context can lead to invalid
marshaling of object across contexts. Consider using
'Platform::Agile<Windows::Security::Credentials::UI::CredentialPickerOptions>' instead
```

When you add a reference—at member scope or global scope—to an object that has a marshaling behavior of "Standard", the compiler issues a warning that advises you to wrap the type in `Platform::Agile<T>`: `Consider using 'Platform::Agile<Windows::Security::Credentials::UI::CredentialPickerOptions>' instead` If you use `Agile<T>`, you can consume the class like you can any other agile class. Use `Platform::Agile<T>` in these circumstances:

- The non-agile variable is declared at global scope.
- The non-agile variable is declared at class scope and there is a chance that consuming code might smuggle the pointer—that is, use it in a different apartment without correct marshaling.

If neither of those conditions apply, then you can mark the containing class as non-agile. In other words, you should directly hold non-agile objects only in non-agile classes, and hold non-agile objects via Platform::Agile<T> in agile classes.

The following example shows how to use `Agile<T>` so that you can safely ignore the warning.

```
#include <agile.h>
ref class MyOptions
    {
    public:
        property Windows::Security::Credentials::UI::CredentialPickerOptions^ Options

        {
            Windows::Security::Credentials::UI::CredentialPickerOptions^ get()
            {
                return m_myOptions.Get();
            }
        }
    private:
        Platform::Agile<Windows::Security::Credentials::UI::CredentialPickerOptions^> m_myOptions;

    };
```

Notice that `Agile` cannot be passed as a return value or parameter in a ref class. The `Agile<T>::Get()` method returns a handle-to-object (^) that you can pass across the application binary interface (ABI) in a public method

or property.

When you create a reference to an in-proc Windows Runtime class that has a marshaling behavior of "None", the compiler issues warning C4451 but doesn't suggest that you consider using `Platform::Agile<T>`. The compiler can't offer any help beyond this warning, so it's your responsibility to use the class correctly and ensure that your code calls STA components only from the user-interface thread, and MTA components only from a background thread.

Authoring agile Windows Runtime components

When you define a ref class in C++/CX, it's agile by default—that is, it has `ThreadingModel`=Both and `MarshallingType`=Agile. If you're using the Windows Runtime C++ Template Library, you can make your class agile by deriving from `FtmBase`, which uses the `FreeThreadedMarshaller`. If you author a class that has `ThreadingModel`=Both or `ThreadingModel`=MTA, make sure that the class is thread-safe.

You can modify the threading model and marshaling behavior of a ref class. However, if you make changes that render the class non-agile, you must understand the implications that are associated with those changes.

The following example shows how to apply `MarshalingBehavior` and `ThreadingModel` attributes to a runtime class in a Windows Runtime class library. When an app uses the DLL and uses the `ref new` keyword to activate a `MySTAClass` class object, the object is activated in a single-threaded apartment and doesn't support marshaling.

```
using namespace Windows::Foundation::Metadata;
using namespace Platform;

[Threading(ThreadingModel::STA)]
[MarshalingBehavior(MarshalingType::None)]
public ref class MySTAClass
{
};
```

An unsealed class must have marshaling and threading attribute settings so that the compiler can verify that derived classes have the same value for these attributes. If the class doesn't have the settings set explicitly, the compiler generates an error and fails to compile. Any class that's derived from an unsealedclass generates a compiler error in either of these cases:

- The `ThreadingModel` and `MarshallingBehavior` attributes are not defined in the derived class.
- The values of the `ThreadingModel` and `MarshallingBehavior` attributes in the derived class don't match those in the base class.

The threading and marshaling information that's required by a third-party Windows Runtime component is specified in the app manifest registration information for the component. We recommend that you make all of your Windows Runtime components agile. This ensures that client code can call your component from any thread in the app, and improves the performance of those calls because they are direct calls that have no marshaling. If you author your class in this way, then client code doesn't have to use `Platform::Agile<T>` to consume your class.

See also

ThreadingModel
MarshallingBehavior

Weak references and breaking cycles (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

In any type system that's based on reference-counting, references to types can form *cycles*—that is, one object refers to a second object, the second object refers to a third object, and so on until some final object refers back to the first object. In a cycle, objects can't be deleted correctly when one object's reference count becomes zero. To help you solve this problem, C++/CX provides the Platform::WeakReference Class class. A `WeakReference` object supports the Resolve method, which returns null if the object no longer exists, or throws an Platform::InvalidCastException if the object is alive but is not of type `T`.

One scenario in which `WeakReference` must be used is when the `this` pointer is captured in a lambda expression that's used to define an event handler. We recommend that you use named methods when you define event handlers, but if you want to use a lambda for your event handler—or if you have to break a reference counting cycle in some other situation—use `WeakReference`. Here's an example:

```
using namespace Platform::Details;
using namespace Windows::UI::Xaml;
using namespace Windows::UI::Xaml::Input;
using namespace Windows::UI::Xaml::Controls;

Class1::Class1()
{
    // Class1 has a reference to m_Page
    m_Page = ref new Page();

    // m_Page will have a reference to this Class1
    // so create a weak reference to this
    WeakReference wr(this);
    m_Page->DoubleTapped += ref new DoubleTappedEventHandler(
        [wr](Object^ sender, DoubleTappedRoutedEventArgs^ args)
    {
       // Use the weak reference to get the object
       Class1^ c = wr.Resolve<Class1>();
       if (c != nullptr)
       {
           c->m_eventFired = true;
       }
       else
       {
           // Inform the event that this handler should be removed
           // from the subscriber list
           throw ref new DisconnectedException();
       }
    });
}

}
```

When an event handler throws `DisconnectedException`, it causes the event to remove the handler from the subscriber list.

Namespaces Reference (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

The articles in this section of the documentation describe namespaces that support the compiler for C++/CX.

Compiler-supplied namespaces

To simplify the coding of programs that target the Windows Runtime, the C++/CX compiler and its supporting header files provide namespaces that define a wide range of types. The namespaces define the built-in numeric types; strings, arrays, and collections; C++ exceptions that represent Windows Runtime errors; and language-specific enhancements to standard Windows Runtime types.

Related topics

TITLE	DESCRIPTION
default namespace	Contains descriptions of built-in, fundamental types.
Platform namespace	Contains descriptions of types that you can use, and also internal types that are used only by the compiler infrastructure.
Windows::Foundation::Collections Namespace	Contains descriptions of enhancements and extensions to the Windows Runtime `Windows::Foundation::Collections` namespace.

See also

C++/CX Language Reference

default namespace

9/21/2022 • 2 minutes to read • Edit Online

The `default` namespace scopes the built-in types that are supported by C++/CX.

Syntax

```
namespace default;
```

Members

All built-in types inherit the following members.

NAME	DESCRIPTION
default::(type_name)::Equals	Determines whether the specified object is equal to the current object.
default::(type_name)::GetHashCode	Returns the hash code for this instance.
default::(type_name)::GetType	Returns a string that represents the current type.
default::(type_name)::ToString	Returns a string that represents the current type.

Built-in types

NAME	DESCRIPTION
`char16`	A 16-bit nonnumeric value that represents a Unicode (UTF-16) code point.
`float32`	A 32-bit IEEE 754 floating-point number.
`float64`	A 64-bit IEEE 754 floating-point number.
`int16`	A 16-bit signed integer.
`int32`	A 32-bit signed integer.
`int64`	A 64-bit signed integer.
`int8`	An 8-bit signed numeric value.
`uint16`	A 16-bit unsigned integer.
`uint32`	A 32-bit unsigned integer.
`uint64`	A 64-bit unsigned integer.

NAME	DESCRIPTION
`uint8`	An 8-bit unsigned numeric value.

Requirements

Header: vccorlib.h

See also

C++/CX Language Reference

default::(type_name)::Equals Method

9/21/2022 • 2 minutes to read • Edit Online

Determines whether the specified object is equal to the current object.

Syntax

```
bool Equals(
    Object^ obj
)
```

Parameters

obj
The object to compare.

Return Value

`true` if the objects are equal, otherwise `false`.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: default

Header: vccorlib.h

See also

default namespace

default::(type_name)::GetHashCode Method

9/21/2022 • 2 minutes to read • Edit Online

Returns the hash code for this instance.

Syntax

```
public:int GetHashCode();
```

Return Value

The hash code for this instance.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: default

Header: vccorlib.h

See also

default namespace

default::(type_name)::GetType Method

9/21/2022 • 2 minutes to read • Edit Online

Returns a Platform::Type^ that represents the current type.

Syntax

```
Platform::Type^ GetType();
```

Return Value

A Platform::Type^ object that represents the current object.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: default

Header: vccorlib.h

See also

default namespace

default::(type_name)::ToString Method

9/21/2022 • 2 minutes to read • Edit Online

Returns a string that represents the current type.

Syntax

```
String^ ToString();
```

Return Value

A string that represents the current object.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: default

Header: vccorlib.h

See also

default namespace

Platform namespace (C++/CX)

9/21/2022 • 3 minutes to read • Edit Online

Contains built-in types that are compatible with the Windows Runtime.

Syntax

```
using namespace Platform;
```

Members

Attributes

The Platform namespace contains attributes, classes, enumerations, interfaces, and structures. Platform also contains nested namespaces.

ATTRIBUTE	DESCRIPTION
Flags	Indicates that an enumeration can be treated as a bit field; that is, a set of flags.
MTAThread	Indicates that the threading model for an application is multi-threaded apartment (MTA).
STAThread	Indicates that the threading model for an application is single-threaded apartment (STA).

Classes

The Platform namespace has the following classes.

CLASS	DESCRIPTION
Platform::AccessDeniedException Class	Raised when access is denied to a resource or feature.
Platform::Agile Class	Represents a non-agile object as an agile object.
Platform::Array Class	Represents a one-dimensional, modifiable array.
Platform::ArrayReference Class	Represents an array whose initialization is optimized to minimize copying operations.
Platform::Box Class	Used to declare a boxed type that encapsulates a value type such as Windows::Foundation::DateTime or int64 when that type is passed across the application binary interface (ABI) or stored in a variable of type Platform::Object^.
Platform::ChangedStateException Class	Thrown when methods of a collection iterator or a collection view are called after the parent collection has changed, invalidating the results of the method.

CLASS	DESCRIPTION
Platform::ClassNotRegisteredException Class	Thrown when a COM class has not been registered.
Platform::COMException Class	Represents the exception that is thrown when an unrecognized value is returned from a COM method call.
Platform::Delegate Class	Represents the signature of a callback function.
Platform::DisconnectedException Class	The object has disconnected from its clients.
Platform::Exception Class	Represents errors that occur during application execution. The base class for exceptions.
Platform::FailureException Class	Thrown when the operation has failed. It is the equivalent of the E_FAIL HRESULT.
Platform::Guid value class	Represents a GUID in the Windows Runtime type system.
Platform::InvalidArgumentException Class	Thrown when one of the arguments provided to a method is not valid.
Platform::InvalidCastException Class	Thrown in cases of invalid casting or explicit conversion.
Platform::MTAThreadAttribute Class	Indicates that the threading model for an application is multi-threaded apartment (MTA).
Platform::NotImplementedException Class	Thrown if an interface method has not been implemented on the class.
Platform::NullReferenceException Class	Thrown when there is an attempt to dereference a null object reference.
Platform::Object Class	A base class that provides common behavior.
Platform::ObjectDisposedException Class	Thrown when an operation is performed on a disposed object.
Platform::OperationCanceledException Class	Thrown when an operation is aborted.
Platform::OutOfBoundsException Class	Thrown when an operation attempts to access data outside the valid range.
Platform::OutOfMemoryException Class	Thrown when there's insufficient memory to complete the operation.
Platform::STAThreadAttribute Class	Indicates that the threading model for an application is single-threaded apartment (STA).
Platform::String Class	A sequential collection of Unicode characters that is used to represent text.
Platform::StringReference Class	Enables access to string buffers with minimum of copy overhead.

CLASS	DESCRIPTION
Platform::Type Class	Identifies a built-in type by a category enumeration.
Platform::ValueType Class	The base class for instances of value types.
Platform::WeakReference Class	Provides a weak reference to ref class objects that does not increment the reference count.
Platform::WriteOnlyArray Class	Represents a one-dimensional write-only array which is used as an input parameter on methods that implement the FillArray pattern.
Platform::WrongThreadException Class	Thrown when a thread calls via an interface pointer which is for a proxy object that does not belong to the thread's apartment.

Interface implementations

The Platform namespace defines the following interfaces.

INTERFACE	DESCRIPTION
Platform::IBox Interface	Used to pass value types to functions whose parameters are typed as Platform::Object^.
Platform::IBoxArray Interface	Interface used to pass arrays of value types to functions whose parameters are typed as Platform::Array.
Platform::IDisposable Interface	Used to release unmanaged resources.

Enumerations

The Platform namespace has the following enumerations.

INTERFACE	DESCRIPTION
Platform::CallbackContext Enumeration	An enumeration that is used as a parameter of the delegate constructor. It determines whether the callback is to be marshalled to the originating thread or to the caller thread.
Platform::TypeCode Enumeration	Specifies a numeric category that represents a built-in type.

Structures

The Platform namespace has the following structures.

STRUCTURE	DESCRIPTION
Platform::Enum Class	Represents a named constant.
Platform::Guid value class	Represents a GUID.
Platform::IntPtr value class	A signed pointer whose size is appropriate for the platform (32-bit or 64-bit).

STRUCTURE	DESCRIPTION
Platform::SizeT value class	An unsigned data type used to represent the size of an object.
Platform::UIntPtr value class	An unsigned pointer whose size is appropriate for the platform (32-bit or 64-bit).

See also

Platform::Collections Namespace
Platform::Runtime::CompilerServices Namespace
Platform::Runtime::InteropServices Namespace
Platform::Metadata Namespace

Platform::AccessDeniedException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when access to a resource or feature is denied.

Syntax

```
public ref class AccessDeniedException : COMException,    IException,    IPrintable,   IEquatable
```

Remarks

If you hit this exception, ensure that you have requested the appropriate capability and made the required declarations in the package manifest of your app. For more information, see COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::Agile Class

9/21/2022 • 4 minutes to read • Edit Online

Represents an object that has a MashalingBehavior=Standard as an agile object, which greatly reduces the chances for runtime threading exceptions. The `Agile<T>` enables the non-agile object to call, or be called from, the same or a different thread. For more information, see Threading and Marshaling.

Syntax

```
template <typename T>
class Agile;
```

Parameters

T

The typename for the non-agile class.

Remarks

Most of the classes in the Windows Runtime are agile. An agile object can call, or be called by, an in-proc or out-of-proc object in the same or a different thread. If an object is not agile, wrap the non-agile object in a `Agile<T>` object, which is agile. Then the `Agile<T>` object can be marshaled, and the underlying non-agile object can be used.

The `Agile<T>` class is a native, standard C++ class and requires `agile.h`. It represents the non-agile object and the Agile object's *context*. The context specifies an agile object's threading model and marshaling behavior. The operating system uses the context to determine how to marshal an object.

Members

Public Constructors

NAME	DESCRIPTION
Agile::Agile	Initializes a new instance of the Agile class.
Agile::~Agile Destructor	Destroys the current instance of the Agile class.

Public Methods

NAME	DESCRIPTION
Agile::Get	Returns a handle to the object that is represented by the current Agile object.
Agile::GetAddressOf	Reinitializes the current Agile object, and then returns the address of a handle to an object of type `T`.
Agile::GetAddressOfForInOut	Returns the address of a handle to the object represented by the current Agile object.
Agile::Release	Discards the current Agile object's underlying object and context.

Public Operators

NAME	DESCRIPTION
Agile::operator->	Retrieves a handle to the object represented by the current Agile object.
Agile::operator=	Assigns the specified value to the current Agile object.

Inheritance Hierarchy

`Object`

`Agile`

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Header: agile.h

Agile::Agile Constructor

Initializes a new instance of the Agile class.

Syntax

```
Agile();
Agile(T^ object);
Agile(const Agile<T>& object);
Agile(Agile<T>&& object);
```

Parameters

T

A type specified by the template typename parameter.

object

In the second version of this constructor, an object used to initialize a new Agile instance. In the third version, the object that is copied to the new Agile instance. In the fourth version, the object that is moved to the new Agile instance.

Remarks

The first version of this constructor is the default constructor. The second version initializes new Agile instance class from the object specified by the `object` parameter. The third version is the copy constructor. The fourth version is the move constructor. This constructor cannot throw exceptions.

Agile::~Agile Destructor

Destroys the current instance of the Agile class.

Syntax

```
~Agile();
```

Remarks

This destructor also releases the object represented by the current Agile object.

Agile::Get Method

Returns a handle to the object that is represented by the current Agile object.

Syntax

```
T^ Get() const;
```

Return Value

A handle to the object that is represented by the current Agile object.

The type of the return value is actually an undisclosed internal type. A convenient way to hold the return value is to assign it to a variable that is declared with the `auto` type deduction keyword. For example, `auto x = myAgileTvariable->Get();`.

Agile::GetAddressOf Method

Reinitializes the current Agile object, and then returns the address of a handle to an object of type `T`.

Syntax

```
T^* GetAddressOf() throw();
```

Parameters

T

A type specified by the template typename parameter.

Return Value

The address of a handle to an object of type `T`.

Remarks

This operation releases the current representation of a object of type `T`, if any; reinitializes the Agile object's data members; acquires the current threading context; and then returns the address of a handle-to-object variable that can represent a non-agile object. To cause an Agile class instance to represent an object, use the assignment operator (Agile::operator=) to assign the object to the Agile class instance.

Agile::GetAddressOfForInOut Method

Returns the address of a handle to the object represented by the current Agile object.

Syntax

```
T^* GetAddressOfForInOut()  throw();
```

Parameters

T

A type specified by the template typename parameter.

Return Value

The address of a handle to the object represented by the current Agile object.

Remarks

This operation acquires the current threading context and then returns the address of a handle to the underlying the object.

Agile::Release Method

Discards the current Agile object's underlying object and context.

Syntax

```
void Release() throw();
```

Remarks

The current Agile object's underlying object and context are discarded, if they exist, and then the value of the Agile object is set to null.

`Agile::operator->` Operator

Retrieves a handle to the object represented by the current Agile object.

Syntax

```
T^ operator->() const throw();
```

Return Value

A handle to the object represented by the current Agile object.

This operator actually returns an undisclosed internal type. A convenient way to hold the return value is to assign it to a variable that is declared with the `auto` type deduction keyword.

Agile::operator= Operator

Assigns the specified object to the current Agile object.

Syntax

```
Agile<T> operator=( T^ object ) throw();
Agile<T> operator=( const Agile<T>& object ) throw();
Agile<T> operator=( Agile<T>&& object ) throw();
T^ operator=( IUnknown* lp ) throw();
```

Parameters

T

The type specified by the template typename.

object

The object or handle to an object that is copied or moved to the current Agile object.

lp

The IUnknown interface pointer of a object.

Return Value

A handle to an object of type `T`

Remarks

The first version of the assignment operator copies a handle to a reference type to the current Agile object. The second version copies a reference to an Agile type to the current Agile object. The third version moves an Agile type to the current Agile object. The fourth version moves a pointer to a COM object to the current Agile object.

The assignment operation automatically persists the context of the current Agile object.

See also

Platform Namespace

Platform::Array Class

9/21/2022 • 2 minutes to read • Edit Online

Represents a one-dimensional, modifiable array that can be received and passed across the application binary interface (ABI).

Syntax

```
template <typename T>
private ref class Array<TArg, 1> :
    public WriteOnlyArray<TArg, 1>,
    public IBoxArray<TArg>
```

Members

Platform::Array inherits all its methods from Platform::WriteOnlyArray Class and implements the `Value` property of the Platform::IBoxArray Interface.

Public Constructors

NAME	DESCRIPTION
Array Constructors	Initializes a one-dimensional, modifiable array of types specified by the class template parameter, *T*.

Methods

See Platform::WriteOnlyArray Class.

Properties

NAME	DESCRIPTION
Array::Value	Retrieves a handle to the current array.

Remarks

The Array class is sealed and cannot be inherited.

The Windows Runtime type system does not support the concept of jagged arrays and therefore you cannot pass an `IVector<Platform::Array<T>>` as a return value or method parameter. To pass a jagged array or a sequence of sequences across the ABI, use `IVector<IVector<T>^>`.

For more information about when and how to use Platform::Array, see Array and WriteOnlyArray.

This class is defined in the vccorlib.h header, which is automatically included by the compiler. It is visible in IntelliSense but not in Object Browser because it is not a public type defined in platform.winmd.

Requirements

Compiler option: **/ZW**

Array Constructors

Initializes a one-dimensional, modifiable array of types specified by the class template parameter, *T*.

Syntax

```
Array(unsigned int size);
Array(T* data, unsigned int size);
```

Parameters

T

Class template parameter.

size

The number of elements in the array.

data

A pointer to an array of data of type `T` that is used to initialize this Array object.

Remarks

For more information about how to create instances of Platform::Array, see Array and WriteOnlyArray.

Array::get Method

Retrieves a reference to the array element at the specified index location.

Syntax

```
T& get(unsigned int index)  const;
```

Parameters

index

A zero-based index that identifies an element in the array. The minimum index is 0 and the maximum index is the value specified by the `size` parameter in the Array constructor.

Return Value

The array element specified by the `index` parameter.

Array::Value Property

Retrieves a handle to the current array.

Syntax

```
property Array^ Value;
```

Return Value

A handle to the current array.

See also

Platform namespace
Array and WriteOnlyArray

Platform::ArrayReference Class

9/21/2022 • 2 minutes to read • Edit Online

`ArrayReference` is an optimization type that you can substitute for Platform::Array^ in input parameters when you want to fill a C-style array with the input data.

Syntax

```
class ArrayReference
```

Members

Public Constructors

NAME	DESCRIPTION
ArrayReference::ArrayReference	Initializes a new instance of the `ArrayReference` class.

Public Operators

NAME	DESCRIPTION
ArrayReference::operator() Operator	Converts this `ArrayReference` to a `Platform::Array<T>^*`.
ArrayReference::operator= Operator	Assigns the contents of another `ArrayReference` to this instance.

Exceptions

Remarks

By using `ArrayReference` to fill a C-style array, you avoid the extra copy operation that would be involved in copying first to a `Platform::Array` variable, and then into the C-style array. When you use `ArrayReference`, there is only one copy operation. For a code example, see Array and WriteOnlyArray.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Header: vccorlib.h

ArrayReference::ArrayReference Constructor

Initializes a new instance of the Platform::ArrayReference class.

Syntax

```
ArrayReference(TArg* ataArg, unsigned int sizeArg, bool needsInitArg = false);
ArrayReference(ArrayReference&& otherArg)
```

Parameters

dataArg
A pointer to the array data.

sizeArg
The number of elements in the source array.

otherArg
An `ArrayReference` object whose data will be moved to initialize the new instance.

Remarks

ArrayReference::operator= Operator

Assigns the specified object to the current Platform::ArrayReference object by using move semantics.

Syntax

```
ArrayReference& operator=(ArrayReference&& otherArg);
```

Parameters

otherArg
The object that is moved to the current `ArrayReference` object.

Return Value

A reference to an object of type `ArrayReference`.

Remarks

`Platform::ArrayReference` is a standard C++ class template, not a ref class.

ArrayReference::operator() Operator

Converts the current Platform::ArrayReference object back to a Platform::Array class.

Syntax

```
Array<TArg>^ operator ();
```

Return Value

A handle-to-object of type `Array<TArg>^`

Remarks

Platform::ArrayReference is a standard C++ class template, and Platform::Array is a ref class.

See also

Platform namespace

Platform::Boolean value class

9/21/2022 • 2 minutes to read • Edit Online

Represents a Boolean value. The equivalent of `bool`.

Syntax

```
public value struct Boolean
```

Members

Boolean has the Equals(), GetHashCode(), and ToString() methods derived from the Platform::Object Class, and the GetTypeCode() method derived from the Platform::Type Class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform namespace

Platform::Box Class

9/21/2022 • 2 minutes to read • Edit Online

Enables a value type such as `Windows::Foundation::DateTime` or a scalar type such as `int` to be stored in a `Platform::Object` type. It is usually not necessary to use `Box` explicitly because boxing happens implicitly when you cast a value type to `Object^`.

Syntax

```
ref class Box abstract;
```

Requirements

Header: vccorlib.h

Namespace: Platform

Members

MEMBER	DESCRIPTION
Box	Creates a `Box` that can encapsulate a value of the specified type.
`operator Box<const T>^`	Enables boxing conversions from a `const` value class `T` or `enum` class `T` to `Box<T>`.
`operator Box<const volatile T>^`	Enables boxing conversions from a `const volatile` value class `T` or `enum` type `T` to `Box<T>`.
`operator Box<T>^`	Enables boxing conversions from a value class `T` to `Box<T>`.
`operator Box<volatile T>^`	Enables boxing conversions from a `volatile` value class `T` or `enum` type `T` to `Box<T>`.
Box::operator T	Enables boxing conversions from a value class `T` or `enum` class `T` to `Box<T>`.
Value property	Returns the value that is encapsulated in the `Box` object.

Box::Box Constructor

Creates a `Box` that can encapsulate a value of the specified type.

Syntax

```
Box(T valueArg);
```

Parameters

valueArg

The type of value to be boxed—for example, `int`, `bool`, `float64`, `DateTime`.

`Box::operator Box<const T>^` Operator

Enables boxing conversions from a `const` value class `T` or `enum` class `T` to `Box<T>`.

Syntax

```
operator Box<const T>^(const T valueType);
```

Parameters

T

Any value class, value struct, or enum type. Includes the built-in types in the default namespace.

Return Value

A `Platform::Box<T>^` instance that represents the original value boxed in a ref class.

`Box::operator Box<const volatile T>^` Operator

Enables boxing conversions from a `const volatile` value class `T` or `enum` type `T` to `Box<T>`.

Syntax

```
operator Box<const volatile T>^(const volatile T valueType);
```

Parameters

T

Any enum type, value class, or value struct. Includes the built-in types in the default namespace.

Return Value

A `Platform::Box<T>^` instance that represents the original value boxed in a ref class.

`Box::operator Box<T>^` Operator

Enables boxing conversions from a value class `T` to `Box<T>`.

Syntax

```
operator Box<const T>^(const T valueType);
```

Parameters

T

Any enum type, value class, or value struct. Includes the built-in types in the default namespace.

Return Value

A `Platform::Box<T>^` instance that represents the original value boxed in a ref class.

`Box::operator Box<volatile T>^` Operator

Enables boxing conversions from a `volatile` value class `T` or `enum` type `T` to `Box<T>`.

Syntax

```
operator Box<volatile T>^(volatile T valueType);
```

Parameters

T

Any enum type, value class, or value struct. Includes the built-in types in the default namespace.

Return Value

A `Platform::Box<T>^` instance that represents the original value boxed in a ref class.

Box::operator T Operator

Enables boxing conversions from a value class `T` or `enum` class `T` to `Box<T>`.

Syntax

```
operator Box<T>^(T valueType);
```

Parameters

T

Any enum type, value class, or value struct. Includes the built-in types in the default namespace.

Return Value

A `Platform::Box<T>^` instance that represents the original value boxed in a ref class.

Box::Value Property

Returns the value that is encapsulated in the `Box` object.

Syntax

```
virtual property T Value{
   T get();
}
```

Return Value

Returns the boxed value with the same type as it originally had before it was boxed.

See also

Platform namespace
Boxing

Platform::CallbackContext Enumeration

9/21/2022 • 2 minutes to read • Edit Online

Specifies the thread context in which a callback function (event handler) executes.

Syntax

```
enum class CallbackContext {};
```

Members

TYPE CODE	DESCRIPTION
Any	The callback function can execute on any thread context.
Same	The callback function can execute on only the thread context that started the asynchronous operation.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

Platform::ChangedStateException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when the internal state of an object has changed, thereby invalidating the results of the method.

Syntax

```
public ref class ChangedStateException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

One example where this exception is thrown is when methods of a collection iterator or a collection view are called after the parent collection has changed, invalidating the results of the method.

For more information, see the COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::ClassNotRegisteredException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when a COM class has not been registered.

Syntax

```
public ref class ClassNotRegisteredException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

For more information, see the COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::COMException Class

9/21/2022 • 2 minutes to read • Edit Online

Represents COM errors that occur during application execution. COMException is the base class for a set of predefined, standard exceptions.

Syntax

```
public ref class COMException : Exception,    IException,    IPrintable,    IEquatable
```

Members

The COMException class inherits from the Object class and the IException, IPrintable, and IEquatable interfaces.

COMException also has the following types of members.

Constructors

MEMBER	DESCRIPTION
COMException	Initializes a new instance of the COMException class.

Methods

The COMException class inherits the Equals(), Finalize(), GetHashCode(), GetType(), MemberwiseClose(), and ToString() methods from the Platform::Object Class.

Properties

The COMException class has the following properties.

MEMBER	DESCRIPTION
Exception::HResult	The HRESULT that corresponds to the exception.
Exception::Message	Message that describes the exception.

Derived Exceptions

The following predefined exceptions are derived from COMException. They differ from COMException only in their name, the name of their constructor, and their underlying HRESULT value.

NAME	UNDERLYING HRESULT	DESCRIPTION
COMException	*user-defined hresult*	Thrown when an unrecognized HRESULT is returned from a COM method call.
AccessDeniedException	E_ACCESSDENIED	Thrown when access is denied to a resource or feature.

NAME	UNDERLYING HRESULT	DESCRIPTION
ChangedStateException	E_CHANGED_STATE	Thrown when methods of a collection iterator or a collection view are called after the parent collection has changed, invalidating the results of the method.
ClassNotRegisteredException	REGDB_E_CLASSNOTREG	Thrown when a COM class has not been registered.
DisconnectedException	RPC_E_DISCONNECTED	Thrown when an object is disconnected from its clients.
FailureException	E_FAIL	Thrown when an operation fails.
InvalidArgumentException	E_INVALIDARG	Thrown when one of the arguments provided to a method is not valid.
InvalidCastException	E_NOINTERFACE	Thrown when a type can't be cast to another type.
NotImplementedException	E_NOTIMPL	Thrown if an interface method hasn't been implemented on a class.
NullReferenceException	E_POINTER	Thrown when there is an attempt to dereference a null object reference.
OperationCanceledException	E_ABORT	Thrown when an operation is aborted.
OutOfBoundsException	E_BOUNDS	Thrown when an operation attempts to access data outside the valid range.
OutOfMemoryException	E_OUTOFMEMORY	Thrown when there's insufficient memory to complete the operation.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

COMException::COMException Constructor

Intializes a new instance of the COMException class.

Syntax

```
COMException( int hresult )
```

Parameters

hresult

The error HRESULT that is represented by the exception.

COMException::HResult Property

The HRESULT that corresponds to the exception.

Syntax

```
public:
    property int HResult { int get();}
```

Property Value

An HRESULT value that specifies the error.

Remarks

For more information about how to interpret the HRESULT value, see Structure of COM Error Codes.

COMException::Message Property

Message that describes the exception.

Syntax

```
public:property String^ Message {    String^ get();}
```

Property Value

A description of the exception.

See also

Platform namespace

Platform::Delegate Class

9/21/2022 • 2 minutes to read • Edit Online

Represents a function object.

Syntax

```
public delegate void delegate_name();
```

Members

The Delegate class has the Equals(), GetHashCode(), and ToString() methods derived from the Platform::Object Class.

Remarks

Use the delegate keyword to create delegates; do not use Platform::Delegate explicitly. For more information, see Delegates. For an example of how to create and consume a delegate, see Creating Windows Runtime Components in C++.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform namespace

Platform::DisconnectedException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when a COM proxy object attempts to reference a COM server that no longer exists

Syntax

```
public ref class DisconnectedException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

When class A references another class (class B) that is in a separate process, class A requires a proxy object to communicate with the out-of-process COM server that holds class B. Sometimes the server can go out of memory without class A knowing about it. In that case the RPC_E_DISCONNECTED exception is thrown and it gets translated to Platform::DisconnectedException. One scenario in which is occurs is when an event source invokes a delegate that was passed to it, but the delegate has been destroyed at some point after it subscribed to the event. When this happens, the event source removes that delegate from its invocation list.

For more information, see the COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::Enum Class

9/21/2022 • 2 minutes to read • Edit Online

A value class that represents a set of named constants.

Syntax

```
public class Enum
```

Members

The Enum class inherits the Equals(), GetHashCode(), and ToString() methods from the Platform::Object Class.

Remarks

Use the public enum class keyword to create enumerations. Do not use the Platform::Enum type explicitly. For more information, see Enums.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform namespace

Platform::Exception Class

9/21/2022 • 2 minutes to read • Edit Online

Represents errors that occur during application execution. Custom exception classes can't be derived from `Platform::Exception`. If you require a custom exception, you can use `Platform::COMException` and specify an app-specific HRESULT.

Syntax

```
public ref class Exception : Object,    IException,    IPrintable,    IEquatable
```

Members

The `Exception` class inherits from the `Object` class and the `IException`, `IPrintable`, and `IEquatable` interfaces.

The `Exception` class also has the following kinds of members.

Constructors

MEMBER	DESCRIPTION
Exception::Exception	Initializes a new instance of the `Exception` class.

Methods

The `Exception` class inherits the `Equals()`, `Finalize()`, `GetHashCode()`, `GetType()`, `MemberwiseClose()`, and `ToString()` methods from the Platform::Object Class. The `Exception` class also has the following method.

MEMBER	DESCRIPTION
Exception::CreateException	Creates an exception that represents the specified HRESULT value.

Properties

The Exception class also has the following properties.

MEMBER	DESCRIPTION
Exception::HResult	The HRESULT that corresponds to the exception.
Exception::Message	A message that describes the exception. This value is read-only and cannot be modified after the `Exception` is constructed.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

Exception::CreateException Method

Creates a Platform::Exception^ from a specified HRESULT value.

Syntax

```
Exception^ CreateException(int32 hr);
Exception^ CreateException(int32 hr, Platform::String^ message);
```

Parameters

hr
An HRESULT value that you typically get from a call to a COM method. If the value is 0, which is equal to S_OK, this method throws Platform::InvalidArgumentException because COM methods that succeed should not throw exceptions.

message
A string that describes the error.

Return Value

An exception that represents the error HRESULT.

Remarks

Use this method to create an exception out of an HRESULT that is returned, for example, from a call to a COM interface method. You can use the overload that takes a String^ parameter to provide a custom message.

It is strongly recommended to use CreateException to create a strongly-typed exception rather than creating a Platform::COMException that merely contains the HRESULT.

Exception::Exception Constructor

Intializes a new instance of the Exception class.

Syntax

```
Exception(int32 hresult);
Exception(int32 hresult, ::Platform::String^ message);
```

Parameters

hresult
The error HRESULT that is represented by the exception.

message
A user-specified message, such as prescriptive text, that is associated with the exception. In general you should prefer the second overload in order to provide a descriptive message that is as specific as possible about how and why the error has occurred.

Exception::HResult Property

The HRESULT that corresponds to the exception.

Syntax

```
public:
    property int HResult { int get(); }
```

Property Value

An HRESULT value.

Remarks

Most exceptions start out as COM errors, which are returned as HRESULT values. C++/CX converts these values into Platform::Exception^ objects, and this property stores the value of the original error code.

Exception::Message Property

Message that describes the error.

Syntax

```
public:property String^ Message;
```

Property Value

In exceptions that originate in the Windows Runtime, this is a system-supplied description of the error.

Remarks

In Windows 8, this property is read-only because exceptions in that version of the Windows Runtime are transported across the ABI only as HRESULTS. In Windows 8.1, richer exception information is transported across the ABI and you can provide a custom message that other components can access programmatically. For more information, see Exceptions (C++/CX).

See also

Platform namespace

Platform::FailureException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when the operation has failed. It is the equivalent of the E_FAIL HRESULT.

Syntax

```
public ref class FailureException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

For more information, see the COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::Guid value class

9/21/2022 • 2 minutes to read • Edit Online

Represents a [GUID](/windows/win32/api/guiddef/ns-guiddef-guid type in the Windows Runtime type system.

Syntax

```
public value struct Guid
```

Members

`Platform::Guid` has the `Equals()`, `GetHashCode()`, and `ToString()` methods derived from the Platform::Object Class, and the `GetTypeCode()` method derived from the Platform::Type Class. `Platform::Guid` also has the following members.

MEMBER	DESCRIPTION
Guid	Initializes a new instance of a `Platform::Guid`.
operator==	Equals operator.
operator!=	Not equals operator.
`operator<`	Less than operator.
operator()	Converts a `Platform::Guid` to a `GUID`.

Remarks

To generate a new `Platform::Guid`, use the Windows::Foundation::GuidHelper::CreateNewGuid static method.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

Guid::Guid Constructors

Initializes a new instance of a `Platform::Guid`.

Syntax

```
Guid(
    unsigned int a,
    unsigned short b,
    unsigned short c,
    unsigned char d,
    unsigned char e,
    unsigned char f,
    unsigned char g,
    unsigned char h,
    unsigned char i,
    unsigned char j,
    unsigned char k );

Guid(GUID m);

Guid(
    unsigned int a,
    unsigned short b,
    unsigned short c,
    Array<unsigned char>^ n );
```

Parameters

a
The first 4 bytes of the `GUID`.

b
The next 2 bytes of the `GUID`.

c
The next 2 bytes of the `GUID`.

d
The next byte of the `GUID`.

e
The next byte of the `GUID`.

f
The next byte of the `GUID`.

g
The next byte of the `GUID`.

h
The next byte of the `GUID`.

i
The next byte of the `GUID`.

j
The next byte of the `GUID`.

k
The next byte of the `GUID`.

m
A `GUID` in the form a GUID structure.

n
The remaining 8 bytes of the `GUID`.

Guid::operator== Operator

Compares two `Platform::Guid` instances for equality.

Syntax

```
static bool Platform::Guid::operator==(Platform::Guid guid1, Platform::Guid guid2);
```

Parameters

guid1
The first `Platform::Guid` to compare.

guid2
The second `Platform::Guid` to compare.

Return Value

True if the two `Platform::Guid` instances are equal.

Remarks

Prefer using the `==` operator instead of the Windows::Foundation::GuidHelper::Equals static method.

Guid::operator!= Operator

Compares two `Platform::Guid` instances for inequality.

Syntax

```
static bool Platform::Guid::operator!=(Platform::Guid guid1, Platform::Guid guid2);
```

Parameters

guid1
The first `Platform::Guid` to compare.

guid2
The second `Platform::Guid` to compare.

Return Value

True if the two `Platform::Guid` instances are not equal.

`Guid::operator<` Operator

Compares two `Platform::Guid` instances for ordering.

Syntax

```
static bool Platform::Guid::operator<(Platform::Guid guid1, Platform::Guid guid2);
```

Parameters

guid1
The first `Platform::Guid` to compare.

guid2
The second `Platform::Guid` to compare.

Return Value

True if *guid1* is ordered before *guid2*. The ordering is lexicographic after treating each `Platform::Guid` as if it's an array of four 32-bit unsigned values. This isn't the ordering used by SQL Server or the .NET Framework, nor is it the same as lexicographical ordering by string representation.

This operator is provided so that `Guid` objects can be more easily consumed by the C++ standard library.

Guid::operator() Operator

Implicitly converts a `Platform::Guid` to a GUID structure.

Syntax

```
const GUID& Platform::Guid::operator();
```

Return Value

A GUID structure.

See also

Platform namespace

Platform::IBox Interface

9/21/2022 • 2 minutes to read • Edit Online

The Platform::IBox interface is the C++ name for the `Windows::Foundation::IReference` interface.

Syntax

```
template <typename T>
interface class IBox
```

Parameters

T

The type of the boxed value.

Remarks

The `IBox<T>` interface is primarily used internally to represent nullable value types, as described in Value classes and structs (C++/CX). The interface is also used to box value types that are passed to C++ methods that take parameters of type `Object^`. You can explicitly declare an input parameter as `IBox<SomeValueType>`. For an example, see Boxing.

Requirements

Members

The `Platform::IBox` interface inherits from the Platform::IValueType interface. `IBox` has these members:

Properties

METHOD	DESCRIPTION
Value	Returns the unboxed value that was previously stored in this `IBox` instance.

IBox::Value Property

Returns the value that was originally stored in this object.

Syntax

```
property T Value {T get();}
```

Parameters

T

The type of the boxed value.

Property Value/Return Value

Returns the value that was originally stored in this object.

Remarks

For an example, see Boxing.

See also

Platform namespace

Platform::IBoxArray Interface

9/21/2022 • 2 minutes to read • Edit Online

`IBoxArray` is the wrapper for arrays of value types that are passed across the application binary interface (ABI) or stored in collections of `Platform::Object^` elements such as those in XAML controls.

Syntax

```
template <typename T>
interface class IBoxArray
```

Parameters

T

The type of the boxed value in each array element.

Remarks

`IBoxArray` is the C++/CX name for `Windows::Foundation::IReferenceArray`.

Members

The `IBoxArray` interface inherits from the `IValueType` interface. `IBoxArray` also has these members:

METHOD	DESCRIPTION
Value	Returns the unboxed array that was previously stored in this `IBoxArray` instance.

IBoxArray::Value Property

Returns the value that was originally stored in this object.

Syntax

```
property T Value {T get();}
```

Parameters

T

The type of the boxed value.

Property Value/Return Value

Returns the value that was originally stored in this object.

Remarks

For an example, see Boxing.

See also

Array and WriteOnlyArray

Platform::IDisposable Interface

9/21/2022 • 2 minutes to read • Edit Online

Used to release unmanaged resources.

Syntax

```
public interface class IDisposable
```

Attributes

GuidAttribute("de0cbaea-8065-4a45-b196-c9d443f9bab3")

VersionAttribute(NTDDI_WIN8)

Members

The IDisposable interface inherits from the IUnknown interface. IDisposable also has the following types of members:

Methods

The IDisposable interface has the following methods.

METHOD	DESCRIPTION
Dispose	Used to release unmanaged resources.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Platform::IntPtr value class

9/21/2022 • 2 minutes to read • Edit Online

Represents an signed pointer or handle, and whose size is platform-specific (32-bit or 64-bit).

Syntax

```
public value struct IntPtr
```

Members

IntPtr has the following members:

MEMBER	DESCRIPTION
IntPtr::IntPtr	Initializes a new instance of IntPtr.
IntPtr::op_explicit Operator	Converts the specified parameter to an IntPtr or a pointer to an IntPtr value.
IntPtr::ToInt32	Converts the current IntPtr to a 32-bit integer.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

IntPtr::IntPtr Constructor

Initializes a new instance of an IntPtr with the specified value.

Syntax

```
IntPtr( __int64 handle-or-pointer );   IntPtr( void* value );   IntPtr( int 32-bit_value );
```

Parameters

value
A 64-bit handle or pointer, or a pointer to a 64-bit value, or a 32-bit value that can be converted to a 64-bit value.

IntPtr::op_explicit Operator

Converts the specified parameter to an IntPtr or a pointer to an IntPtr value.

Syntax

```
static IntPtr::operator IntPtr( void* value1);   static IntPtr::operator IntPtr( int value2);   static
IntPtr::operator void*( IntPtr value3 );
```

Parameters

value1
A pointer to a handle or IntPtr.

value2
An 32-bit integer that can be converted to an IntPtr.

value3
An IntPtr.

Return Value

The first and second operators return an IntPtr. The third operator returns a pointer to the value represented by the current IntPtr.

IntPtr::ToInt32 Method

Converts the current IntPtr value to a 32-bit integer.

Syntax

```
int32 IntPtr::ToInt32();
```

Return Value

A 32-bit integer.

See also

Platform namespace

Platform::InvalidArgumentException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when one of the arguments provided to a method is not valid.

Syntax

```
public ref class InvalidArgumentException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

For more information, see the COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::InvalidCastException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when a cast or explicit conversion is invalid.

Syntax

```
public ref class InvalidCastException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

For more information, see the COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::IValueType Interface

9/21/2022 • 2 minutes to read • Edit Online

`Platform::IValueType` is an infrastructure interface that is implemented by value classes and value structs. Not to be used explicitly in your code.

Syntax

```
interface class IValueType
```

See also

Platform namespace

Platform::MTAThreadAttribute Class

9/21/2022 • 2 minutes to read • Edit Online

Indicates that the threading model for an application is multi-threaded apartment (MTA).

Syntax

```
public ref class MTAThreadAttribute sealed : Attribute
```

Members

Public Constructors

NAME	DESCRIPTION
MTAThreadAttribute Constructor 1 constructor	Initializes a new instance of the class.

Public Methods

The MTAThreadAttribute attribute inherits from Platform::Object Class. MTAThreadAttribute also overloads or has the following members:

NAME	DESCRIPTION
MTAThreadAttribute::Equals	Determines whether the specified object is equal to the current object.
MTAThreadAttribute::GetHashCode	Returns the hash code for this instance.
MTAThreadAttribute::ToString	Returns a string that represents the current object.

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform

MTAThreadAttribute Constructor

Initializes a new instance of the MTAThreadAttribute class.

Syntax

```
public:MTAThreadAttribute();
```

MTAThreadAttribute::Equals

Determines whether the specified object is equal to the current object.

Syntax

```
public:virtual override bool Equals( Object^ obj );
```

Parameters

obj
The object to compare.

Return Value

`true` if the objects are equal; otherwise, `false`.

MTAThreadAttribute::GetHashCode

Returns the hash code for this instance.

Syntax

```
public:int GetHashCode();
```

Return Value

The hash code for this instance.

MTAThreadAttribute::ToString

Returns a string that represents the current object.

Syntax

```
public:String^ ToString();
```

Return Value

A string that represents the current object.

See also

Platform Namespace

Platform::NotImplementedException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when an interface member is not been implemented in a derived type.

Syntax

```
public ref class NotImplementedException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

For more information, see the COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::NullReferenceException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when there is an attempt to dereference a null object reference.

Syntax

```
public ref class NullReferenceException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

For more information, see the COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::Object Class

9/21/2022 • 2 minutes to read • Edit Online

Provides common behavior for ref classes and ref structs in Windows Runtime apps. All ref class and ref struct instances are implicitly convertible to Platform::Object^ and can override its virtual ToString method.

Syntax

```
public ref class Object : Object
```

Members

Public Constructors

NAME	DESCRIPTION
Object::Object	Initializes a new instance of the Object class.

Public Methods

NAME	DESCRIPTION
Object::Equals	Determines whether the specified object is equal to the current object.
Object::GetHashCode	Returns the hash code for this instance.
Object::ReferenceEquals	Determines whether the specified Object instances are the same instance.
ToString	Returns a string that represents the current object. Can be overridden.
GetType	Gets a Platform::Type that describes the current instance.

Inheritance Hierarchy

`Object`

`Object`

Requirements

Header: vccorlib.h

Namespace: Platform

Object::Equals Method

Determines whether the specified object is equal to the current object.

Syntax

```
bool Equals(
    Object^ obj
)
```

Parameters

obj
The object to compare.

Return Value

`true` if the objects are equal, otherwise `false`.

Object::GetHashCode Method

Returns the `IUnknown` * identity value for this instance if it is a COM object, or a computed hash value if it is not a COM object.

Syntax

```
public:int GetHashCode();
```

Return Value

A numeric value that uniquely identifies this object.

Remarks

You can use GetHashCode to create keys for objects in maps. You can compare hash codes by using Object::Equals. If the code path is extremely critical and `GetHashCode` and `Equals` are not sufficiently fast, then you can drop down to the underlying COM layer and do native `IUnknown` pointer comparisons.

Object::GetType Method

Returns a Platform::Type object that describes the runtime type of an object.

Syntax

```
Object::GetType();
```

Property Value/Return Value

A Platform::Type object that describes the runtime type of the object.

Remarks

The static Type::GetTypeCode can be used to get a Platform::TypeCode Enumeration value that represents the current type. This is mostly useful for built-in types. The type code for any ref class besides Platform::String is Object (1).

The Windows::UI::Xaml::Interop::TypeName class is used in the Windows APIs as a language-independent way of passing type information between Windows components and apps. The TPlatform::Type Class has operators for converting between `Type` and `TypeName`.

Use the typeid operator to return a `Platform::Type` object for a class name, for example when navigating between XAML pages:

```
rootFrame->Navigate(TypeName(MainPage::typeid), e->Arguments);
```

Object::Object Constructor

Initializes a new instance of the Object class.

Syntax

```
public:Object();
```

Object::ReferenceEquals Method

Determines whether the specified Object instances are the same instance.

Syntax

```
public:static bool ReferenceEquals(  Object^ obj1,   Object^ obj2);
```

Parameters

obj1
The first object to compare.

obj2
The second object to compare.

Return Value

`true` if the two objects are the same; otherwise, `false`.

Object::ToString Method (C++/CX)

Returns a string that represents the current object.

Syntax

```
public:
virtual String^ ToString();
```

Return Value

A string that represents the current object. You can override this method to provide a custom string message in your ref class or struct:

```
public ref class Tree sealed
{
public:
    Tree(){}
    virtual Platform::String^ ToString() override
    {
      return "I'm a Tree";
    };
};
```

See also

Platform Namespace
Platform::Type Class
Type System

Platform::ObjectDisposedException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when an operation is performed on a disposed object.

Syntax

```
public ref class ObjectDisposedException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

For more information, see COMException.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::OperationCanceledException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when an operation is aborted.

Syntax

```
public ref class OperationCanceledException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

For more information, see the COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::OutOfBoundsException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when an operation attempts to access data outside the valid range.

Syntax

```
public ref class OutOfBoundsException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

For more information, see the COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::OutOfMemoryException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when there's insufficient memory to complete the operation.

Syntax

```
public ref class OutOfMemoryException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

For more information, see the COMException class.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::ReCreateException Method

9/21/2022 • 2 minutes to read • Edit Online

This method is for internal use only and is not intended for user code. Use the Exception::CreateException method instead.

Syntax

```
static Exception^ ReCreateException(int hr)
```

Parameters

hr

Property Value/Return Value

Returns a new Platform::Exception^, based on the specified HRESULT.

Platform::SizeT value class

9/21/2022 • 2 minutes to read • Edit Online

Represents the size of an object. SizeT is an unsigned data type.

Syntax

```
public ref class SizeT sealed : ValueType
```

Members

MEMBER	DESCRIPTION
SizeT::SizeT constructor	Initializes a new instance of the class with the specified value.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

SizeT::SizeT constructor

Initializes a new instance of SizeT with the specified value.

Syntax

```
SizeT( uint32 value1 );   SizeT( void* value2 );
```

Parameters

value1
An unsigned 32-bit value.

value2
Pointer to an unsigned 32-bit value.

See also

Platform namespace

Platform::STAThreadAttribute Class

9/21/2022 • 2 minutes to read • Edit Online

Indicates that the threading model for an application is single-threaded apartment (STA).

Syntax

```
public ref class STAThreadAttribute sealed : Attribute
```

Members

Public Constructors

NAME	DESCRIPTION
STAThreadAttribute constructor 1	Initializes a new instance of the class.

Public Methods

The STAThreadAttribute attribute inherits from Platform::Object Class. STAThreadAttribute also overloads or has the following members:

NAME	DESCRIPTION
STAThreadAttribute::Equals	Determines whether the specified object is equal to the current object.
STAThreadAttribute::GetHashCode	Returns the hash code for this instance.
STAThreadAttribute::ToString	Returns a string that represents the current object.

Inheritance Hierarchy

`Platform`

Requirements

Header: collection.h

Namespace: Platform

STAThreadAttribute constructor

Initializes a new instance of the STAThreadAttribute class.

Syntax

```
public:STAThreadAttribute();
```

STAThreadAttribute::Equals

Determines whether the specified object is equal to the current object.

Syntax

```
public:virtual override bool Equals( Object^ obj );
```

Parameters

obj
The object to compare.

Return Value

`true` if the objects are equal; otherwise, `false`.

STAThreadAttribute::GetHashCode

Returns the hash code for this instance.

Syntax

```
public:int GetHashCode();
```

Return Value

The hash code for this instance.

STAThreadAttribute::ToString

Returns a string that represents the current object.

Syntax

```
public:String^ ToString();
```

Return Value

A string that represents the current object.

See also

Platform Namespace

Platform::String Class

9/21/2022 • 7 minutes to read • Edit Online

Represents a sequential collection of Unicode characters that is used to represent text. For more information and examples, see Strings.

Syntax

```
public ref class String sealed : Object,
    IDisposable,
    IEquatable,
    IPrintable
```

Iterators

Two iterator functions, which are not members of the String class, can be used with the `std::for_each` template function to enumerate the characters in a String object.

MEMBER	DESCRIPTION
`const char16* begin(String^ s)`	Returns a pointer to the beginning of the specified String object.
`const char16* end(String^ s)`	Returns a pointer past the end of the specified String object.

Members

The String class inherits from Object, and the IDisposable, IEquatable, and IPrintable interfaces.

The String class also has the following types of members.

Constructors

MEMBER	DESCRIPTION
String::String	Initializes a new instance of the String class.

Methods

The String class inherits the Equals(), Finalize(), GetHashCode(), GetType(), MemberwiseClose(), and ToString() methods from the Platform::Object Class. String also has the following methods.

METHOD	DESCRIPTION
String::Begin	Returns a pointer to the beginning of the current string.
String::CompareOrdinal	Compares two `String` objects by evaluating the numeric values of the corresponding characters in the two string values represented by the objects.

METHOD	DESCRIPTION
String::Concat	Concatenates the values of two String objects.
String::Data	Returns a pointer to the beginning of the current string.
String::Dispose	Frees or releases resources.
String::End	Returns a pointer past the end of the current string.
String::Equals	Indicates whether the specified object is equal to the current object.
String::GetHashCode	Returns the hash code for this instance.
String::IsEmpty	Indicates whether the current String object is empty.
String::IsFastPass	Indicates whether the current String object is participating in a *fast pass* operation. In a fast pass operation, reference counting is suspended.
String::Length	Retrieves the length of the current String object.
String::ToString	Returns a String object whose value is the same as the current string.

Operators

The String class has the following operators.

MEMBER	DESCRIPTION
String::operator== Operator	Indicates whether two specified String objects have the same value.
operator+ Operator	Concatenates two String objects into a new String object.
String::operator> Operator	Indicates whether the value of one String object is greater than the value of a second String object.
String::operator>= Operator	Indicates whether the value of one String object is greater than or equal to the value of a second String object.
String::operator!= Operator	Indicates whether two specified String objects have different values.
String::operator< Operator	Indicates whether the value of one String object is less than the value of a second String object.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Header vccorlib.h (included by default)

String::Begin Method

Returns a pointer to the beginning of the current string.

Syntax

```
char16* Begin();
```

Return Value

A pointer to the beginning of the current string.

String::CompareOrdinal Method

Static method that compares two `String` objects by evaluating the numeric values of the corresponding characters in the two string values represented by the objects.

Syntax

```
static int CompareOrdinal( String^ str1, String^ str2 );
```

Parameters

str1
The first String object.

str2
The second String object.

Return Value

An integer that indicates the lexical relationship between the two comparands. The following table lists the possible return values.

VALUE	CONDITION
-1	`str1` is less than `str2`.
0	`str1` is equals `str2`.
1	`str1` is greater than `str2`.

String::Concat Method

Concatenates the values of two String objects.

Syntax

```
String^ Concat( String^ str1, String^ str2);
```

Parameters

str1
The first String object, or `null`.

str2
The second String object, or `null`.

Return Value

A new String^ object whose value is the concatenation of the values of `str1` and `str2`.

If `str1` is `null` and `str2` is not, `str1` is returned. If `str2` is `null` and `str1` is not, `str2` is returned. If `str1` and `str2` are both `null`, the empty string (L"") is returned.

String::Data Method

Returns a pointer to the beginning of the object's data buffer as a C-style array of `char16` (`wchar_t`) elements.

Syntax

```
const char16* Data();
```

Return Value

A pointer to the beginning of a `const char16` array of Unicode characters (`char16` is a typedef for `wchar_t`).

Remarks

Use this method to convert from `Platform::String^` to `wchar_t*`. When the `String` object goes out of scope, the Data pointer is no longer guaranteed to be valid. To store the data beyond the lifetime of the original `String` object, use wcscpy_s to copy the array into memory that you have allocated yourself.

String::Dispose Method

Frees or releases resources.

Syntax

```
virtual override void Dispose();
```

String::End Method

Returns a pointer past the end of the current string.

Syntax

```
char16* End();
```

Return Value

A pointer to past the end of the current string.

Remarks

End() returns Begin() + Length.

String::Equals Method

Indicates whether the specified String has the same value as the current object.

Syntax

```
bool String::Equals(Object^ str);
bool String::Equals(String^ str);
```

Parameters

str
The object to compare.

Return Value

`true` if `str` is equal to the current object; otherwise, `false`.

Remarks

This method is equivalent to the static String::CompareOrdinal. In the first overload, it is expected the `str` parameter can be cast to a String^ object.

String::GetHashCode Method

Returns the hash code for this instance.

Syntax

```
virtual override int GetHashCode();
```

Return Value

The hash code for this instance.

String::IsEmpty Method

Indicates whether the current String object is empty.

Syntax

```
bool IsEmpty();
```

Return Value

`true` if the current `String` object is **null** or the empty string (L""); otherwise, `false`.

String::IsFastPass Method

Indicates whether the current String object is participating in a *fast pass* operation. In a fast pass operation, reference counting is suspended.

Syntax

```
bool IsFastPass();
```

Return Value

`true` if the current `String` object is fast-past; otherwise, `false`.

Remarks

In a call to a function where a reference-counted object is a parameter, and the called function only reads that object, the compiler can safely suspend reference counting and improve calling performance. There is nothing useful that your code can do with this property. The system handles all the details.

String::Length Method

Retrieves the number of characters in the current `String` object.

Syntax

```
unsigned int Length();
```

Return Value

The number of characters in the current `String` object.

Remarks

The length of a String with no characters is zero. The length of the following string is 5:

```
String^ str = "Hello";
int len = str->Length(); //len = 5
```

The character array returned by the String::Data has one additional character, which is the terminating NULL or '\0'. This character is also two bytes long.

String::operator+ Operator

Concatenates two String objects into a new String object.

Syntax

```
bool String::operator+( String^ str1, String^ str2);
```

Parameters

str1
The first `String` object.

str2
The second `String` object, whose contents will be appended to `str1`.

Return Value

`true` if *str1* is equal to *str2*; otherwise, `false`.

Remarks

This operator creates a `String^` object that contains the data from the two operands. Use it for convenience when extreme performance is not critical. A few calls to "`+`" in a function will probably not be noticeable, but if you are manipulating large objects or text data in a tight loop, then use the standard C++ mechanisms and types.

String::operator== Operator

Indicates whether two specified String objects have the same text value.

Syntax

```
bool String::operator==( String^ str1, String^ str2);
```

Parameters

str1
The first `String` object to compare.

str2
The second `String` object to compare.

Return Value

`true` if the contents of `str1` are equal to `str2`; otherwise, `false`.

Remarks

This operator is equivalent to String::CompareOrdinal.

`String::operator>`

Indicates whether the value of one `String` object is greater than the value of a second `String` object.

Syntax

```
bool String::operator>( String^ str1, String^ str2);
```

Parameters

str1
The first `String` object.

str2
The second `String` object.

Return Value

`true` if the value of `str1` is greater than the value of `str2`; otherwise, `false`.

Remarks

This operator is equivalent to explicitly calling String::CompareOrdinal and getting a result greater than zero.

`String::operator>=`

Indicates whether the value of one `String` object is greater than or equal to the value of a second `String` object.

Syntax

```
bool String::operator>=( String^ str1, String^ str2);
```

Parameters

str1
The first `String` object.

str2
The second `String` object.

Return Value

`true` if the value of `str1` is greater than or equal to the value of `str2`; otherwise, `false`.

String::operator!=

Indicates whether two specified `String` objects have different values.

Syntax

```
bool String::operator!=( String^ str1, String^ str2);
```

Parameters

str1
The first `String` object to compare.

str2
The second `String` object to compare.

Return Value

`true` if `str1` is not equal to `str2`; otherwise, `false`.

`String::operator<`

Indicates whether the value of one `String` object is less than the value of a second `String` object.

Syntax

```
bool String::operator<( String^ str1, String^ str2);
```

Parameters

str1
The first `String` object.

str2
The second `String` object.

Return Value

`true` if the value of *str1* is less than the value of *str2*; otherwise, `false`.

String::String Constructor

Initializes a new instance of the `String` class with a copy of the input string data.

Syntax

```
String();
String(char16* s);
String(char16* s, unsigned int n);
```

Parameters

s
A series of wide characters that initialize the string. char16

n
A number that specifies the length of the string.

Remarks

If performance is critical and you control the lifetime of the source string, you can use Platform::StringReference in place of String.

Example

```
String^ s = L"Hello!";
```

String::ToString

Returns a `String` object whose value is the same as the current string.

Syntax

```
String^ String::ToString();
```

Return Value

A `String` object whose value is the same as the current string.

See also

Platform namespace

Platform::StringReference Class

9/21/2022 • 2 minutes to read • Edit Online

An optimization type that you can use to pass string data from `Platform::String^` input parameters to other methods with a minimum of copy operations.

Syntax

```
class StringReference
```

Remarks

Members

Public Constructors

NAME	DESCRIPTION
StringReference::StringReference	Two constructors for creating instances of `StringReference`.

Public Methods

NAME	DESCRIPTION
StringReference::Data	Returns the string data as an array of char16 values.
StringReference::Length	Returns the number of characters in the string.
StringReference::GetHSTRING	Returns the string data as an HSTRING.
StringReference::GetString	Returns the string data as a `Platform::String^`.

Public Operators

NAME	DESCRIPTION
StringReference::operator=	Assigns a `StringReference` to a new `StringReference` instance.
StringReference::operator()	Converts a `StringReference` to a `Platform::String^`.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Header: vccorlib.h

StringReference::Data Method

Returns the contents of this `StringReference` as an array of char16 values.

Syntax

```
const ::default::char16 * Data() const;
```

Return Value

An array of char16 UNICODE text characters.

StringReference::GetHSTRING Method

Returns the contents of the string as an `__abi_HSTRING`.

Syntax

```
__abi_HSTRING GetHSTRING() const;
```

Return Value

An `__abi_HSTRING` that contains the string data.

Remarks

StringReference::GetString Method

Returns the contents of the string as a `Platform::String^`.

Syntax

```
__declspec(no_release_return) __declspec(no_refcount)
    ::Platform::String^ GetString() const;
```

Return Value

A `Platform::String^` that contains the string data.

StringReference::Length Method

Returns the number of characters in the string.

Syntax

```
unsigned int Length() const;
```

Return Value

An unsigned integer that specifies the number of characters in the string.

Remarks

StringReference::operator= Operator

Assigns the specified object to the current `StringReference` object.

Syntax

```
StringReference& operator=(const StringReference& __fstrArg);
StringReference& operator=(const ::default::char16* __strArg);
```

Parameters

__fstrArg
The address of a `StringReference` object that is used to initialize the current `StringReference` object.

__strArg
Pointer to an array of char16 values that is used to initialize the current `StringReference` object.

Return Value

A reference to an object of type `StringReference`.

Remarks

Because `StringReference` is a standard C++ class and not a ref class, it does not appear in the **Object Browser**.

StringReference::operator() Operator

Converts a `StringReference` object to a `Platform::String^` object.

Syntax

```
__declspec(no_release_return) __declspec(no_refcount)
        operator ::Platform::String^() const;
```

Return Value

A handle to an object of type `Platform::String`.

StringReference::StringReference Constructor

Initializes a new instance of the `StringReference` class.

Syntax

```
StringReference();
StringReference(const StringReference& __fstrArg);
StringReference(const ::default::char16* __strArg);
StringReference(const ::default::char16* __strArg, size_t __lenArg);
```

Parameters

__fstrArg
The `StringReference` whose data is used to initialize the new instance.

__strArg
Pointer to an array of char16 values that is used to initialize the new instance.

__lenArg
The number of elements in `__strArg`.

Remarks

The first version of this constructor is the default constructor. The second version initializes a new `StringReference` instance class from the object that's specified by the `__fstrArg` parameter. The third and fourth overloads initialize a new `StringReference` instance from an array of char16 values. char16 represents a 16-bit UNICODE text character.

See also

Platform::StringReference Class

Platform::Type Class

9/21/2022 • 2 minutes to read • Edit Online

Contains run-time information about a type—specifically, a string name and a typecode. Obtained by calling Object::GetType on any object or using the typeid operator on a class or struct name.

Syntax

```
public ref class Platform::Type :
    Platform::Object, Platform::Details::IEquatable,
    Platform::Details::IPrintable
```

Remarks

The `Type` class is useful in applications that must direct processing by using an `if` or `switch` statement that branches based on the run-time type of an object. The type code that describes the category of a type is retrieved by using the Type::GetTypeCode member function.

Public methods

NAME	DESCRIPTION
Type::GetTypeCode Method	Returns a Platform::TypeCode Enumeration value for the object.
Type::ToString Method	Returns the name of the type as specified in its metadata.

Public properties

NAME	DESCRIPTION
Type::FullName	Returns a Platform::String Class^ that represents the fully qualified name of the type, and uses . (dot) as a separator, not :: (double colon)—for example, `MyNamespace.MyClass`.

Conversion operators

NAME	DESCRIPTION
operator Type^	Enables conversion from `Windows::UI::Xaml::Interop::TypeName` to `Platform::Type`.
operator Windows::UI::Xaml::Interop::TypeName	Enables conversion from `Platform::Type` to `Windows::UI::Xaml::Interop::TypeName`.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

Type::FullName Property

Retrieves the fully-qualified name of the current type in the form `Namespace.Type`.

Syntax

```
String^ FullName();
```

Return Value

The name of the type.

Example

```
//  namespace is TestApp
MainPage::MainPage()
{
    InitializeComponent();
    Type^ t = this->GetType();
    auto s = t->FullName; // returns "TestApp.MainPage"
    auto s2 = t->ToString(); //also returns "TestApp.MainPage"
}
```

Type::GetTypeCode Method

Retrieves a built-in types numerical type category.

Syntax

```
Platform::TypeCode GetTypeCode();
```

Return Value

One of the Platform::TypeCode enumerated values.

Remarks

The equivalent of the GetTypeCode() member method is the `typeid` property.

Type::ToString Method

Retrieves a the name of the type.

Syntax

```
Platform::String^ ToString();
```

Return Value

A name of the type as specified in its metadata.

See also

Platform namespace

operator Type^

9/21/2022 • 2 minutes to read • Edit Online

Enables conversion from Windows::UI::Xaml::Interop::TypeName to `Platform::Type`.

Syntax

```
Operator Type^(Windows::UI::Xaml::Interop::TypeName typeName);
```

Return Value

Returns a `Platform::Type` when given a Windows::UI::Xaml::Interop::TypeName.

Remarks

`TypeName` is the language-neutral Windows Runtime struct for representing type information. Platform::Type is specific to C++ and can't be passed across the application binary interface (ABI). Here's one use of `TypeName`, in the Navigate function:

```
rootFrame->Navigate(TypeName(MainPage::typeid), e->Arguments);
```

Example

The next example shows how to convert between `TypeName` and `Type`.

```
// Convert from Type to TypeName
TypeName tn = TypeName(MainPage::typeid);

// Convert back from TypeName to Type
Type^ tx2 = (Type^)(tn);
```

.NET Framework Equivalent

.NET Framework programs project `TypeName` as Type

Requirements

See also

operator Windows::UI::Xaml::Interop::TypeName
Platform::Type Class

Platform::TypeCode Enumeration

9/21/2022 • 2 minutes to read • Edit Online

Specifies a numeric category that represents a built-in type.

Syntax

```
enum class TypeCode {};
```

Members

TYPE CODE	DESCRIPTION
Boolean	A Platform::Boolean type.
Char16	A default::char16 type.
DateTime	A DateTime type.
Decimal	A numeric type.
Double	A default::float64 type.
Empty	Void
Int16	A default::int16 type.
Int32	A default::int32 type.
Int64	A default::int64 type.
Int8	A default::int8 type.
Object	A Platform::Object type.
Single	A default::float32 type.
String	A Platform::String type.
UInt16	A default::uint16 type.
UInt32	A default::uint32 type.
UInt64	A default::uint64 type.
UInt8	A default::uint8 type.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

Platform::UIntPtr value class

9/21/2022 • 2 minutes to read • Edit Online

Represents an unsigned pointer whose size is appropriate for the platform (32-bit or 64-bit).

Syntax

```
public value struct UintPtr
```

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform namespace

Platform::ValueType Class

9/21/2022 • 2 minutes to read • Edit Online

The base class for instances of value types.

Syntax

```
public ref class ValueType : Object
```

Public methods

NAME	DESCRIPTION
ValueType::ToString	Returns a string representation of the object. Inherited from Platform::Object.

Remarks

The ValueType class is used to construct value types. ValueType is derived from Object, which has basic members. However, the compiler detaches those basic members from value types that are derived from the ValueType class. The compiler reattaches those basic members when a value type is boxed.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

ValueType::ToString Method

Returns a string representation of the object.

Syntax

```
Platform::String ToString();
```

Return Value

A Platform::String that represents the value.

See also

Platform namespace

Platform::WeakReference Class

9/21/2022 • 2 minutes to read • Edit Online

Represents a weak reference to an instance of a ref class.

Syntax

```
class WeakReference
```

Parameters

Members

Constructors

MEMBER	DESCRIPTION
WeakReference::WeakReference	Initializes a new instance of the WeakReference class.

Methods

MEMBER	DESCRIPTION
WeakReference::Resolve	Returns a handle to the underlying ref class, or nullptr if the object no longer exists.

Operators

MEMBER	DESCRIPTION
WeakReference::operator=	Assigns a new value to the WeakReference object.
WeakReference::operator BoolType	Implements the safe bool pattern.

Remarks

The WeakReference class itself is not a ref class and therefore it does not inherit from Platform::Object^ and cannot be used in the signature of a public method.

WeakReference::operator=

Assigns a value to a WeakReference.

Syntax

```
WeakReference& operator=(decltype(__nullptr));
WeakReference& operator=(const WeakReference& otherArg);
WeakReference& operator=(WeakReference&& otherArg);
WeakReference& operator=(const volatile ::Platform::Object^ const otherArg);
```

Remarks

The last overload in the list above enables you to assign a ref class to a WeakReference variable. In this case the

ref class is downcast to Platform::Object^. You restore the original type later by specifying it as the argument for the type parameter in the WeakReference::Resolve<T> member function.

WeakReference::operator BoolType

Implements the safe bool pattern for the WeakReference class. Not to be called explicitly from your code.

Syntax

```
BoolType BoolType();
```

WeakReference::Resolve Method (Platform namespace)

Returns a handle to the original ref class, or `nullptr` if the object no longer exists.

Syntax

```
template<typename T>
T^ Resolve() const;
```

Parameters

Property Value/Return Value

A handle to the ref class that the WeakReference object was previously associated with, or nullptr.

Example

```
Bar^ bar = ref new Bar();
//use bar...

if (bar != nullptr)
{
    WeakReference wr(bar);
    Bar^ newReference = wr.Resolve<Bar>();
}
```

Note that the type parameter is T, not T^.

WeakReference::WeakReference Constructor

Provides various ways to construct a WeakReference.

Syntax

```
WeakReference();
WeakReference(decltype(__nullptr));
WeakReference(const WeakReference& otherArg);
WeakReference(WeakReference&& otherArg);
explicit WeakReference(const volatile ::Platform::Object^ const otherArg);
```

Example

```
MyClass^ mc = ref new MyClass();
WeakReference wr(mc);
MyClass^ copy2 = wr.Resolve<MyClass>();
```

See also

Platform namespace

Platform::WriteOnlyArray Class

9/21/2022 • 2 minutes to read • Edit Online

Represents a one-dimensional array that's used as an input parameter when the caller passes an array for the method to fill.

This ref class is declared as private in vccorlib.h; therefore, it's not emitted in metadata and is only consumable from C++. This class is intended only for use as an input parameter that receives an array that the caller has allocated. It is not constructible from user code. It enables a C++ method to write directly into that array—a pattern that's known as the *FillArray* pattern. For more information, see Array and WriteOnlyArray.

Syntax

```
private ref class WriteOnlyArray<T, 1>
```

Members

Public Methods

These methods have internal accessibility—that is, they are only accessible within the C++ app or component.

NAME	DESCRIPTION
WriteOnlyArray::begin	An iterator that points to the first element of the array.
WriteOnlyArray::Data	A pointer to the data buffer.
WriteOnlyArray::end	An iterator that points to one past the last element in the array.
WriteOnlyArray::FastPass	Indicates whether the array can use the FastPass mechanism, which is an optimization transparently performed by the system. Don't use this in your code
WriteOnlyArray::Length	Returns the number of elements in the array.
WriteOnlyArray::set	Sets the specified element to the specified value.

Inheritance Hierarchy

`WriteOnlyArray`

Requirements

Compiler option: **/ZW**

Metadata: Platform.winmd

Namespace: Platform

WriteOnlyArray::begin Method

Returns a pointer to the first element in the array.

Syntax

```
T* begin() const;
```

Return Value

A pointer to the first element in the array.

Remarks

This iterator can be used with STL algorithms such as `std::sort` to operate on elements in the array.

WriteOnlyArray::Data Property

Pointer to the data buffer.

Syntax

```
property T* Data{
   T* get() const;
}
```

Return Value

A pointer to the raw array bytes.

WriteOnlyArray::end Method

Returns a pointer to one past the last element in the array.

Syntax

```
T* end() const;
```

Return Value

A pointer iterator to one past the last element in the array.

Remarks

This iterator can be used with STL algorithms to perform operations such as `std::sort` on the array elements.

WriteOnlyArray::FastPass Property

Indicates whether the internal FastPass optimization can be performed. Not intended for use by user code.

Syntax

```
property bool FastPass{
   bool get() const;
}
```

Return Value

Boolean value that indicates whether the array is FastPass.

WriteOnlyArray::get Method

Returns the element at the specified index.

Syntax

```
T& get(unsigned int indexArg) const;
```

Parameters

indexArg
The index to use.

Return Value

WriteOnlyArray::Length Property

Returns the number of elements in the caller-allocated array.

Syntax

```
property unsigned int Length{
   unsigned int get() const;
}
```

Return Value

The number of elements in the array.

WriteOnlyArray::set Function

Sets the specified value at the specified index in the array.

Syntax

```
T& set(
   unsigned int indexArg,
   T valueArg);
```

Parameters

indexArg
The index of the element to set.

valueArg
The value to set at `indexArg`.

Return Value

A reference to the element that was just set.

Remarks

For more information about how to interpret the HRESULT value, see Structure of COM Error Codes.

See also

Platform Namespace
Creating Windows Runtime Components in C++

Platform::WrongThreadException Class

9/21/2022 • 2 minutes to read • Edit Online

Thrown when a thread calls by way of an interface pointer for a proxy object that doesn't belong to the thread's apartment.

Syntax

```
public ref class WrongThreadException : COMException,    IException,    IPrintable,    IEquatable
```

Remarks

For more information, see the COMException.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform

Metadata: platform.winmd

See also

Platform::COMException Class

Platform::Collections Namespace

9/21/2022 • 2 minutes to read • Edit Online

The Platform::Collections namespace contains the `Map`, `MapView`, `Vector`, and `VectorView` classes. These classes are concrete implementations of the corresponding interfaces that are defined in the Windows::Foundation::Collections namespace. The concrete collection types are not portable across the ABI (for example when a Javascript or C# program calls into a C++ component), but they are implicitly convertible to their corresponding interface types. For example, if you implement a public method that populates and returns a collection, then use Platform::Collections::Vector to implement the collection internally and use Windows::Foundation::Collections::IVector as the return type. For more information, see Collections and Creating Windows Runtime Components in C++.

You can construct a Platform::Collections::Vector from a std::vector and a Platform::Collections::Map from a std::map.

In addition, the Platform::Collections namespace provides support for back insert and input iterators, and `Vector` and `VectorView` iterators.

You must include (`#include`) the collection.h header to use the types in the Platform::Collections namespace.

Syntax

```
#include <collection.h>
using namespace Platform::Collections;
```

Members

This namespace contains the following members.

NAME	DESCRIPTION
Platform::Collections::BackInsertIterator Class	Represents an iterator that inserts an element at the end of a collection.
Platform::Collections::InputIterator Class	Represents an iterator that inserts an element at the beginning of a collection.
Platform::Collections::Map Class	Represents a modifiable collection of key-value pairs that are accessed by a key. Similar to std::map.
Platform::Collections::MapView Class	Represents a read-only collection of key-value pairs that are accessed by a key.
Platform::Collections::Vector Class	Represents a modifiable sequence of elements. Similar to std::vector.
Platform::Collections::VectorIterator Class	Represents an iterator that traverses a `Vector` collection.
Platform::Collections::VectorView Class	Represents a read-only sequence of elements.

NAME	DESCRIPTION
Platform::Collections::VectorViewIterator Class	Represents an iterator that traverses a `VectorView` collection.

Inheritance hierarchy

Platform namespace

Requirements

Metadata: platform.winmd

Namespace: Platform::Collections

Compiler option: /ZW

See also

Platform Namespace

Platform::Collections::BackInsertIterator Class

9/21/2022 • 2 minutes to read • Edit Online

Represents an iterator that inserts, rather than overwrites, elements into the back end of a sequential collection.

Syntax

```
template <typename T>
class BackInsertIterator :
public ::std::iterator<::std::output_iterator_tag, void, void, void, void>;
```

Parameters

T

The type of item in the current collection.

Remarks

The BackInsertIterator class implements the rules required by the back_insert_iterator Class.

Members

Public Constructors

NAME	DESCRIPTION
BackInsertIterator::BackInsertIterator	Initializes a new instance of the BackInsertIterator class.

Public Operators

NAME	DESCRIPTION
BackInsertIterator::operator* Operator	Retrieves a reference to the current BackInsertIterator.
BackInsertIterator::operator++ Operator	Returns a reference to the current BackInsertIterator. The iterator is unmodified.
BackInsertIterator::operator= Operator	Appends the specified object to the end of the current sequential collection.

Inheritance Hierarchy

`BackInsertIterator`

Requirements

Header: collection.h

Namespace: Platform::Collections

BackInsertIterator::BackInsertIterator Constructor

Initializes a new instance of the `BackInsertIterator` class.

Syntax

```
explicit BackInsertIterator(
   Windows::Foundation::Collections::IVector<T>^ v);
```

Parameters

v

An IVector<T> object.

Remarks

A `BackInsertIterator` inserts elements after the last element of the object specified by parameter `v`.

BackInsertIterator::operator= Operator

Appends the specified object to the end of the current sequential collection.

Syntax

```
BackInsertIterator& operator=( const T& t);
```

Parameters

t

The object to append to the current collection.

Return Value

A reference to the current BackInsertIterator.

BackInsertIterator::operator* Operator

Retrieves a reference to the current BackInsertIterator.

Syntax

```
BackInsertIterator& operator*();
```

Return Value

A reference to the current BackInsertIterator.

Remarks

This operator returns a reference to the current BackInsertIterator; not to any element in the current collection.

BackInsertIterator::operator++ Operator

Returns a reference to the current BackInsertIterator. The iterator is unmodified.

Syntax

```
BackInsertIterator& operator++();

BackInsertIterator operator++(int);
```

Return Value

A reference to the current BackInsertIterator.

Remarks

By design, the first syntax example pre-increments the current BackInsertIterator, and the second syntax post-

increments the current BackInsertIterator. The `int` type in the second syntax indicates a post-increment operation, not an actual integer operand.

However, this operator does not actually modify the BackInsertIterator. Instead, this operator returns a reference to the unmodified, current iterator. This is the same behavior as operator*.

See also

Platform Namespace

Platform::Collections::InputIterator Class

9/21/2022 • 2 minutes to read • Edit Online

Provides a Standard Template Library InputIterator for collections derived from the Windows Runtime.

Syntax

```
template <typename X>
class InputIterator;
```

Parameters

X

The typename of the InputIterator template class.

Members

Public Typedefs

NAME	DESCRIPTION
`difference_type`	A pointer difference (ptrdiff_t).
`iterator_category`	The category of a input iterator (::std::input_iterator_tag).
`pointer`	A pointer to a `const X`
`reference`	A reference to a `const X`
`value_type`	The `X` typename.

Public Constructors

NAME	DESCRIPTION
InputIterator::InputIterator	Initializes a new instance of the InputIterator class.

Public Operators

NAME	DESCRIPTION
InputIterator::operator!= Operator	Indicates whether the current InputIterator is not equal to a specified InputIterator.
InputIterator::operator* Operator	Retrieves a reference to the element specified by the current InputIterator.
InputIterator::operator++ Operator	Increments the current InputIterator.
InputIterator::operator== Operator	Indicates whether the current InputIterator is equal to a specified InputIterator.

NAME	DESCRIPTION
InputIterator::operator-> Operator	Retrieves the address of the element referenced by the current InputIterator.

Inheritance Hierarchy

`InputIterator`

Requirements

Header: collection.h

Namespace: Platform::Collections

InputIterator::InputIterator Constructor

Initializes a new instance of the InputIterator class.

Syntax

```
InputIterator();
explicit InputIterator(Windows::Foundation::Collections<X>^ iterator);
```

Parameters

iterator
An iterator object.

`InputIterator::operator->` Operator

Retrieves the address of the element specified by the current InputIterator.

Syntax

```
pointer operator->() const;
```

Return Value

The address of the element specified by the current InputIterator.

InputIterator::operator* Operator

Retrieves a reference to the element specified by the current InputIterator.

Syntax

```
reference operator*() const;
```

Return Value

The element specified by the current InputIterator.

InputIterator::operator== Operator

Indicates whether the current InputIterator is equal to a specified InputIterator.

Syntax

```
bool operator== (const InputIterator& other) const;
```

Parameters

other
Another InputIterator.

Return Value

`true` if the current InputIterator is equal to *other*, otherwise, `false`.

InputIterator::operator++ Operator

Increments the current InputIterator.

Syntax

```
InputIterator& operator++();
InputIterator operator++(int);
```

Return Value

The first syntax increments and then returns the current InputIterator. The second syntax returns a copy of the current InputIterator and then increments the current InputIterator.

Remarks

The first InputIterator syntax pre-increments the current InputIterator.

The second syntax post-increments the current InputIterator. The `int` type in the second syntax indicates a post-increment operation, not an actual integer operand.

InputIterator::operator!= Operator

Indicates whether the current InputIterator is not equal to a specified InputIterator.

Syntax

```
bool operator!=(const InputIterator& other) const;
```

Parameters

other
Another InputIterator.

Return Value

`true` if the current InputIterator is not equal to *other*, otherwise, `false`.

See also

Platform Namespace

Platform::Collections::Map Class

9/21/2022 • 4 minutes to read • Edit Online

Represents a *map*, which is a collection of key-value pairs. Implements Windows::Foundation::Collections::IObservableMap to help with XAML data binding.

Syntax

```
template <
   typename K,
   typename V,
   typename C = std::less<K>>
ref class Map sealed;
```

Parameters

K
The type of the key in the key-value pair.

V
The type of the value in the key-value pair.

C
A type that provides a function object that can compare two element values as sort keys to determine their relative order in the Map. By default, std::less<K>.

__is_valid_winrt_type() A compiler-generated function that validates the type of *K* and *V* and provides a friendly error message if the type cannot be stored in the Map.

Remarks

Allowed types are:

- integers
- interface class^
- public ref class^
- value struct
- public enum class

Map is basically a wrapper for std::map. It is a C++ concrete implementation of the Windows::Foundation::Collections::IMap<Windows::Foundation::Collections::IKeyValuePair<K,V>> and IObservableMap types that are passed across public Windows Runtime interfaces. If you try to use a `Platform::Collections::Map` type in a public return value or parameter, compiler error C3986 is raised. You can fix the error by changing the type of the parameter or return value to Windows::Foundation::Collections::IMap<K,V>.

For more information, see Collections.

Members

Public Constructors

NAME	DESCRIPTION
Map::Map	Initializes a new instance of the Map class.

Public Methods

NAME	DESCRIPTION
Map::Clear	Removes all key-value pairs from the current Map object.
Map::First	Returns an iterator that specifies the first element in the map.
Map::GetView	Returns a read-only view of the current Map; that is, a Platform::Collections::MapView Class.
Map::HasKey	Determines whether the current Map contains the specified key.
Map::Insert	Adds the specified key-value pair to the current Map object.
Map::Lookup	Retrieves the element at the specified key in the current Map object.
Map::Remove	Deletes the specified key-value pair from the current Map object.
Map::Size	Returns the number of elements in the current Map object.

Events

NAME	DESCRIPTION
Map::MapChanged event	Occurs when the Map changes.

Inheritance Hierarchy

`Map`

Requirements

Header: collection.h

Namespace: Platform::Collections

Map::Clear Method

Removes all key-value pairs from the current Map object.

Syntax

```
virtual void Clear();
```

Map::First Method

Returns an iterator that specifies the first element in the map, or `nullptr` if the map is empty.

Syntax

```
virtual Windows::Foundation::Collections::IIterator<
Windows::Foundation::Collections::IKeyValuePair<K, V>^>^ First();
```

Return Value

An iterator that specifies the first element in the map.

Remarks

A convenient way to hold the iterator returned by First() is to assign the return value to a variable that is declared with the `auto` type deduction keyword. For example, `auto x = myMap->First();`.

Map::GetView Method

Returns a read-only view of the current Map; that is, a Platform::Collections::MapView Class, which implements the Windows::Foundation::Collections::IMapView<K,V> interface.

Syntax

```
Windows::Foundation::Collections::IMapView<K, V>^ GetView();
```

Return Value

A `MapView` object.

Map::HasKey Method

Determines whether the current Map contains the specified key.

Syntax

```
bool HasKey(K key);
```

Parameters

key
The key used to locate the Map element. The type of *key* is typename *K*.

Return Value

`true` if the key is found; otherwise, `false`.

Map::Insert Method

Adds the specified key-value pair to the current Map object.

Syntax

```
virtual bool Insert(K key, V value);
```

Parameters

key
The key portion of the key-value pair. The type of *key* is typename *K*.

value
The value portion of the key-value pair. The type of *value* is typename *V*.

Return Value

`true` if the key of an existing element in the current Map matches *key* and the value portion of that element is set to *value*. `false` if no existing element in the current Map matches *key* and the *key* and *value* parameters are made into a key-value pair and then added to the current Map.

Map::Lookup Method

Retrieves the value of type V that is associated with the specified key of type K, if the key exists.

Syntax

```
V Lookup(K key);
```

Parameters

key
The key used to locate an element in the Map. The type of *key* is typename *K*.

Return Value

The value that is paired with the *key*. The type of the return value is typename *V*.

Remarks

If the key does not exist, then a Platform::OutOfBoundsException is thrown.

Map::Map Constructor

Initializes a new instance of the Map class.

Syntax

```
explicit Map(const C& comp = C());
explicit Map(const StdMap& m);
explicit Map(StdMap&& m ;
template <typename InIt>
Map(
   InItfirst,
   InItlast,
   const C& comp = C());
```

Parameters

InIt
The typename of the current Map.

comp
A type that provides a function object that can compare two element values as sort keys to determine their relative order in the Map.

m
A reference or rvalue to a `map Class` that is used to initialize the current Map.

first
The input iterator of the first element in a range of elements used to initialize the current Map.

last

The input iterator of the first element after a range of elements used to initialize the current Map.

Map::MapChanged Event

Raised when an item is inserted into or removed from the map.

Syntax

```
event Windows::Foundation::Collections::MapChangedEventHandler<K,V>^ MapChanged;
```

Property Value/Return Value

A MapChangedEventHandler<K,V> that contains information about the object that raised the event, and the kind of change that occurred. See also IMapChangedEventArgs<K> and CollectionChange Enumeration.

.NET Framework Equivalent

Windows Runtime apps that use C# or Visual Basic project IMap<K,V> as IDictionary<K,V>.

Map::Remove Method

Deletes the specified key-value pair from the current Map object.

Syntax

```
virtual void Remove(K key);
```

Parameters

key
The key portion of the key-value pair. The type of *key* is typename *K*.

Map::Size Method

Returns the number of Windows::Foundation::Collections::IKeyValuePair<K,V> elements in the Map.

Syntax

```
virtual property unsigned int Size;
```

Return Value

The number of elements in the Map.

See also

Collections (C++/CX)
Platform Namespace
Creating Windows Runtime Components in C++

Platform::Collections::MapView Class

9/21/2022 • 2 minutes to read • Edit Online

Represents a read-only view into a *map*, which is a collection of key-value pairs.

Syntax

```
template <
   typename K,
   typename V,
   typename C = ::std::less<K>>
ref class MapView sealed;
```

Parameters

K

The type of the key in the key-value pair.

V

The type of the value in the key-value pair.

C

A type that provides a function object that can compare two element values as sort keys to determine their relative order in the MapView. By default, std::less<K>.

Remarks

MapView is a concrete C++ implementation of the Windows::Foundation::Collections::IMapView <K,V> interface that is passed across the application binary interface (ABI). For more information, see Collections (C++/CX).

Members

Public Constructors

NAME	DESCRIPTION
MapView::MapView	Initializes a new instance of the MapView class.

Public Methods

NAME	DESCRIPTION
MapView::First	Returns an iterator that is initialized to the first element in the map view.
MapView::HasKey	Determines whether the current MapView contains the specified key.
MapView::Lookup	Retrieves the element at the specified key in the current MapView object.
MapView::Size	Returns the number of elements in the current MapView object.

NAME	DESCRIPTION
MapView::Split	Splits an original MapView object into two MapView objects.

Inheritance Hierarchy

`MapView`

Requirements

Header: collection.h

Namespace: Platform::Collections

MapView::First Method

Returns an iterator that specifies the first element in the map view.

Syntax

```
virtual Windows::Foundation::Collections::IIterator<
   Windows::Foundation::Collections::IKeyValuePair<K, V>^>^ First();
```

Return Value

An iterator that specifies the first element in the map view.

Remarks

A convenient way to hold the iterator returned by First() is to assign the return value to a variable that is declared with the `auto` type deduction keyword. For example, `auto x = myMapView->First();`.

MapView::HasKey Method

Determines whether the current MapView contains the specified key.

Syntax

```
bool HasKey(K key);
```

Parameters

key
The key used to locate the MapView element. The type of *key* is typename *K*.

Return Value

`true` if the key is found; otherwise, `false`.

MapView::Lookup Method

Retrieves the value of type V that is associated with the specified key of type K.

Syntax

```
V Lookup(K key);
```

Parameters

key
The key used to locate an element in the MapView. The type of `key` is typename *K*.

Return Value

The value that is paired with the `key`. The type of the return value is typename *V*.

MapView::MapView Constructor

Initializes a new instance of the MapView class.

Syntax

```
explicit MapView(const C& comp = C());

explicit MapView(const ::std::map<K, V, C>& m);

explicit MapView(std::map<K, V, C>&& m);

template <typename InIt> MapView(
    InIt first,
    InIt last,
    const C& comp = C());

MapView(
    ::std::initializer_list<std::pair<const K, V>> il,
    const C& comp = C());
```

Parameters

InIt
The typename of the current MapView.

comp
A function object that can compare two element values as sort keys to determine their relative order in the MapView.

m
A reference or Lvalues and Rvalues to a `map Class` that is used to initialize the current MapView.

first
The input iterator of the first element in a range of elements used to initialize the current MapView.

last
The input iterator of the first element after a range of elements used to initialize the current MapView.

il
A std::initializer_list<std::pair<K,V>> whose elements will be inserted into the MapView.

MapView::Size Method

Returns the number of elements in the current MapView object.

Syntax

```
virtual property unsigned int Size;
```

Return Value

The number of elements in the current MapView.

MapView::Split Method

Divides the current MapView object into two MapView objects. This method is non-operational.

Syntax

```
void Split(
   Windows::Foundation::Collections::IMapView<
                        K, V>^ * firstPartition,
   Windows::Foundation::Collections::IMapView<
                        K, V>^ * secondPartition);
```

Parameters

firstPartition

The first part of the original MapView object.

secondPartition

The second part of the original MapView object.

Remarks

This method is not operational; it does nothing.

See also

Platform Namespace

Platform::Collections::UnorderedMap Class

9/21/2022 • 5 minutes to read • Edit Online

Represents an unordered *map*, which is a collection of key-value pairs.

Syntax

```
template <
   typename K,
   typename V,
   typename C = std::equal_to<K>
>
ref class Map sealed;
```

Parameters

K

The type of the key in the key-value pair.

V

The type of the value in the key-value pair.

C

A type that provides a function object that can compare two element values as sort keys to determine their relative order in the Map. By default, std::equal_to<K>.

Remarks

Allowed types are:

- integers
- interface class^
- public ref class^
- value struct
- public enum class

UnorderedMap is basically a wrapper for std::unordered_map that supports storage of Windows Runtime types. It is the a concrete implementation of the Windows::Foundation::Collections::IMap and IObservableMap types that are passed across public Windows Runtime interfaces. If you try to use a `Platform::Collections::UnorderedMap` type in a public return value or parameter, compiler error C3986 is raised. You can fix the error by changing the type of the parameter or return value to Windows::Foundation::Collections::IMap.

For more information, see Collections.

Members

Public Constructors

NAME	DESCRIPTION
UnorderedMap::UnorderedMap	Initializes a new instance of the Map class.

Public Methods

NAME	DESCRIPTION
UnorderedMap::Clear	Removes all key-value pairs from the current Map object.
UnorderedMap::First	Returns an iterator that specifies the first element in the map.
UnorderedMap::GetView	Returns a read-only view of the current Map; that is, a Platform::Collections::UnorderedMapView Class.
UnorderedMap::HasKey	Determines whether the current Map contains the specified key.
UnorderedMap::Insert	Adds the specified key-value pair to the current Map object.
UnorderedMap::Lookup	Retrieves the element at the specified key in the current Map object.
UnorderedMap::Remove	Deletes the specified key-value pair from the current Map object.
UnorderedMap::Size	Returns the number of elements in the current Map object.

Events

NAME	DESCRIPTION
Map::MapChanged event	Occurs when the Map changes.

Inheritance Hierarchy

`UnorderedMap`

Requirements

Header: collection.h

Namespace: Platform::Collections

UnorderedMap::Clear Method

Removes all key-value pairs from the current UnorderedMap object.

Syntax

```
virtual void Clear();
```

UnorderedMap::First Method

Returns an iterator that specifies the first Windows::Foundation::Collections::IKeyValuePair<K,V> element in the unordered map.

Syntax

```
virtual Windows::Foundation::Collections::IIterator<
   Windows::Foundation::Collections::IKeyValuePair<K, V>^>^
   First();
```

Return Value

An iterator that specifies the first element in the map.

Remarks

A convenient way to hold the iterator returned by First() is to assign the return value to a variable that is declared with the `auto` type deduction keyword. For example, `auto x = myUnorderedMap->First();`.

UnorderedMap::GetView Method

Returns a read-only view of the current UnorderedMap; that is, an Platform::Collections::UnorderedMapView Class that implements the Windows::Foundation::Collections::IMapView::IMapView interface.

Syntax

```
Windows::Foundation::Collections::IMapView<K, V>^ GetView();
```

Return Value

An `UnorderedMapView` object.

UnorderedMap::HasKey Method

Determines whether the current UnorderedMap contains the specified key.

Syntax

```
bool HasKey(
   K key
);
```

Parameters

key
The key used to locate the UnorderedMap element. The type of *key* is typename *K*.

Return Value

`true` if the key is found; otherwise, `false`.

UnorderedMap::Insert Method

Adds the specified key-value pair to the current UnorderedMap object.

Syntax

```
virtual bool Insert(
   K key,
   V value
);
```

Parameters

key

The key portion of the key-value pair. The type of *key* is typename *K*.

value
The value portion of the key-value pair. The type of *value* is typename *V*.

Return Value

`true` if the key of an existing element in the current Map matches *key* and the value portion of that element is set to *value*. `false` if no existing element in the current Map matches *key* and the *key* and *value* parameters are made into a key-value pair and then added to the current UnorderedMap.

UnorderedMap::Lookup Method

Retrieves the value of type V that is associated with the specified key of type K.

Syntax

```
V Lookup(
   K key
);
```

Parameters

key
The key used to locate an element in the UnorderedMap. The type of *key* is typename *K*.

Return Value

The value that is paired with the *key*. The type of the return value is typename *V*.

UnorderedMap::MapChanged

Raised when an item is inserted into or removed from the map.

Syntax

```
event Windows::Foundation::Collections::MapChangedEventHandler<K,V>^ MapChanged;
```

Property Value/Return Value

A MapChangedEventHandler<K,V> that contains information about the object that raised the event, and the kind of change that occurred. See also IMapChangedEventArgs<K> and CollectionChange Enumeration.

.NET Framework Equivalent

Windows Runtime apps that us C# or Visual Basic project IMap<K,V> as IDictionary<K,V>.

UnorderedMap::Remove Method

Deletes the specified key-value pair from the UnorderedMap object.

Syntax

```
virtual void Remove(
   K key);
```

Parameters

key

The key portion of the key-value pair. The type of *key* is typename *K*.

UnorderedMap::Size Method

Returns the number of Windows::Foundation::Collections::IKeyValuePair<K,V> elements in the UnorderedMap.

Syntax

```
virtual property unsigned int Size;
```

Return Value

The number of elements in the Unordered Map.

UnorderedMap::UnorderedMap Constructor

Initializes a new instance of the UnorderedMap class.

Syntax

```
UnorderedMap();

explicit UnorderedMap(
    size_t n
    );

UnorderedMap(
    size_t n,
    const H& h
    );

UnorderedMap(
    size_t n,
    const H& h,
    const P& p
    );

explicit UnorderedMap(
    const std::unordered_map<K, V, H, P>& m
    );

explicit UnorderedMap(
    std::unordered_map<K, V, H, P>&& m
    );

template <typename InIt>
UnorderedMap(
    InIt first,
    InIt last
    );

template <typename InIt>
UnorderedMap(
    InIt first,
    InIt last,
    size_t n
    );

template <typename InIt>
UnorderedMap(
    InIt first,
    InIt last,
    size_t n,
    const H& h
    );
```

```
template <typename InIt>
UnorderedMap(
    InIt first,
    InIt last,
    size_t n,
    const H& h,
    const P& p
    );

UnorderedMap(
    std::initializer_list< std::pair<const K, V>> il
    );

UnorderedMap(
    std::initializer_list< std::pair<const K, V>> il,
    size_t n
    );

UnorderedMap(
    std::initializer_list< std::pair<const K, V>> il,
    size_t n,
    const H& h
    );

UnorderedMap(
    std::initializer_list< std::pair<const K, V>> il,
    size_t n,
    const H& h,
    const P& p
    );
```

Parameters

InIt

The typename of the current UnorderedMap.

P

A function object that can compare two keys to determine whether they are equal. This parameter defaults to std::equal_to<K>.

H

A function object that produces a hash value for a keys. This parameter defaults to hash Class 1 for the key types that the class supports.

m

A reference or Lvalues and Rvalues to a std::unordered_map that is used to initialize the current UnorderedMap.

il

A std::initializer_list of std::pair objects that is used to initialize the map.

first

The input iterator of the first element in a range of elements used to initialize the current UnorderedMap.

last

The input iterator of the first element after a range of elements used to initialize the current UnorderedMap.

See also

Platform Namespace
Platform::Collections Namespace
Platform::Collections::Map Class

Platform::Collections::UnorderedMapView Class

Collections

Creating Windows Runtime Components in C++

Platform::Collections::UnorderedMapView Class

9/21/2022 • 3 minutes to read • Edit Online

Represents a read-only view into a *map*, which is a collection of key-value pairs.

Syntax

```
template <
   typename K,
   typename V,
   typename C = ::std::equal_to<K>>
ref class UnorderedMapView sealed;
```

Parameters

K

The type of the key in the key-value pair.

V

The type of the value in the key-value pair.

C

A type that provides a function object that can compare two key values for equality. By default, std::equal_to<K>

Remarks

UnorderedMapView is a concrete C++ implementation of the Windows::Foundation::Collections::IMapView<K,V> interface that is passed across the application binary interface (ABI). For more information, see Collections (C++/CX).

Members

Public Constructors

NAME	DESCRIPTION
UnorderedMapView::UnorderedMapView	Initializes a new instance of the UnorderedMapView class.

Public Methods

NAME	DESCRIPTION
UnorderedMapView::First	Returns an iterator that is initialized to the first element in the map view.
UnorderedMapView::HasKey	Determines whether the current UnorderedMapView contains the specified key.
UnorderedMapView::Lookup	Retrieves the element at the specified key in the current UnorderedMapView object.
UnorderedMapView::Size	Returns the number of elements in the current UnorderedMapView object.

NAME	DESCRIPTION
UnorderedMapView::Split	Splits an original UnorderedMapView object into two UnorderedMapView objects.

Inheritance Hierarchy

`UnorderedMapView`

Requirements

Header: collection.h

Namespace: Platform::Collections

UnorderedMapView::First Method

Returns an iterator that specifies the first Windows::Foundation::Collections::IKeyValuePair<K,V> element in the unordered map.

Syntax

```
virtual Windows::Foundation::Collections::IIterator<
    Windows::Foundation::Collections::IKeyValuePair<K, V>^>^
    First();
```

Return Value

An iterator that specifies the first element in the map view.

Remarks

A convenient way to hold the iterator returned by First() is to assign the return value to a variable that is declared with the `auto` type deduction keyword. For example, `auto x = myMapView->First();`.

UnorderedMapView::HasKey Method

Determines whether the current UnorderedMap contains the specified key.

Syntax

```
bool HasKey(K key);
```

Parameters

key
The key used to locate the element. The type of `key` is typename *K*.

Return Value

`true` if the key is found; otherwise, `false`.

UnorderedMapView::Lookup Method

Retrieves the value of type V that is associated with the specified key of type K.

Syntax

```
V Lookup(K key);
```

Parameters

key
The key used to locate an element in the UnorderedMapView. The type of `key` is typename *K*.

Return Value

The value that is paired with the `key`. The type of the return value is typename *V*.

UnorderedMapView::Size Method

Returns the number of Windows::Foundation::Collections::IKeyValuePair<K,V> elements in the UnorderedMapView.

Syntax

```
virtual property unsigned int Size;
```

Return Value

The number of elements in the Unordered MapView.

UnorderedMapView::Split Method

Divides the current UnorderedMapView object into two UnorderedMapView objects. This method is non-operational.

Syntax

```
void Split(
   Windows::Foundation::Collections::IMapView<
                        K,V>^ * firstPartition,
   Windows::Foundation::Collections::IMapView<
                        K,V>^ * secondPartition);
```

Parameters

firstPartition
The first part of the original UnorderedMapView object.

secondPartition
The second part of the original UnorderedMapView object.

Remarks

This method is not operational; it does nothing.

UnorderedMapView::UnorderedMapView Constructor

Initializes a new instance of the UnorderedMapView class.

Syntax

```
UnorderedMapView();
explicit UnorderedMapView(size_t n);
UnorderedMapView(size_t n, const H& h);
UnorderedMapView(size_t n, const H& h, const P& p);

explicit UnorderedMapView(
    const std::unordered_map<K, V, H, P>& m);
explicit UnorderedMapView(
    std::unordered_map<K, V, H, P>&& m);

template <typename InIt> UnorderedMapView(InIt first, InIt last );
template <typename InIt> UnorderedMapView(InIt first, InIt last, size_t n );

template <typename InIt> UnorderedMapView(
    InIt first,
    InIt last,
    size_t n,
    const H& h );

template <typename InIt> UnorderedMapView(
    InIt first,
    InIt last,
    size_t n,
    const H& h,
    const P& p );

UnorderedMapView(std::initializer_list<std::pair<const K, V>);

UnorderedMapView(std::initializer_list< std::pair<const K, V>> il, size_t n

UnorderedMapView(
    std::initializer_list< std::pair<const K, V>> il,
    size_t n,
    const H& h);

UnorderedMapView(
    std::initializer_list< std::pair<const K, V>> il,
    size_t n,
    const H& h,
    const P& p );
```

Parameters

n

The number of elements to preallocate space for.

InIt

The typename of the UnorderedMapView.

H

A function object that can a hash value for a key. Defaults to std::hash<K> for the types that `std::hash` supports.

P

A type that provides a function object that can compare two keys to determine their equality. Defaults to std::equal_to<K>.

m

A reference or Lvalues and Rvalues to a std::unordered_map that is used to initialize the UnorderedMapView.

first

The input iterator of the first element in a range of elements used to initialize the UnorderedMapView.

last

The input iterator of the first element after a range of elements used to initialize the UnorderedMapView.

See also

Platform::Collections Namespace
Windows::Foundation::IMapView

Platform::Collections::Vector Class

9/21/2022 • 6 minutes to read • Edit Online

Represents a sequential collection of objects that can be individually accessed by index. Implements Windows::Foundation::Collections::IObservableVector to help with XAML data binding.

Syntax

```
template <typename T, typename E>
   ref class Vector sealed;
```

Parameters

T

The type of the elements contained in the Vector object.

E

Specifies a binary predicate for testing equality with values of type *T*. The default value is `std::equal_to<T>`.

Remarks

Allowed types are:

1. integers
2. interface class^
3. public ref class^
4. value struct
5. public enum class

The **Vector** class is the C++ concrete implementation of the Windows::Foundation::Collections::IVector interface.

If you attempt to use a **Vector** type in a public return value or parameter, compiler error C3986 is raised. You can fix the error by changing the parameter or return value type to Windows::Foundation::Collections::IVector. For more information, see Collections (C++/CX).

Members

Public Constructors

NAME	DESCRIPTION
Vector::Vector	Initializes a new instance of the Vector class.

Public Methods

NAME	DESCRIPTION
Vector::Append	Inserts the specified item after the last item in the current Vector.
Vector::Clear	Deletes all the elements in the current Vector.

NAME	DESCRIPTION
Vector::First	Returns an iterator that specifies the first element in the Vector.
Vector::GetAt	Retrieves the element of the current Vector that is identifed by the specified index.
Vector::GetMany	Retrieves a sequence of items from the current Vector, starting at the specified index.
Vector::GetView	Returns a read-only view of a Vector; that is, a Platform::Collections::VectorView.
Vector::IndexOf	Searches for the specified item in the current Vector, and if found, returns the index of the item.
Vector::InsertAt	Inserts the specified item into the current Vector at the element identified by the specified index.
Vector::ReplaceAll	Deletes the elements in the current Vector and then inserts the elements from the specified array.
Vector::RemoveAt	Deletes the element identified by the specified index from the current Vector.
Vector::RemoveAtEnd	Deletes the element at the end of the current Vector.
Vector::SetAt	Assigns the specified value to the element in the current Vector that is identified by the specified index.
Vector::Size	Returns the number of elements in the current Vector object.

Events

NAME	DESCRIPTION
event Windows::Foundation::Collection::VectorChangedEventHandler<T>^ VectorChanged	Occurs when the Vector changes.

Inheritance Hierarchy

`Vector`

Requirements

Header: collection.h

Namespace: Platform::Collections

Vector::Append Method

Inserts the specified item after the last item in the current Vector.

Syntax

```
virtual void Append(T item);
```

Parameters

index
The item to insert into the Vector. The type of *item* is defined by the *T* typename.

Vector::Clear Method

Deletes all the elements in the current Vector.

Syntax

```
virtual void Clear();
```

Vector::First Method

Returns an iterator that points to the first element in the Vector.

Syntax

```
virtual Windows::Foundation::Collections::IIterator <T>^ First();
```

Return Value

An iterator that points to the first element in the Vector.

Remarks

A convenient way to hold the iterator returned by First() is to assign the return value to a variable that is declared with the `auto` type deduction keyword. For example, `auto x = myVector->First();`. This iterator knows the length of the collection.

When you need a pair of iterators to pass to an STL function, use the free functions Windows::Foundation::Collections::begin and Windows::Foundation::Collections::end

Vector::GetAt Method

Retrieves the element of the current Vector that is identifed by the specified index.

Syntax

```
virtual T GetAt(unsigned int index);
```

Parameters

index
A zero-based, unsigned integer that specifies a particular element in the Vector object.

Return Value

The element specified by the *index* parameter. The element type is defined by the *T* typename.

Vector::GetMany Method

Retrieves a sequence of items from the current Vector, starting at the specified index, and copies them into the caller-allocated array.

Syntax

```
virtual unsigned int GetMany(
    unsigned int startIndex,
    Platform::WriteOnlyArray<T>^ dest);
```

Parameters

startIndex
The zero-based index of the start of the items to retrieve.

dest
A caller-allocated array of items that begin at the element specified by *startIndex* and end at the last element in the Vector.

Return Value

The number of items retrieved.

Remarks

This function is not intended for use directly by client code. It is used internally in the to_vector Function to enable efficient conversion of Platform::Vector intances to std::vector instances.

Vector::GetView Method

Returns a read-only view of a Vector; that is, an IVectorView.

Syntax

```
Windows::Foundation::Collections::IVectorView<T>^ GetView();
```

Return Value

An IVectorView object.

Vector::IndexOf Method

Searches for the specified item in the current Vector, and if found, returns the index of the item.

Syntax

```
virtual bool IndexOf(T value, unsigned int* index);
```

Parameters

value
The item to find.

index
The zero-based index of the item if parameter *value* is found; otherwise, 0.

The *index* parameter is 0 if either the item is the first element of the Vector or the item was not found. If the return value is `true`, the item was found and it is the first element; otherwise, the item was not found.

Return Value

`true` if the specified item is found; otherwise, `false`.

Remarks

IndexOf uses std::find_if to find the item. Custom element types should therefore overload the == and !=

operator in order to enable the equality comparisons that find_if requires.

Vector::InsertAt Method

Inserts the specified item into the current Vector at the element identified by the specified index.

Syntax

```
virtual void InsertAt(unsigned int index, T item)
```

Parameters

index
A zero-based, unsigned integer that specifies a particular element in the Vector object.

item
An item to insert into the Vector at the element specified by *index*. The type of *item* is defined by the *T* typename.

Vector::RemoveAt Method

Deletes the element identified by the specified index from the current Vector.

Syntax

```
virtual void RemoveAt(unsigned int index);
```

Parameters

index
A zero-based, unsigned integer that specifies a particular element in the Vector object.

Vector::RemoveAtEnd Method

Deletes the element at the end of the current Vector.

Syntax

```
virtual void RemoveAtEnd();
```

Vector::ReplaceAll Method

Deletes the elements in the current Vector and then inserts the elements from the specified array.

Syntax

```
virtual void ReplaceAll(const ::Platform::Array<T>^ arr);
```

Parameters

arr
An array of objects whose type is defined by the *T* typename.

Vector::SetAt Method

Assigns the specified value to the element in the current Vector that is identified by the specified index.

Syntax

```
virtual void SetAt(unsigned int index, T item);
```

Parameters

index

A zero-based, unsigned integer that specifies a particular element in the Vector object.

item

The value to assign to the specified element. The type of *item* is defined by the *T* typename.

Vector::Size Method

Returns the number of elements in the current Vector object.

Syntax

```
virtual property unsigned int Size;
```

Return Value

The number of elements in the current Vector.

Vector::Vector Constructor

Initializes a new instance of the Vector class.

Syntax

```
Vector();

explicit Vector(unsigned int size);
Vector( unsigned int size, T value);
template <typename U> explicit Vector( const ::std::vector<U>& v);
template <typename U> explicit Vector( std::vector<U>&& v);

Vector( const T * ptr, unsigned int size);
template <size_t N> explicit Vector(const T(&arr)[N]);
template <size_t N> explicit Vector(const std::array<T, N>& a);
explicit Vector(const Array<T>^ arr);

template <typename InIt> Vector(InIt first, InIt last);
Vector(std::initializer_list<T> il);
```

Parameters

a

A std::array that will be used to initialize the Vector.

arr

A Platform::Array that will be used to initialize the Vector.

InIt

The type of a collection of objects that is used to initialize the current Vector.

il

A std::initializer_list of objects of type *T* that will be used to initialize the Vector.

N

The number of elements in a collection of objects that is used to initialize the current Vector.

size
The number of elements in the Vector.

value
A value that is used to initialize each element in the current Vector.

v
An Lvalues and Rvalues to a std::vector that is used to initialize the current Vector.

ptr
Pointer to a `std::vector` that is used to initialize the current Vector.

first
The first element in a sequence of objects that are used to initialize the current Vector. The type of *first* is passed by means of *perfect forwarding*. For more information, see Rvalue Reference Declarator: &&.

last
The last element in a sequence of objects that are used to initialize the current Vector. The type of *last* is passed by means of *perfect forwarding*. For more information, see Rvalue Reference Declarator: &&.

See also

Collections (C++/CX)
Platform Namespace
Creating Windows Runtime Components in C++

Platform::Collections::VectorIterator Class

9/21/2022 • 5 minutes to read • Edit Online

Provides a Standard Template Library iterator for objects derived from the Windows Runtime IVector interface.

VectorIterator is a proxy iterator that stores elements of type VectorProxy<T>. However, the proxy object is almost never visible to user code. For more information, see Collections (C++/CX).

Syntax

```
template <typename T>
class VectorIterator;
```

Parameters

T

The typename of the VectorIterator template class.

Members

Public Typedefs

NAME	DESCRIPTION
`difference_type`	A pointer difference (ptrdiff_t).
`iterator_category`	The category of a random access iterator (::std::random_access_iterator_tag).
`pointer`	A pointer to an internal type, Platform::Collections::Details::VectorProxy<T>, that is required for the implementation of VectorIterator.
`reference`	A reference to an internal type, Platform::Collections::Details::VectorProxy<T>,, that is required for the implementation of VectorIterator.
`value_type`	The `T` typename.

Public Constructors

NAME	DESCRIPTION
VectorIterator::VectorIterator	Initializes a new instance of the VectorIterator class.

Public Operators

NAME	DESCRIPTION
VectorIterator::operator- Operator	Subtracts either a specified number of elements from the current iterator yielding a new iterator, or a specified iterator from the current iterator yielding the number of elements between the iterators.

NAME	DESCRIPTION
VectorIterator::operator-- Operator	Decrements the current VectorIterator.
VectorIterator::operator!= Operator	Indicates whether the current VectorIterator is not equal to a specified VectorIterator.
VectorIterator::operator* Operator	Retrieves a reference to the element specified by the current VectorIterator.
VectorIterator::operator[]	Retrieves a reference to the element that is a specified displacement from the current VectorIterator.
VectorIterator::operator+ Operator	Returns a VectorIterator that references the element at the specified displacement from the specified VectorIterator.
VectorIterator::operator++ Operator	Increments the current VectorIterator.
VectorIterator::operator+= Operator	Increments the current VectorIterator by the specified displacement.
VectorIterator::operator< Operator	Indicates whether the current VectorIterator is less than a specified VectorIterator.
VectorIterator::operator<= Operator	Indicates whether the current VectorIterator is less than or equal to a specified VectorIterator.
VectorIterator::operator-= Operator	Decrements the current VectorIterator by the specified displacement.
VectorIterator::operator== Operator	Indicates whether the current VectorIterator is equal to a specified VectorIterator.
VectorIterator::operator> Operator	Indicates whether the current VectorIterator is greater than a specified VectorIterator.
VectorIterator::operator-> Operator	Retrieves the address of the element referenced by the current VectorIterator.
VectorIterator::operator>= Operator	Indicates whether the current VectorIterator is greater than or equal to a specified VectorIterator.

Inheritance Hierarchy

`VectorIterator`

Requirements

Header: collection.h

Namespace: Platform::Collections

`VectorIterator::operator->` Operator

Retrieves the address of the element referenced by the current VectorIterator.

Syntax

```
Detail::ArrowProxy<T> operator->() const;
```

Return Value

The value of the element that is referenced by the current VectorIterator.

The type of the return value is an unspecified internal type that is required for the implementation of this operator.

VectorIterator::operator-- Operator

Decrements the current VectorIterator.

Syntax

```
VectorIterator& operator--();
VectorIterator operator--(int);
```

Return Value

The first syntax decrements and then returns the current VectorIterator. The second syntax returns a copy of the current VectorIterator and then decrements the current VectorIterator.

Remarks

The first VectorIterator syntax pre-decrements the current VectorIterator.

The second syntax post-decrements the current VectorIterator. The `int` type in the second syntax indicates a post-decrement operation, not an actual integer operand.

VectorIterator::operator* Operator

Retrieves the address of the element specified by the current VectorIterator.

Syntax

```
reference operator*() const;
```

Return Value

The element specified by the current VectorIterator.

VectorIterator::operator== Operator

Indicates whether the current VectorIterator is equal to a specified VectorIterator.

Syntax

```
bool operator==(const VectorIterator& other) const;
```

Parameters

other
Another VectorIterator.

Return Value

`true` if the current VectorIterator is equal to *other*, otherwise, `false`.

`VectorIterator::operator>` Operator

Indicates whether the current VectorIterator is greater than a specified VectorIterator.

Syntax

```
bool operator>(const VectorIterator& other) const
```

Parameters

other
Another VectorIterator.

Return Value

`true` if the current VectorIterator is greater than *other*, otherwise, `false`.

`VectorIterator::operator>=` Operator

Indicates whether the current VectorIterator is greater than or equal to the specified VectorIterator.

Syntax

```
bool operator>=(const VectorIterator& other) const
```

Parameters

other
Another VectorIterator.

Return Value

`true` if the current VectorIterator is greater than or equal to *other*, otherwise, `false`.

VectorIterator::operator++ Operator

Increments the current VectorIterator.

Syntax

```
VectorIterator& operator++();
VectorIterator operator++(int);
```

Return Value

The first syntax increments and then returns the current VectorIterator. The second syntax returns a copy of the current VectorIterator and then increments the current VectorIterator.

Remarks

The first VectorIterator syntax pre-increments the current VectorIterator.

The second syntax post-increments the current VectorIterator. The `int` type in the second syntax indicates a post-increment operation, not an actual integer operand.

VectorIterator::operator!= Operator

Indicates whether the current VectorIterator is not equal to a specified VectorIterator.

Syntax

```
bool operator!=(const VectorIterator& other) const;
```

Parameters

other
Another VectorIterator.

Return Value

`true` if the current VectorIterator is not equal to *other*, otherwise, `false`.

`VectorIterator::operator<` Operator

Indicates whether the current VectorIterator is less than a specified VectorIterator.

Syntax

```
bool operator<(const VectorIterator& other) const
```

Parameters

other
Another VectorIterator.

Return Value

`true` if the current VectorIterator is less than *other*, otherwise, `false`.

`VectorIterator::operator<=` Operator

Indicates whether the current VectorIterator is less than or equal to a specified VectorIterator.

Syntax

```
bool operator<=(const VectorIterator& other) const
```

Parameters

other
Another VectorIterator.

Return Value

`true` if the current VectorIterator is less than or equal to *other*, otherwise, `false`.

VectorIterator::operator- Operator

Subtracts either a specified number of elements from the current iterator yielding a new iterator, or a specified iterator from the current iterator yielding the number of elements between the iterators.

Syntax

```
VectorIterator operator-(difference_type n) const;

difference_type operator-(const VectorIterator& other) const;
```

Parameters

n
A number of elements.

other
Another VectorIterator.

Return Value

The first operator syntax returns a VectorIterator object that is `n` elements less than the current VectorIterator. The second operator syntax returns the number of elements between the current and the `other` VectorIterator.

VectorIterator::operator+= Operator

Increments the current VectorIterator by the specified displacement.

Syntax

```
VectorIterator& operator+=(difference_type n);
```

Parameters

n
A integer displacement.

Return Value

The updated VectorIterator.

VectorIterator::operator+ Operator

Returns a VectorIterator that references the element at the specified displacement from the specified VectorIterator.

Syntax

```
VectorIterator operator+(difference_type n);

template <typename T>
inline VectorIterator<T> operator+(
  ptrdiff_t n,
  const VectorIterator<T>& i);
```

Parameters

T
In the second syntax, the typename of the VectorIterator.

n
An integer displacement.

i
In the second syntax, a VectorIterator.

Return Value

In the first syntax, a VectorIterator that references the element at the specified displacement from the current VectorIterator.

In the second syntax, a VectorIterator that references the element at the specified displacement from the beginning of parameter `i`.

Remarks

The first syntax example

VectorIterator::operator-= Operator

Decrements the current VectorIterator by the specified displacement.

Syntax

```
VectorIterator& operator-=(difference_type n);
```

Parameters

n
An integer displacement.

Return Value

The updated VectorIterator.

VectorIterator::operator[]

Retrieves a reference to the element that is a specified displacement from the current VectorIterator.

Syntax

```
reference operator[](difference_type n) const;
```

Parameters

n
An integer displacement.

Return Value

The element that is displaced by `n` elements from the current VectorIterator.

VectorIterator::VectorIterator Constructor

Initializes a new instance of the VectorIterator class.

Syntax

```
VectorIterator();

explicit VectorIterator(
   Windows::Foundation::Collections::IVector<T>^ v);
```

Parameters

v
An IVector<T> object.

Remarks

The first syntax example is the default constructor. The second syntax example is an explicit constructor that is used to construct a VectorIterator from an IVector<T> object.

See also

Platform Namespace

Platform::Collections::VectorView Class

9/21/2022 • 3 minutes to read • Edit Online

Represents a read-only view of a sequential collection of objects that can be individually accessed by index. The type of each object in the collection is specified by the template parameter.

Syntax

```
template <typename T, typename E>
   ref class VectorView sealed;
```

Parameters

T

The type of the elements contained in the `VectorView` object.

E

Specifies a binary predicate for testing equality with values of type `T`. The default value is `std::equal_to<T>`.

Remarks

The `VectorView` class implements the Windows::Foundation::Collections::IVectorView<T> interface, and support for Standard Template Library iterators.

Members

Public Constructors

NAME	DESCRIPTION
VectorView::VectorView	Initializes a new instance of the VectorView class.

Public Methods

NAME	DESCRIPTION
VectorView::First	Returns an iterator that specifies the first element in the VectorView.
VectorView::GetAt	Retrieves the element of the current VectorView that is indicated by the specified index.
VectorView::GetMany	Retrieves a sequence of items from the current VectorView, starting at the specified index.
VectorView::IndexOf	Searches for the specified item in the current VectorView, and if found, returns the index of the item.
VectorView::Size	Returns the number of elements in the current VectorView object.

Inheritance Hierarchy

`VectorView`

Requirements

Header: collection.h

Namespace: Platform::Collections

VectorView::First Method

Returns an iterator that specifies the first element in the VectorView.

Syntax

```
virtual Windows::Foundation::Collections::IIterator<T>^
   First();
```

Return Value

An iterator that specifies the first element in the VectorView.

Remarks

A convenient way to hold the iterator returned by First() is to assign the return value to a variable that is declared with the `auto` type deduction keyword. For example, `auto x = myVectorView->First();`.

VectorView::GetAt Method

Retrieves the element of the current VectorView that is indicated by the specified index.

Syntax

```
T GetAt(
   UInt32 index
);
```

Parameters

index
A zero-based, unsigned integer that specifies a particular element in the VectorView object.

Return Value

The element specified by the `index` parameter. The element type is specified by the VectorView template parameter, *T*.

VectorView::GetMany Method

Retrieves a sequence of items from the current VectorView, starting at the specified index.

Syntax

```
virtual unsigned int GetMany(
   unsigned int startIndex,
   ::Platform::WriteOnlyArray<T>^ dest
);
```

Parameters

startIndex
The zero-based index of the start of the items to retrieve.

dest
When this operation completes, an array of items that begin at the element specified by `startIndex` and end at the last element in the VectorView.

Return Value

The number of items retrieved.

VectorView::IndexOf Method

Searches for the specified item in the current VectorView, and if found, returns the index of the item.

Syntax

```
virtual bool IndexOf(
   T value,
   unsigned int* index
);
```

Parameters

value
The item to find.

index
The zero-based index of the item if parameter `value` is found; otherwise, 0.

The *index* parameter is 0 if either the item is the first element of the `VectorView` or the item was not found. If the return value is `true`, the item was found and it is the first element; otherwise, the item was not found.

Return Value

`true` if the specified item is found; otherwise, `false`.

VectorView::Size Method

Returns the number of elements in the current VectorView object.

Syntax

```
virtual property unsigned int Size;
```

Return Value

The number of elements in the current VectorView.

VectorView::VectorView Constructor

Initializes a new instance of the VectorView class.

Syntax

```
VectorView();
explicit VectorView(
   UInt32 size
);
VectorView(
   UInt32 size,
   T value
);
explicit VectorView(
   const ::std::vector<T>& v
);
explicit VectorView(
   ::std::vector<T>&& v
);
VectorView(
   const T * ptr,
   UInt32 size
);

template <
   size_t N
>
explicit VectorView(
   const T (&arr)[N]
);

template <
   size_t N
>
explicit VectorView(
   const ::std::array<T,
   N>& a
);

explicit VectorView(
   const ::Platform::Array<T>^ arr
);

template <
   typename InIt
>
VectorView(
   InItfirst,
   InItlast
);

VectorView(
   std::initializer_list<T> il
);
```

Parameters

InIt

The type of a collection of objects that is used to initialize the current VectorView.

il

A std::initializer_list whose elements will be used to initialize the VectorView.

N

The number of elements in a collection of objects that is used to initialize the current VectorView.

size

The number of elements in the VectorView.

value

A value that is used to initialize each element in the current VectorView.

v
An Lvalues and Rvalues to a std::vector that is used to initialize the current VectorView.

ptr
Pointer to a `std::vector` that is used to initialize the current VectorView.

arr
A Platform::Array object that is used to initialize the current VectorView.

a
A std::array object that is used to initialize the current VectorView.

first
The first element in a sequence of objects that are used to initialize the current VectorView. The type of `first` is passed by means of *perfect forwarding*. For more information, see Rvalue Reference Declarator: &&.

last
The last element in a sequence of objects that are used to initialize the current VectorView. The type of `last` is passed by means of *perfect forwarding*. For more information, see Rvalue Reference Declarator: &&.

See also

Platform Namespace
Creating Windows Runtime Components in C++

Platform::Collections::VectorViewIterator Class

9/21/2022 • 5 minutes to read • Edit Online

Provides a Standard Template Library iterator for objects derived from the Windows Runtime `IVectorView` interface.

`ViewVectorIterator` is a proxy iterator that stores elements of type `VectorProxy<T>`. However, the proxy object is almost never visible to user code. For more information, see Collections (C++/CX).

Syntax

```
template <typename T>
class VectorViewIterator;
```

Parameters

T

The typename of the VectorViewIterator template class.

Members

Public Typedefs

NAME	DESCRIPTION
`difference_type`	A pointer difference (ptrdiff_t).
`iterator_category`	The category of a random access iterator (::std::random_access_iterator_tag).
`pointer`	A pointer to an internal type that is required for the implementation of VectorViewIterator.
`reference`	A reference to an internal type that is required for the implementation of VectorViewIterator.
`value_type`	The `T` typename.

Public Constructors

NAME	DESCRIPTION
VectorViewIterator::VectorViewIterator	Initializes a new instance of the VectorViewIterator class.

Public Operators

NAME	DESCRIPTION
VectorViewIterator::operator- Operator	Subtracts either a specified number of elements from the current iterator yielding a new iterator, or a specified iterator from the current iterator yielding the number of elements between the iterators.

NAME	DESCRIPTION
VectorViewIterator::operator-- Operator	Decrements the current VectorViewIterator.
VectorViewIterator::operator!= Operator	Indicates whether the current VectorViewIterator is not equal to a specified VectorViewIterator.
VectorViewIterator::operator* Operator	Retrieves a reference to the element specified by the current VectorViewIterator.
VectorViewIterator::operator[]	Retrieves a reference to the element that is a specified displacement from the current VectorViewIterator.
VectorViewIterator::operator+ Operator	Returns a VectorViewIterator that references the element at the specified displacement from the specified VectorViewIterator.
VectorViewIterator::operator++ Operator	Increments the current VectorViewIterator.
VectorViewIterator::operator+= Operator	Increments the current VectorViewIterator by the specified displacement.
VectorViewIterator::operator< Operator	Indicates whether the current VectorViewIterator is less than a specified VectorViewIterator.
VectorViewIterator::operator<= Operator	Indicates whether the current VectorViewIterator is less than or equal to a specified VectorViewIterator.
VectorViewIterator::operator-= Operator	Decrements the current VectorViewIterator by the specified displacement.
VectorViewIterator::operator== Operator	Indicates whether the current VectorViewIterator is equal to a specified VectorViewIterator.
VectorViewIterator::operator> Operator	Indicates whether the current VectorViewIterator is greater than a specified VectorViewIterator.
VectorViewIterator::operator-> Operator	Retrieves the address of the element referenced by the current VectorViewIterator.
VectorViewIterator::operator>= Operator	Indicates whether the current VectorViewIterator is greater than or equal to a specified VectorViewIterator.

Inheritance Hierarchy

`VectorViewIterator`

Requirements

Header: collection.h

Namespace: Platform::Collections

`VectorViewIterator::operator->` Operator

Retrieves the address of the element referenced by the current VectorViewIterator.

Syntax

```
Detail::ArrowProxy<T> operator->() const;
```

Return Value

The value of the element that is referenced by the current VectorViewIterator.

The type of the return value is an unspecified internal type that is required for the implementation of this operator.

VectorViewIterator::operator-- Operator

Decrements the current VectorViewIterator.

Syntax

```
VectorViewIterator& operator--();
VectorViewIterator operator--(int);
```

Return Value

The first syntax decrements and then returns the current VectorViewIterator. The second syntax returns a copy of the current VectorViewIterator and then decrements the current VectorViewIterator.

Remarks

The first VectorViewIterator syntax pre-decrements the current VectorViewIterator.

The second syntax post-decrements the current VectorViewIterator. The `int` type in the second syntax indicates a post-decrement operation, not an actual integer operand.

VectorViewIterator::operator* Operator

Retrieves a reference to the element specified by the current VectorViewIterator.

Syntax

```
reference operator*() const;
```

Return Value

The element specified by the current VectorViewIterator.

VectorViewIterator::operator== Operator

Indicates whether the current VectorViewIterator is equal to a specified VectorViewIterator.

Syntax

```
bool operator==(const VectorViewIterator& other) const;
```

Parameters

other
Another VectorViewIterator.

Return Value

`true` if the current `VectorViewIterator` is equal to *other*, otherwise, `false`.

`VectorViewIterator::operator>` Operator

Indicates whether the current VectorViewIterator is greater than a specified VectorViewIterator.

Syntax

```
bool operator>(const VectorViewIterator& other) const;
```

Parameters

other
Another VectorViewIterator.

Return Value

`true` if the current VectorViewIterator is greater than *other*, otherwise, `false`.

`VectorViewIterator::operator>=` Operator

Indicates whether the current `VectorViewIterator` is greater than or equal to the specified `VectorViewIterator`.

Syntax

```
bool operator>=(const VectorViewIterator& other) const;
```

Parameters

other
Another VectorViewIterator.

Return Value

`true` if the current `VectorViewIterator` is greater than or equal to *other*, otherwise, `false`.

VectorViewIterator::operator++ Operator

Increments the current VectorViewIterator.

Syntax

```
VectorViewIterator& operator++();
VectorViewIterator operator++(int);
```

Return Value

The first syntax increments and then returns the current VectorViewIterator. The second syntax returns a copy of the current VectorViewIterator and then increments the current VectorViewIterator.

Remarks

The first VectorViewIterator syntax pre-increments the current VectorViewIterator.

The second syntax post-increments the current VectorViewIterator. The `int` type in the second syntax indicates a post-increment operation, not an actual integer operand.

VectorViewIterator::operator!= Operator

Indicates whether the current VectorViewIterator is not equal to a specified VectorViewIterator.

Syntax

```
bool operator!=(const VectorViewIterator& other) const;
```

Parameters

other
Another VectorViewIterator.

Return Value

`true` if the current `VectorViewIterator` is not equal to *other*; otherwise, `false`.

`VectorViewIterator::operator<` Operator

Indicates whether the current VectorIterator is less than a specified VectorIterator.

Syntax

```
bool operator<(const VectorViewIterator& other) const;
```

Parameters

other
Another `VectorIterator`.

Return Value

`true` if the current `VectorIterator` is less than *other*; otherwise, `false`.

`VectorViewIterator::operator<=` Operator

Indicates whether the current `VectorIterator` is less than or equal to a specified `VectorIterator`.

Syntax

```
bool operator<=(const VectorViewIterator& other) const;
```

Parameters

other
Another `VectorIterator`.

Return Value

`true` if the current `VectorIterator` is less than or equal to *other*; otherwise, `false`.

VectorViewIterator::operator- Operator

Subtracts either a specified number of elements from the current iterator yielding a new iterator, or a specified iterator from the current iterator yielding the number of elements between the iterators.

Syntax

```
VectorViewIterator operator-(difference_type n) const;

difference_type operator-(const VectorViewIterator& other) const;
```

Parameters

n
A number of elements.

other
Another VectorViewIterator.

Return Value

The first operator syntax returns a VectorViewIterator object that is `n` elements less than the current VectorViewIterator. The second operator syntax returns the number of elements between the current and the `other` VectorViewIterator.

VectorViewIterator::operator+= Operator

Increments the current VectorViewIterator by the specified displacement.

Syntax

```
VectorViewIterator& operator+=(difference_type n);
```

Parameters

n
A integer displacement.

Return Value

The updated VectorViewIterator.

VectorViewIterator::operator+ Operator

Returns a VectorViewIterator that references the element at the specified displacement from the specified VectorViewIterator.

Syntax

```
VectorViewIterator operator+(difference_type n) const;

template <typename T>
inline VectorViewIterator<T> operator+
   (ptrdiff_t n,
   const VectorViewIterator<T>& i);
```

Parameters

T
In the second syntax, the typename of the VectorViewIterator.

n
An integer displacement.

i

In the second syntax, a VectorViewIterator.

Return Value

In the first syntax, a VectorViewIterator that references the element at the specified displacement from the current VectorViewIterator.

In the second syntax, a VectorViewIterator that references the element at the specified displacement from the beginning of parameter `i`.

VectorViewIterator::operator-= Operator

Decrements the current VectorIterator by the specified displacement.

Syntax

```
VectorViewIterator& operator-=(difference_type n);
```

Parameters

n
An integer displacement.

Return Value

The updated VectorIterator.

VectorViewIterator::operator[]

Retrieves a reference to the element that is a specified displacement from the current VectorViewIterator.

Syntax

```
reference operator[](difference_type n) const;
```

Parameters

n
An integer displacement.

Return Value

The element that is displaced by `n` elements from the current VectorViewIterator.

VectorViewIterator::VectorViewIterator Constructor

Initializes a new instance of the VectorViewIterator class.

Syntax

```

VectorViewIterator();

explicit VectorViewIterator(
   Windows::Foundation::Collections::IVectorView<T>^ v
);
```

Parameters

v
An IVectorView<T> object.

Remarks

The first syntax example is the default constructor. The second syntax example is an explicit constructor that is used to construct a VectorViewIterator from an IVectorView<T> object.

See also

Platform Namespace

Platform::Collections::Details Namespace

9/21/2022 • 2 minutes to read • Edit Online

This namespace supports the `Platform` infrastructure and is not intended to be used directly from your code.

Syntax

```
namespace Platform {  namespace Collections {    namespace Details {}}}
```

Members

Members of this namespace are defined in collection.h and are not displayed in Object Browser.

Inheritance Hierarchy

Platform::Collections Namespace

Requirements

Header: Collection.h

Namespace: Platform::Collection::Details

See also

Platform Namespace

Platform::Details Namespace

9/21/2022 • 2 minutes to read • Edit Online

This namespace is intended for internal use only, and is not intended to be used for development.

Syntax

```
namespace Platform {
   namespace Details {
}}
```

Members

Although this namespace is intended for internal use, browsers can display the following members of this namespace.

NAME	REMARK
Console	Class. Displays output in unit tests.
_GUID	Struct
Heap	Class
HeapAllocationTrackingLevel	Enumeration
HeapEntryHandler	Delegate
IActivationFactory	Interface
IAgileObject	Interface
IClassFactory	Interface
IEquatable	Interface
IPrintable	Interface
IWeakReference	Interface
IWeakReferenceSource	Interface

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform::Details

See also

Platform Namespace

Platform::Details::__GUID Struct

9/21/2022 • 2 minutes to read • Edit Online

This struct is intended for internal use only, and is not intended to be used for development.

Syntax

```
ref struct __GUID;
```

Remarks

This struct is provided solely for completeness because it can be inspected with browsers.

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform::Details

See also

Platform Namespace

Platform::Details::Console Class

9/21/2022 • 2 minutes to read • Edit Online

This class is intended for internal use only, and is not intended to be used for development.

Syntax

```
ref class Console sealed;
```

Remarks

This class is provided solely for completeness because it can be inspected with browsers.

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform::Details

See also

Platform Namespace

Platform::Details::Heap Class

9/21/2022 • 2 minutes to read • Edit Online

This class is intended for internal use only, and is not intended to be used for development.

Syntax

```
ref class Heap sealed;
```

Remarks

This class is provided solely for completeness because it can be inspected with browsers.

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform::Details

See also

Platform Namespace

Platform::Details::HeapAllocationTrackingLevel Enumeration

9/21/2022 • 2 minutes to read • Edit Online

This enumeration is intended for internal use only, and is not intended to be used for development.

Syntax

```
enumm class HeapAllocationTrackingLevel;
```

Remarks

This enumeration is provided solely for completeness because it can be inspected with browsers.

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform::Details

See also

Platform Namespace

Platform::Details::HeapEntryHandler Delegate

9/21/2022 • 2 minutes to read • Edit Online

This delegate is intended for internal use only, and is not intended to be used for development.

Syntax

```
delegate HeapEntryHandler;
```

Remarks

This class is provided solely for completeness because it can be inspected with browsers.

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform::Details

See also

Platform Namespace

Platform::Details::IEquatable Interface

9/21/2022 • 2 minutes to read • Edit Online

This interface is intended for internal use only, and is not intended to be used for development.

Syntax

```
interface class IEquatable;
```

Remarks

This interface is provided solely for completeness because it can be inspected with browsers.

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform::Details

See also

Platform Namespace

Platform::Details::IPrintable Interface

9/21/2022 • 2 minutes to read • Edit Online

This interface is intended for internal use only, and is not intended to be used for development.

Syntax

```
interface class IPrintable;
```

Remarks

This interface is provided solely for completeness because it can be inspected with browsers.

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform::Details

See also

Platform Namespace

Platform::Metadata Namespace

9/21/2022 • 2 minutes to read • Edit Online

This namespace contains attributes that modify the declarations of types.

Syntax

```
namespace Platform {
   namespace Metadata {
}}
```

Members

Although this namespace is intended for internal use, browsers can display the following members of this namespace.

NAME	REMARK
Attribute	The base class for attributes.
Platform::Metadata::DefaultMemberAttribute Attribute	Indicates the preferred function to invoke among several possible overloaded functions.
Platform::Metadata::FlagsAttribute AttributeFlags	Declares an enumeration as an enumeration of bit fields. The following example shows how to apply the `Flags` attribute an enumeration. `[Flags] enum class MyEnumeration { enumA = 1, enumB = 2, enumC = 3}`
Platform::Metadata::RuntimeClassNameAttribute	Ensures that a private ref class has a valid runtime class name.

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform::Metadata

See also

Platform Namespace

Platform::Metadata::Attribute Attribute

9/21/2022 • 2 minutes to read • Edit Online

Represents the base class for all attributes.

Syntax

```
public ref class Attribute abstract : Object
```

Inheritance

Platform::Object

Platform::Metadata::Attribute

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform::Metadata

Metadata: platform.winmd

See also

Platform::Metadata Namespace

Platform::Metadata::DefaultMemberAttribute Attribute

9/21/2022 • 2 minutes to read • Edit Online

Indicates the preferred function to invoke among several possible overloaded functions.

Syntax

```
public ref class DefaultMember abstract : Attribute
```

Inheritance

Platform::Object

Platform::Metadata::Attribute

Remarks

Apply the DefaultMember attribute to a method that will be consumed by a JavaScript application.

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform::Metadata

Metadata: platform.winmd

See also

Platform::Metadata Namespace

Platform::Metadata::FlagsAttribute Attribute

9/21/2022 • 2 minutes to read • Edit Online

Indicates that an enumeration can be treated as a bit field; that is, a set of flags.

Syntax

```
public ref class Flags abstract : Attribute
```

Inheritance

Platform::Object

Platform::Metadata::Attribute

Remarks

Requirements

Minimum supported client: Windows 8

Minimum supported server: Windows Server 2012

Namespace: Platform::Metadata

Metadata: platform.winmd

See also

Platform::Metadata Namespace

Platform::Metadata::RuntimeClassName

9/21/2022 • 2 minutes to read • Edit Online

When applied to a class definition, ensures that a private class returns a valid name from the GetRuntimeClassName function..

Syntax

```
[Platform::Metadata::RuntimeClassName] name
```

Parameters

name

The name of an existing public type that is visible in the Windows Runtime.

Remarks

Use this attribute on private ref classes to specify a custom runtime type name and/or when the existing name does not meet the requirements. Specify as a name a public interface that the class implements.

Example

The following example shows how to use the attribute. In this example, the runtime type name of HellowWorldImpl is Test::Native::MyComponent::IHelloWorld

```
namespace Test
{
    namespace Native
    {
        namespace MyComponent
        {
            public interface class IHelloWorld
            {
                Platform::String^ SayHello();
            };

            private ref class HelloWorldImpl sealed :[Platform::Metadata::RuntimeClassName] IHelloWorld
            {
            public:
                HelloWorldImpl();
                virtual Platform::String^ SayHello();
            };

            Platform::String^ HelloWorldImpl::SayHello()
            {
                return L"Hello World!";
            }
        }
    }
}
```

See also

Platform::Metadata Namespace

Platform::Runtime::CompilerServices Namespace

9/21/2022 • 2 minutes to read • Edit Online

This namespace is intended for internal use only, and is not intended to be used for development.

Syntax

```
namespace Platform {
   namespace CompilerServices{
}}
```

Members

Although this namespace is intended for internal use, browsers can display the following members of this namespace.

NAME	REMARK
CallConvCdecl	
CallConvFastcall	
CallConvStdcall	
CallConvThiscall	
IndexerNameAttribute	
IsBoxed	
IsByValue	
IsConst	
IsCopyConstructed	
IsExplicitlyDereferenced	
IsImplicitlyDereferenced	
IsLong	
IsSignUnspecifiedByte	
IsSigned	
IsUdtReturn	
IsVolatile	

NAME	REMARK
OnePhaseConstructedAttribute	

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform::CompilerServices

See also

Platform Namespace

Platform::Runtime::InteropServices Namespace

9/21/2022 • 2 minutes to read • Edit Online

This namespace is intended for internal use only, and is not intended to be used for development.

Syntax

```
namespace Platform {
   namespace InteropServices {
}}
```

Members

Although this namespace is intended for internal use, browsers can display the following members of this namespace.

NAME	REMARK
ComInterfaceType	enumeration
InterfaceTypeAttribute	
LayoutKind	enumeration
MarshalAsAttribute	
StuctLayoutAttribute	
UnmanagedType	enumeration

Inheritance Hierarchy

`Platform`

Requirements

Metadata: platform.winmd

Namespace: Platform::InteropServices

See also

Platform Namespace

Windows::Foundation::Collections Namespace (C++/CX)

9/21/2022 • 2 minutes to read • Edit Online

C++/CX supplements the Windows::Foundation::Collections namespace with functions that simplify using the Vector, VectorView, Map, and MapView collection classes.

Syntax

```
namespace Windows {
    namespace Foundation {
        namespace Collections;
    }
}
```

Functions

NAME	DESCRIPTION
back_inserter Function	Returns an iterator that can be used to insert a value at the end of a collection.
begin Function	Returns an iterator that points to the beginning of a collection.
end Function	Returns an iterator that points beyond the end of a collection.
to_vector Function	Returns a collection as a std::vector.

Requirements

Header: collection.h

Namespace: Windows::Foundation::Collections

back_inserter Function

9/21/2022 • 2 minutes to read • Edit Online

Returns an iterator that is used to insert elements at the end of the specified collection.

Syntax

```
template <typename T>
Platform::BackInsertIterator<T>
    back_inserter(IVector<T>^ v);

template<typename T>
Platform::BackInsertIterator<T>
   back_inserter(IObservableVector<T>^ v);
```

Parameters

T

A template type parameter.

v

An interface pointer that provides access to the underlying collection.

Return Value

An iterator.

Requirements

Header: collection.h

Namespace: Windows::Foundation::Collections

See also

Windows::Foundation::Collections Namespace

begin Function

9/21/2022 • 2 minutes to read • Edit Online

Returns an iterator that points to the beginning of a collection that is accessed by the specified interface parameter.

Syntax

```
template <typename T>
    ::Platform::Collections::VectorIterator<T>
    begin(
          IVector<T>^ v         );

template <typename T>
    ::Platform::Collections::VectorViewIterator<T>
    begin(
          IVectorView<T>^ v
         );

template <typename T>
    ::Platform::Collections::InputIterator<T>
    begin(
          IIterable<T>^ i         );
```

Parameters

T

A template type parameter.

v

A collection of Vector<T> or VectorView<T> objects that are accessed by an IVector<T> or IVectorView<T> interface.

i

A collection of arbitrary Windows Runtime objects that are accessed by an IIterable<T> interface.

Return Value

An iterator that points to the beginning of the collection.

Remarks

The first two template functions return iterators, and the third template function returns an input iterator.

The VectorIterator object that is returned by begin is a proxy iterator that stores elements of type VectorProxy<T>. However, the proxy object is almost never visible to user code. For more information, see Collections (C++/CX).

Requirements

Header: collection.h

Namespace: Windows::Foundation::Collections

See also

Windows::Foundation::Collections Namespace

end Function

9/21/2022 • 2 minutes to read • Edit Online

Returns an iterator that points beyond the end of a collection that is accessed by the specified interface parameter.

Syntax

```
template <typename T>
    ::Platform::Collections::VectorIterator<T>
    end(
        IVector<T>^ v       );

template <typename T>
    ::Platform::Collections::VectorViewIterator<T>
    end(
        IVectorView<T>^ v
       );
template <typename T>
    ::Platform::Collections::InputIterator<T>
    end(
        IIterable<T>^ i
       );
```

Parameters

T

A template type parameter.

v

A collection of Vector<T> or VectorView<T> objects that are accessed by an IVector<T>, or IVectorView<T> interface.

i

A collection of arbitraty Windows Runtime objects that are accessed by an IIterable<T> interface.

Return Value

An iterator that points beyond the end of the collection.

Remarks

The first two template functions return iterators, and the third template function returns an input iterator.

The Platform::Collections::VectorViewIterator object that is returned by `end` is a proxy iterator that stores elements of type `VectorProxy<T>`. However, the proxy object is almost never visible to user code. For more information, see Collections (C++/CX).

Requirements

Header: collection.h

Namespace: Windows::Foundation::Collections

See also

Windows::Foundation::Collections Namespace

to_vector Function

9/21/2022 • 2 minutes to read • Edit Online

Returns a `std::vector` whose value is the same as the collection underlying the specified IVector or IVectorView parameter.

Syntax

```
template <typename T>
inline ::std::vector<T> to_vector(IVector<T>^ v);

template <typename T>
inline ::std::vector<T> to_vector(IVectorView<T>^ v);
```

Parameters

T

The template type parameter.

v

An IVector or IVectorView interface that provides access to an underlying Vector or VectorView object.

Return Value

Requirements

Header: collection.h

Namespace: Windows::Foundation::Collections

See also

Windows::Foundation::Collections Namespace

operator Windows::UI::Xaml::Interop::TypeName

9/21/2022 • 2 minutes to read • Edit Online

Enables conversion from `Platform::Type` to Windows::UI::Xaml::Interop::TypeName.

Syntax

```
Operator TypeName(Platform::Type^ type);
```

Return Value

Returns a Windows::UI::Xaml::Interop::TypeName when given a `Platform::Type^`.

Remarks

`TypeName` is the language-neutral Windows Runtime struct for representing type information. Platform::Type is specific to C++ and can't be passed across the application binary interface (ABI). Here's one use of `TypeName`, in the Navigate function:

```
rootFrame->Navigate(TypeName(MainPage::typeid), e->Arguments);
```

Example

The next example shows how to convert between `TypeName` and `Type`.

```
// Convert from Type to TypeName
Windows::UI::Xaml::Interop::TypeName tn = TypeName(MainPage::typeid);

// Convert back from TypeName to Type
Type^ tx2 = (Type^)(tn);
```

.NET Framework Equivalent

.NET Framework programs project `TypeName` as System.Type.

Requirements

See also

operator Windows::UI::Xaml::Interop::TypeName
Platform::Type Class

CRT functions not supported in Universal Windows Platform apps

9/21/2022 • 5 minutes to read • Edit Online

Many C runtime (CRT) functions aren't available when you build Universal Windows Platform (UWP) apps. Sometimes workarounds are available—-for example, you can use Windows Runtime or Win32 APIs. In other cases, CRT functions have been banned because the corresponding features or the supporting APIs aren't applicable to UWP apps. To look for an alternative method that's supported for the Windows Runtime, see Alternatives to Windows APIs in UWP apps.

The following table lists the CRT functions that are unavailable when you build UWP apps. It indicates any workarounds that apply.

Unsupported CRT Functions

FUNCTION	DESCRIPTION	WORKAROUND
`_beep` `_sleep` `_seterrormode`	These functions were obsolete in previous versions of the CRT. Also, the corresponding Win32 APIs are not available for UWP apps.	No workaround.
`chdir` `_chdrive` `getcwd`	These functions are obsolete or are not thread-safe.	Use `_chdir`, `_getcwd` and related functions.

FUNCTION	DESCRIPTION	WORKAROUND
`_cgets` `_cgets_s` `_cgetws` `_cgetws_s` `_cprintf` `_cprintf_l` `_cprintf_p` `_cprintf_p_l` `_cprintf_s` `_cprintf_s_l` `_cputs` `_cputws` `_cscanf` `_cscanf_l` `_cscanf_s` `_cscanf_s_l` `_cwait` `_cwprintf` `_cwprintf_l` `_cwprintf_p` `_cwprintf_p_l` `_cwprintf_s` `_cwprintf_s_l` `_cwscanf` `_cwscanf_l` `_cwscanf_s` `_cwscanf_s_l` `_vcprintf` `_vcprintf_l` `_vcprintf_p` `_vcprintf_p_l` `_vcprintf_s` `_vcprintf_s_l` `_vcwprintf` `_vcwprintf_l` `_vcwprintf_p` `_vcwprintf_p_l` `_vcwprintf_s` `_vcwprintf_s_l` `_getch` `_getch_nolock` `_getche` `_getche_nolock` `_getwch` `_getwch_nolock` `_getwche` `_getwche_nolock` `_putch` `_putch_nolock` `_putwch` `_putwch_nolock` `_ungetch` `_ungetch_nolock` `_ungetwch` `_ungetwch_nolock` `_kbhit` `kbhit` `putch` `cgets` `cprintf` `cputs` `cscanf` `cwait` `getch` `getche` `ungetch`	These console I/O functions are unavailable in GUI-based UWP apps.	UWP console apps can use these functions. For more information, see Create a Universal Windows Platform console app.
`getpid` `_getpid`	These functions are obsolete.	Use the Win32 API `GetCurrentProcessId`.
`_getdiskfree`	Not available.	Use the Win32 API `GetDiskFreeSpaceExW`.
`_getdrive` `_getdrives`	Corresponding API is not available for UWP apps.	No workaround.
`_inp` `_inpd` `_inpw` `_outp` `_outpd` `_outpw` `inp` `inpd` `inpw` `outp` `outpd` `outpw`	Port IO is not supported in UWP apps.	No workaround.
`_ismbcalnum` `_ismbcalnum_l` `_ismbcalpha` `_ismbcalpha_l` `_ismbcdigit` `_ismbcdigit_l` `_ismbcgraph` `_ismbcgraph_l` `_ismbchira` `_ismbchira_l` `_ismbckata` `_ismbckata_l` `_ismbcl0` `_ismbcl0_l` `_ismbcl1` `_ismbcl1_l` `_ismbcl2` `_ismbcl2_l` `_ismbclegal` `_ismbclegal_l` `_ismbclower` `_ismbclower_l` `_ismbcprint` `_ismbcprint_l` `_ismbcpunct`	Multi-byte strings are not supported in UWP apps.	Use Unicode strings instead.

FUNCTION	DESCRIPTION	WORKAROUND
_ismbcpunct_l _ismbcspace _ismbcspace_l _ismbcsymbol _ismbcsymbol_l _ismbcupper _ismbcupper_l _mbbtombc _mbbtombc_l _mbbtype _mbbtype_l _mbccpy _mbccpy_l _mbccpy_s _mbccpy_s_l _mbcjistojms _mbcjistojms_l _mbcjmstojis _mbcjmstojis_l _mbclen _mbclen_l _mbctohira _mbctohira_l _mbctokata _mbctokata_l _mbctolower _mbctolower_l _mbctombb _mbctombb_l _mbctoupper _mbctoupper_l _mbsbtype _mbsbtype_l _mbscat _mbscat_l _mbscat_s _mbscat_s_l _mbschr _mbschr_l _mbscmp _mbscmp_l _mbscoll _mbscoll_l _mbscpy _mbscpy_l _mbscpy_s _mbscpy_s_l _mbscspn _mbscspn_l _mbsdec _mbsdec_l _mbsicmp _mbsicmp_l _mbsicoll _mbsicoll_l _mbsinc _mbsinc_l _mbslen _mbslen_l _mbslwr _mbslwr_l _mbslwr_s _mbslwr_s_l _mbsnbcat _mbsnbcat_l _mbsnbcat_s _mbsnbcat_s_l _mbsnbcmp _mbsnbcmp_l _mbsnbcnt _mbsnbcnt_l _mbsnbcoll _mbsnbcoll_l _mbsnbcpy _mbsnbcpy_l _mbsnbcpy_s _mbsnbcpy_s_l _mbsnbicmp _mbsnbicmp_l _mbsnbicoll _mbsnbicoll_l _mbsnbset _mbsnbset_l _mbsnbset_s _mbsnbset_s_l _mbsncat _mbsncat_l _mbsncat_s _mbsncat_s_l _mbsnccnt _mbsnccnt_l _mbsncmp _mbsncmp_l _mbsncoll _mbsncoll_l _mbsncpy _mbsncpy_l _mbsncpy_s _mbsncpy_s_l _mbsnextc _mbsnextc_l _mbsnicmp _mbsnicmp_l _mbsnicoll _mbsnicoll_l _mbsninc _mbsninc_l _mbsnlen _mbsnlen_l _mbsnset _mbsnset_l _mbsnset_s _mbsnset_s_l _mbspbrk _mbspbrk_l _mbsrchr _mbsrchr_l _mbsrev _mbsrev_l _mbsset _mbsset_l _mbsset_s _mbsset_s_l _mbsspn _mbsspn_l _mbsspnp _mbsspnp_l _mbsstr _mbsstr_l _mbstok _mbstok_l _mbstok_s _mbstok_s_l _mbsupr		

FUNCTION	DESCRIPTION	WORKAROUND
`_mbsupr_l` `_mbsupr_s` `_mbsupr_s_l` `is_wctype`		
`_pclose` `_pipe` `_popen` `_wpopen`	Pipe functionality is not available to UWP apps.	No workaround.
`_resetstkoflw`	Supporting Win32 APIs are not available for UWP apps.	No workaround.
`_getsystime` `_setsystime`	These were obsolete APIs in previous CRT versions. Also, a user cannot set the system time in a UWP app due to lack of permissions.	To get the system time only, use the Win32 API `GetSystemTime`. System time cannot be set.
`_environ` `_putenv` `_putenv_s` `_searchenv` `_searchenv_s` `_dupenv_s` `_wputenv` `_wputenv_s` `_wsearchenv` getenv getenv_s putenv `_wdupenv_s` `_wenviron` `_wgetenv` `_wgetenv_s` `_wsearchenv_s` `tzset`	Environment variables are not available to UWP apps.	No workaround. To set the time zone, use `_tzset`.
`_loaddll` `_getdllprocaddr` `_unloaddll`	These were obsolete functions in previous CRT versions. Also, a user can't load DLLs except the ones in the same application package.	Use Win32 APIs `LoadPackagedLibrary`, `GetProcAddress`, and `FreeLibrary` to load and use packaged DLLs.
`_wexecl` `_wexecle` `_wexeclp` `_wexeclpe` `_wexecv` `_wexecve` `_wexecvp` `_wexecvpe` `_execl` `_execle` `_execlp` `_execlpe` `_execv` `_execve` `_execvp` `_execvpe` `_spawnl` `_spawnle` `_spawnlp` `_spawnlpe` `_spawnv` `_spawnve` `_spawnvp` `_spawnvpe` `_wspawnl` `_wspawnle` `_wspawnlp` `_wspawnlpe` `_wspawnv` `_wspawnve` `_wspawnvp` `_wspawnvpe` `_wsystem` `execl` `execle` `execlp` `execlpe` `execv` `execve` `execvp` `execvpe` `spawnl` `spawnle` `spawnlp` `spawnlpe` `spawnv` `spawnve` `spawnvp` `spawnvpe` `system`	The functionality is not available in UWP apps. A UWP app cannot invoke another UWP app or a desktop app.	No workaround.
`_heapwalk` `_heapadd` `_heapchk` `_heapset` `_heapused`	These functions are typically used to work with the heap. However, corresponding Win32 APIs are not supported in UWP apps. And, apps can no longer create or use private heaps.	No workaround. However, `_heapwalk` is available in the DEBUG CRT, for debugging purposes only. These functions can't be used in apps that are uploaded to the Microsoft Store.

The following functions are available in the CRT for UWP apps. However, use them only when you can't use the corresponding Win32 or Windows Runtime APIs, such as when you're porting large code bases:

FUNCTIONS	WORKAROUND

FUNCTIONS	WORKAROUND
Single-byte string functions—for example, `strcat`, `strcpy`, `strlwr`, and so on.	Make your UWP apps strictly Unicode because all Win32 APIs and Windows Runtime APIs that are exposed use Unicode character sets only. Single-byte functions were left for porting large code bases, but should otherwise be avoided. The corresponding wide char functions should be used instead when possible.
Stream IO and low-level file IO functions—for example, `fopen`, `open`, and so on.	These functions are synchronous, which isn't recommended for UWP apps. In your UWP apps, use asynchronous APIs to open, read from, and write to files to prevent locking of the UI thread. Examples of such APIs are the ones in the `Windows::Storage::FileIO` class.

Windows 8.x Store apps and Windows Phone 8.x apps

Both the previously mentioned APIs and the following APIs are unavailable in Windows 8.x Store apps and Windows Phone 8.x apps.

FUNCTIONS	DESCRIPTION	WORKAROUND
`_beginthread` `_beginthreadex` `_endthread` `_endthreadex`	Threading Win32 APIs are not available in Windows 8.x Store apps.	Use the `Windows Runtime Windows::System::Threading::ThreadPool` or `concurrency::task` instead.
`_chdir` `_wchdir` `_getcwd` `_getdcwd` `_wgetcwd` `_wgetdcwd`	The concept of a working directory doesn't apply to Windows 8.x Store apps.	Use full paths instead.
`_isleadbyte_l` `_ismbbalnum`, `_ismbbalnum_l`, `_ismbbalpha`, `_ismbbalpha` `_ismbbalpha_l` `_ismbbgraph` `_ismbbgraph_l` `_ismbbkalnum` `_ismbbkalnum_l` `_ismbbkana` `_ismbbkana_l` `_ismbbkprint` `_ismbbkprint_l` `_ismbbkpunct` `_ismbbkpunct_l` `_ismbblead` `_ismbblead_l` `_ismbbprint` `_ismbbprint_l` `_ismbbpunct` `_ismbbpunct_l` `_ismbbtrail` `_ismbbtrail_l` `_ismbslead` `_ismbslead_l` `_ismbstrail` `_ismbstrail_l` `_mbsdup` `isleadbyte`	Multi-byte strings are not supported in Windows 8.x Store apps.	Use Unicode strings instead.
`_tzset`	Environment variables are not available to Windows 8.x Store apps.	No workaround.
`_get_heap_handle`, `_heapmin`	The corresponding Win32 APIs are not supported in Windows 8.x Store apps. And, apps can no longer create private heaps.	No workaround. However, `_get_heap_handle` is available in the DEBUG CRT, for debugging purposes only.

Windows Runtime C++ Template Library (WRL)

9/21/2022 • 6 minutes to read • Edit Online

The Windows Runtime C++ Template Library (WRL) is a template library that provides a low-level way to author and use Windows Runtime components.

> **NOTE**
>
> WRL is now superseded by C++/WinRT, a standard C++17 language projection for Windows Runtime APIs. C++/WinRT is available in the Windows SDK from version 1803 (10.0.17134.0) onward. C++/WinRT is implemented entirely in header files, and designed to provide you with first-class access to the modern Windows API.
>
> With C++/WinRT, you can both consume and author Windows Runtime APIs using any standards-conformant C++17 compiler. C++/WinRT typically performs better and produces smaller binaries than any other language option for the Windows Runtime. We will continue to support C++/CX and WRL, but highly recommend that new applications use C++/WinRT. For more information, see C++/WinRT.

Benefits

The Windows Runtime C++ Template Library enables you to more easily implement and consume Component Object Model (COM) components. It provides housekeeping techniques like reference-counting to manage the lifetime of objects and testing HRESULT values to determine whether an operation succeeded or failed. To successfully use the Windows Runtime C++ Template Library, you must carefully follow these rules and techniques.

The C++/CX is a high-level, language-based way to use Windows Runtime components. Both the Windows Runtime C++ Template Library and C++/CX simplify the writing of code for the Windows Runtime by automatically performing housekeeping tasks on your behalf.

The Windows Runtime C++ Template Library and C++/CX provide different benefits. Here are some reasons you might want to use the Windows Runtime C++ Template Library instead of C++/CX:

- Windows Runtime C++ Template Library adds little abstraction over the Windows Runtime Application Binary Interface (ABI), giving you the ability to control the underlying code to better create or consume Windows Runtime APIs.
- C++/CX represents COM HRESULT values as exceptions. If you've inherited a code base that uses COM, or one that doesn't use exceptions, you might find that the Windows Runtime C++ Template Library is a more natural way to work with the Windows Runtime because you don't have to use exceptions.

  > **NOTE**
  >
  > The Windows Runtime C++ Template Library uses HRESULT values and does not throw exceptions. In addition, the Windows Runtime C++ Template Library uses smart pointers and the RAII pattern to help guarantee that objects are destroyed correctly when your application code throws an exception. For more info about smart pointers and RAII, see Smart Pointers and Objects Own Resources (RAII).

- The purpose and design of the Windows Runtime C++ Template Library is inspired by the Active Template Library (ATL), which is a set of template-based C++ classes that simplify the programming of COM objects. Because Windows Runtime C++ Template Library uses standard C++ to wrap the Windows Runtime, you can more easily port and interact with many existing COM components written in ATL to

the Windows Runtime. If you already know ATL, you might find that Windows Runtime C++ Template Library programming is easier.

Getting Started

Here are some resources that can help you get working with the Windows Runtime C++ Template Library right away.

How to: Activate and Use a Windows Runtime Component
Shows how to use the Windows Runtime C++ Template Library to initialize the Windows Runtime and activate and use a Windows Runtime component.

How to: Complete Asynchronous Operations
Shows how to use the Windows Runtime C++ Template Library to start asynchronous operations and perform work when the operations complete.

How to: Handle Events
Shows how to use the Windows Runtime C++ Template Library to subscribe to and handle the events of a Windows Runtime object.

Walkthrough: Creating a UWP app using WRL and Media Foundation
Learn how to create a UWP app that uses Microsoft Media Foundation.

How to: Create a Classic COM Component
Shows how to use the Windows Runtime C++ Template Library to create a basic COM component and a basic way to register and consume the COM component from a desktop app.

How to: Instantiate WRL Components Directly
Learn how to use the Microsoft::WRL::Make and Microsoft::WRL::Details::MakeAndInitialize functions to instantiate a component from the module that defines it.

How to: Use winmdidl.exe and midlrt.exe to create .h files from windows metadata
Shows how to consume custom Windows Runtime components from WRL by creating an IDL file from the .winmd metadata.

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 UWP app.

Bing Maps Trip Optimizer sample
Uses the `HttpRequest` class that's defined in Walkthrough: Connecting Using Tasks and XML HTTP Requests in the context of a complete UWP app.

Creating a Windows Runtime DLL component with C++ sample
Shows how to use the Windows Runtime C++ Template Library to create an in-process DLL component and consume it from C++/CX, JavaScript, and C#.

DirectX marble maze game sample
Demonstrates how to use the Windows Runtime C++ Template Library to manage the lifetime of COM components such as DirectX and Media Foundation in the context of a complete 3-D game.

Toast notifications from desktop apps
Demonstrates how to send toast notifications from a desktop app.

Windows Runtime C++ Template Library Compared to ATL

Windows Runtime C++ Template Library resembles the Active Template Library (ATL) because you can use it to create small, fast COM objects. Windows Runtime C++ Template Library and ATL also share concepts such as

definition of objects in modules, explicit registration of interfaces, and open creation of objects by using factories. You might be comfortable with Windows Runtime C++ Template Library if you're familiar with ATL.

Windows Runtime C++ Template Library supports the COM functionality that is required for UWP apps. Therefore, it differs from the ATL because it omits direct support for COM features such as:

- aggregation
- stock implementations
- dual interfaces (`IDispatch`)
- standard enumerator interfaces
- connection points
- tear-off interfaces
- OLE embedding
- ActiveX controls
- COM+

Concepts

Windows Runtime C++ Template Library provides types that represent a few basic concepts. The following sections describe those types.

ComPtr

ComPtr is a *smart pointer* type that represents the interface that's specified by the template parameter. Use `ComPtr` to declare a variable that can access the members of an object that's derived from the interface. `ComPtr` automatically maintains a reference count for the underlying interface pointer and releases the interface when the reference count goes to zero.

RuntimeClass

RuntimeClass represents an instantiated class that inherits a set of specified interfaces. A `RuntimeClass` object can provide a combination of support for one or more Windows Runtime COM interfaces, or a weak reference to a component.

Module

Module represents a collection of related objects. A `Module` object manages class factories, which create objects, and registration, which enables other applications to use an object.

Callback

The Callback function creates an object whose member function is an event handler (a callback method). Use the `Callback` function to write asynchronous operations.

EventSource

EventSource is used to manage *delegate* event handlers. Use Windows Runtime C++ Template Library to implement a delegate, and use `EventSource` to add, remove, and invoke delegates.

AsyncBase

AsyncBase provides virtual methods that represent the Windows Runtime asynchronous programming model. Override the members in this class to create a custom class that can start, stop, or check the progress of an asynchronous operation.

FtmBase

FtmBase represents a free-threaded marshaler object. `FtmBase` creates a global interface table (GIT), and helps manage marshaling and proxy objects.

WeakRef

WeakRef is a smart-pointer type that represents a *weak reference*, which references an object that might or might not be accessible. A `WeakRef` object can be used by only the Windows Runtime, and not by classic COM.

A `WeakRef` object typically represents an object whose existence is controlled by an external thread or application. For example, a `WeakRef` object can reference a file object. When the file is open, the `WeakRef` is valid and the referenced file is accessible. But when the file is closed, the `WeakRef` is invalid and the file is not accessible.

Related Topics

Key APIs by Category
Highlights the primary Windows Runtime C++ Template Library types, functions, and macros.

Reference
Contains reference information for the Windows Runtime C++ Template Library.

Quick Reference (C++/CX)
Briefly describes the C++/CX features that support the Windows Runtime.

Using Windows Runtime Components in Visual C++
Shows how to use C++/CX to create a basic Windows Runtime component.

How to: Activate and Use a Windows Runtime Component Using WRL

9/21/2022 • 4 minutes to read • Edit Online

This document shows how to use the Windows Runtime C++ Template Library (WRL)to initialize the Windows Runtime and how to activate and use a Windows Runtime component.

To use a component, you must acquire an interface pointer to the type that is implemented by the component. And because the underlying technology of the Windows Runtime is the Component Object Model (COM), you must follow COM rules to maintain an instance of the type. For example, you must maintain the *reference count* that determines when the type is deleted from memory.

To simplify the use of the Windows Runtime, Windows Runtime C++ Template Library provides the smart pointer template, ComPtr<T>, that automatically performs reference counting. When you declare a variable, specify `ComPtr<` *interface-name* `>` *identifier*. To access an interface member, apply the arrow member-access operator (`->`) to the identifier.

> **IMPORTANT**
> When you call an interface function, always test the HRESULT return value.

Activating and Using a Windows Runtime Component

The following steps use the `Windows::Foundation::IUriRuntimeClass` interface to demonstrate how to create an activation factory for a Windows Runtime component, create an instance of that component, and retrieve a property value. They also show how to initialize the Windows Runtime. The complete example follows.

> **IMPORTANT**
> Although you typically use the Windows Runtime C++ Template Library in a Universal Windows Platform (UWP) app, this example uses a console app for illustration. Functions such as `wprintf_s` are not available from a UWP app. For more information about the types and functions that you can use in a UWP app, see CRT functions not supported in Universal Windows Platform apps and Win32 and COM for UWP apps.

To activate and use a Windows Runtime component

1. Include (`#include`) any required Windows Runtime, Windows Runtime C++ Template Library, or C++ Standard Library headers.

```
#include <Windows.Foundation.h>
#include <wrl\wrappers\corewrappers.h>
#include <wrl\client.h>
#include <stdio.h>

using namespace ABI::Windows::Foundation;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;
```

   We recommend that you utilize the `using namespace` directive in your .cpp file to make the code more readable.

2. Initialize the thread in which the app executes. Every app must initialize its thread and threading model. This example uses the Microsoft::WRL::Wrappers::RoInitializeWrapper class to initialize the Windows Runtime and specifies RO_INIT_MULTITHREADED as the threading model. The `RoInitializeWrapper` class calls `Windows::Foundation::Initialize` at construction, and `Windows::Foundation::Uninitialize` when it is destroyed.

```
// Initialize the Windows Runtime.
RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
if (FAILED(initialize))
{
    return PrintError(__LINE__, initialize);
}
```

   In the second statement, the RoInitializeWrapper::HRESULT operator returns the `HRESULT` from the call to `Windows::Foundation::Initialize`.

3. Create an *activation factory* for the `ABI::Windows::Foundation::IUriRuntimeClassFactory` interface.

```
// Get the activation factory for the IUriRuntimeClass interface.
ComPtr<IUriRuntimeClassFactory> uriFactory;
HRESULT hr = GetActivationFactory(HStringReference(RuntimeClass_Windows_Foundation_Uri).Get(),
&uriFactory);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

   The Windows Runtime uses fully-qualified names to identify types. The `RuntimeClass_Windows_Foundation_Uri` parameter is a string that's provided by the Windows Runtime and contains the required runtime class name.

4. Initialize a Microsoft::WRL::Wrappers::HString variable that represents the URI `"https://www.microsoft.com"`.

```
// Create a string that represents a URI.
HString uriHString;
hr = uriHString.Set(L"http://www.microsoft.com");
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

   In the Windows Runtime, you don't allocate memory for a string that the Windows Runtime will use. Instead, the Windows Runtime creates a copy of your string in a buffer that it maintains and uses for operations, and then returns a handle to the buffer that it created.

5. Use the `IUriRuntimeClassFactory::CreateUri` factory method to create a `ABI::Windows::Foundation::IUriRuntimeClass` object.

```
// Create the IUriRuntimeClass object.
ComPtr<IUriRuntimeClass> uri;
hr = uriFactory->CreateUri(uriHString.Get(), &uri);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

6. Call the `IUriRuntimeClass::get_Domain` method to retrieve the value of the `Domain` property.

```
// Get the domain part of the URI.
HString domainName;
hr = uri->get_Domain(domainName.GetAddressOf());
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

7. Print the domain name to the console and return. All `ComPtr` and RAII objects leave scope and are released automatically.

```
// Print the domain name and return.
wprintf_s(L"Domain name: %s\n", domainName.GetRawBuffer(nullptr));

// All smart pointers and RAII objects go out of scope here.
```

The WindowsGetStringRawBuffer function retrieves the underlying Unicode form of the URI string.

Here's the complete example:

```
// wrl-consume-component.cpp
// compile with: runtimeobject.lib
#include <Windows.Foundation.h>
#include <wrl\wrappers\corewrappers.h>
#include <wrl\client.h>
#include <stdio.h>

using namespace ABI::Windows::Foundation;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;

// Prints an error string for the provided source code line and HRESULT
// value and returns the HRESULT value as an int.
int PrintError(unsigned int line, HRESULT hr)
{
    wprintf_s(L"ERROR: Line:%d HRESULT: 0x%X\n", line, hr);
    return hr;
}

int wmain()
{
    // Initialize the Windows Runtime.
    RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
    if (FAILED(initialize))
    {
        return PrintError(__LINE__, initialize);
    }

    // Get the activation factory for the IUriRuntimeClass interface.
    ComPtr<IUriRuntimeClassFactory> uriFactory;
    HRESULT hr = GetActivationFactory(HStringReference(RuntimeClass_Windows_Foundation_Uri).Get(),
&uriFactory);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Create a string that represents a URI.
    HString uriHString;
    hr = uriHString.Set(L"http://www.microsoft.com");
    if (FAILED(hr))
    {
```

```
    {
        return PrintError(__LINE__, hr);
    }

    // Create the IUriRuntimeClass object.
    ComPtr<IUriRuntimeClass> uri;
    hr = uriFactory->CreateUri(uriHString.Get(), &uri);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Get the domain part of the URI.
    HString domainName;
    hr = uri->get_Domain(domainName.GetAddressOf());
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Print the domain name and return.
    wprintf_s(L"Domain name: %s\n", domainName.GetRawBuffer(nullptr));

    // All smart pointers and RAII objects go out of scope here.
}
/*
Output:
Domain name: microsoft.com
*/
```

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 `wrl-consume-component.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe wrl-consume-component.cpp runtimeobject.lib
```

See also

Windows Runtime C++ Template Library (WRL)

How to: Complete Asynchronous Operations Using WRL

9/21/2022 • 9 minutes to read • Edit Online

This document shows how to use the Windows Runtime C++ Template Library (WRL) to start asynchronous operations and perform work when the operations complete.

This document shows two examples. The first example starts an asynchronous timer and waits for the timer to expire. In this example, you specify the asynchronous action when you create the timer object. The second example runs a background worker thread. This example shows how to work with a Windows Runtime method that returns an `IAsyncInfo` interface. The Callback function is an important part of both examples because it enables them to specify an event handler to process the results of the asynchronous operations.

For a more basic example that creates an instance of a component and retrieves a property value, see How to: Activate and Use a Windows Runtime Component.

> **TIP**
> These examples use lambda expressions to define the callbacks. You can also use function objects (functors), function pointers, or std::function objects. For more information about C++ lambda expressions, see Lambda Expressions.

Example: Working with a Timer

The following steps start an asynchronous timer and wait for the timer to expire. The complete example follows.

> **WARNING**
> Although you typically use the Windows Runtime C++ Template Library in a Universal Windows Platform (UWP) app, this example uses a console app for illustration. Functions such as `wprintf_s` are not available from a UWP app. For more information about the types and functions that you can use in a UWP app, see CRT functions not supported in Universal Windows Platform apps and Win32 and COM for UWP apps.

1. Include (`#include`) any required Windows Runtime, Windows Runtime C++ Template Library, or C++ Standard Library headers.

```
#include <Windows.Foundation.h>
#include <Windows.System.Threading.h>
#include <wrl/event.h>
#include <stdio.h>
#include <Objbase.h>

using namespace ABI::Windows::Foundation;
using namespace ABI::Windows::System::Threading;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;
```

`Windows.System.Threading.h` declares the types that are required to use an asynchronous timer.

We recommend that you utilize the `using namespace` directive in your .cpp file to make the code more readable.

2. Initialize the Windows Runtime.

```
// Initialize the Windows Runtime.
RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
if (FAILED(initialize))
{
    return PrintError(__LINE__, initialize);
}
```

3. Create an activation factory for the `ABI::Windows::System::Threading::IThreadPoolTimer` interface.

```
// Get the activation factory for the IThreadPoolTimer interface.
ComPtr<IThreadPoolTimerStatics> timerFactory;
HRESULT hr =
GetActivationFactory(HStringReference(RuntimeClass_Windows_System_Threading_ThreadPoolTimer).Get(),
&timerFactory);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

The Windows Runtime uses fully-qualified names to identify types. The `RuntimeClass_Windows_System_Threading_ThreadPoolTimer` parameter is a string that's provided by the Windows Runtime and contains the required runtime class name.

4. Create an Event object that synchronizes the timer callback to the main app.

```
// Create an event that is set after the timer callback completes. We later use this event to wait
for the timer to complete.
// This event is for demonstration only in a console app. In most apps, you typically don't wait for
async operations to complete.
Event timerCompleted(CreateEventEx(nullptr, nullptr, CREATE_EVENT_MANUAL_RESET, WRITE_OWNER |
EVENT_ALL_ACCESS));
hr = timerCompleted.IsValid() ? S_OK : HRESULT_FROM_WIN32(GetLastError());
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

> **NOTE**
>
> This event is for demonstration only as part of a console app. This example uses the event to ensure that an async operation completes before the app exits. In most apps, you typically don't wait for async operations to complete.

5. Create an `IThreadPoolTimer` object that expires after two seconds. Use the `Callback` function to create the event handler (an `ABI::Windows::System::Threading::ITimerElapsedHandler` object).

```
// Create a timer that prints a message after 2 seconds.

TimeSpan delay;
delay.Duration = 20000000; // 2 seconds.

auto callback = Callback<ITimerElapsedHandler>([&timerCompleted](IThreadPoolTimer* timer) -> HRESULT
{
    wprintf_s(L"Timer fired.\n");

    TimeSpan delay;
    HRESULT hr = timer->get_Delay(&delay);
    if (SUCCEEDED(hr))
    {
        wprintf_s(L"Timer duration: %2.2f seconds.\n", delay.Duration / 10000000.0);
    }

    // Set the completion event and return.
    SetEvent(timerCompleted.Get());
    return hr;
});
hr = callback ? S_OK : E_OUTOFMEMORY;
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}

ComPtr<IThreadPoolTimer> timer;
hr = timerFactory->CreateTimer(callback.Get(), delay, &timer);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

6. Print a message to the console and wait for the timer callback to complete. All `ComPtr` and RAII objects leave scope and are released automatically.

```
// Print a message and wait for the timer callback to complete.
wprintf_s(L"Timer started.\nWaiting for timer...\n");

// Wait for the timer to complete.
WaitForSingleObjectEx(timerCompleted.Get(), INFINITE, FALSE);
// All smart pointers and RAII objects go out of scope here.
```

Here is the complete example:

```
// wrl-consume-async.cpp
// compile with: runtimeobject.lib
#include <Windows.Foundation.h>
#include <Windows.System.Threading.h>
#include <wrl/event.h>
#include <stdio.h>
#include <Objbase.h>

using namespace ABI::Windows::Foundation;
using namespace ABI::Windows::System::Threading;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;

// Prints an error string for the provided source code line and HRESULT
// value and returns the HRESULT value as an int.
int PrintError(unsigned int line, HRESULT hr)
{
    wprintf_s(L"ERROR: Line:%d HRESULT: 0x%X\n", line, hr);
    return hr;
```

```
    return hr;
}

int wmain()
{
    // Initialize the Windows Runtime.
    RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
    if (FAILED(initialize))
    {
        return PrintError(__LINE__, initialize);
    }

    // Get the activation factory for the IThreadPoolTimer interface.
    ComPtr<IThreadPoolTimerStatics> timerFactory;
    HRESULT hr =
GetActivationFactory(HStringReference(RuntimeClass_Windows_System_Threading_ThreadPoolTimer).Get(),
&timerFactory);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Create an event that is set after the timer callback completes. We later use this event to wait for
the timer to complete.
    // This event is for demonstration only in a console app. In most apps, you typically don't wait for
async operations to complete.
    Event timerCompleted(CreateEventEx(nullptr, nullptr, CREATE_EVENT_MANUAL_RESET, WRITE_OWNER |
EVENT_ALL_ACCESS));
    hr = timerCompleted.IsValid() ? S_OK : HRESULT_FROM_WIN32(GetLastError());
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Create a timer that prints a message after 2 seconds.

    TimeSpan delay;
    delay.Duration = 20000000; // 2 seconds.

    auto callback = Callback<ITimerElapsedHandler>([&timerCompleted](IThreadPoolTimer* timer) -> HRESULT
    {
        wprintf_s(L"Timer fired.\n");

        TimeSpan delay;
        HRESULT hr = timer->get_Delay(&delay);
        if (SUCCEEDED(hr))
        {
            wprintf_s(L"Timer duration: %2.2f seconds.\n", delay.Duration / 10000000.0);
        }

        // Set the completion event and return.
        SetEvent(timerCompleted.Get());
        return hr;
    });
    hr = callback ? S_OK : E_OUTOFMEMORY;
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    ComPtr<IThreadPoolTimer> timer;
    hr = timerFactory->CreateTimer(callback.Get(), delay, &timer);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Print a message and wait for the timer callback to complete.
    wprintf_s(L"Timer started.\nWaiting for timer...\n");
```

```
    // Wait for the timer to complete.
    WaitForSingleObjectEx(timerCompleted.Get(), INFINITE, FALSE);
    // All smart pointers and RAII objects go out of scope here.
}
/*
Output:
Timer started.
Waiting for timer...
Timer fired.
Timer duration: 2.00 seconds.
*/
```

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 `wrl-consume-async.cpp` and then run the following command in a Visual Studio Command Prompt window.

```
cl.exe wrl-consume-async.cpp runtimeobject.lib
```

Example: Working with a Background Thread

The following steps start a worker thread and define the action that's performed by that thread. The complete example follows.

> **TIP**
> This example demonstrates how to work with the `ABI::Windows::Foundation::IAsyncAction` interface. You can apply this pattern to any interface that implements `IAsyncInfo`: `IAsyncAction`, `IAsyncActionWithProgress`, `IAsyncOperation`, and `IAsyncOperationWithProgress`.

1. Include (`#include`) any required Windows Runtime, Windows Runtime C++ Template Library, or C++ Standard Library headers.

   ```
   #include <Windows.Foundation.h>
   #include <Windows.System.Threading.h>
   #include <wrl/event.h>
   #include <stdio.h>
   #include <Objbase.h>

   using namespace ABI::Windows::Foundation;
   using namespace ABI::Windows::System::Threading;
   using namespace Microsoft::WRL;
   using namespace Microsoft::WRL::Wrappers;
   ```

   Windows.System.Threading.h declares the types that are required to use a worker thread.

   We recommend that you use the `using namespace` directive in your .cpp file to make the code more readable.

2. Initialize the Windows Runtime.

   ```
   // Initialize the Windows Runtime.
   RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
   if (FAILED(initialize))
   {
       return PrintError(__LINE__, initialize);
   }
   ```

3. Create an activation factory for the `ABI::Windows::System::Threading::IThreadPoolStatics` interface.

```
// Get the activation factory for the IThreadPoolStatics interface.
ComPtr<IThreadPoolStatics> threadPool;
HRESULT hr =
GetActivationFactory(HStringReference(RuntimeClass_Windows_System_Threading_ThreadPool).Get(),
&threadPool);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

4. Create an Event object that synchronizes completion of the worker thread to the main app.

```
// Create an event that is set after the timer callback completes. We later use this event to wait
for the timer to complete.
// This event is for demonstration only in a console app. In most apps, you typically don't wait for
async operations to complete.
Event threadCompleted(CreateEventEx(nullptr, nullptr, CREATE_EVENT_MANUAL_RESET, WRITE_OWNER |
EVENT_ALL_ACCESS));
hr = threadCompleted.IsValid() ? S_OK : HRESULT_FROM_WIN32(GetLastError());
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

> **NOTE**
>
> This event is for demonstration only as part of a console app. This example uses the event to ensure that an async operation completes before the app exits. In most apps, you typically don't wait for async operations to complete.

5. Call the `IThreadPoolStatics::RunAsync` method to create a worker thread. Use the `Callback` function to define the action.

```
wprintf_s(L"Starting thread...\n");

// Create a thread that computes prime numbers.
ComPtr<IAsyncAction> asyncAction;
hr = threadPool->RunAsync(Callback<IWorkItemHandler>([&threadCompleted](IAsyncAction* asyncAction) ->
HRESULT
{
    // Print a message.
    const unsigned int start = 0;
    const unsigned int end = 100000;
    unsigned int primeCount = 0;
    for (int n = start; n < end; n++)
    {
        if (IsPrime(n))
        {
            primeCount++;
        }
    }

    wprintf_s(L"There are %u prime numbers from %u to %u.\n", primeCount, start, end);

    // Set the completion event and return.
    SetEvent(threadCompleted.Get());
    return S_OK;

}).Get(), &asyncAction);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

The `IsPrime` function is defined in the complete example that follows.

6. Print a message to the console and wait for the thread to complete. All `ComPtr` and RAII objects leave scope and are released automatically.

```
// Print a message and wait for the thread to complete.
wprintf_s(L"Waiting for thread...\n");

// Wait for the thread to complete.
WaitForSingleObjectEx(threadCompleted.Get(), INFINITE, FALSE);

wprintf_s(L"Finished.\n");

// All smart pointers and RAII objects go out of scope here.
```

Here is the complete example:

```
// wrl-consume-asyncOp.cpp
// compile with: runtimeobject.lib
#include <Windows.Foundation.h>
#include <Windows.System.Threading.h>
#include <wrl/event.h>
#include <stdio.h>
#include <Objbase.h>

using namespace ABI::Windows::Foundation;
using namespace ABI::Windows::System::Threading;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;

// Prints an error string for the provided source code line and HRESULT
// value and returns the HRESULT value as an int.
int PrintError(unsigned int line, HRESULT hr)
```

```
int PrintError(unsigned int line, HRESULT hr)
{
    wprintf_s(L"ERROR: Line:%d HRESULT: 0x%X\n", line, hr);
    return hr;
}

// Determines whether the input value is prime.
bool IsPrime(int n)
{
    if (n < 2)
    {
        return false;
    }
    for (int i = 2; i < n; ++i)
    {
        if ((n % i) == 0)
        {
            return false;
        }
    }
    return true;
}

int wmain()
{
    // Initialize the Windows Runtime.
    RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
    if (FAILED(initialize))
    {
        return PrintError(__LINE__, initialize);
    }

    // Get the activation factory for the IThreadPoolStatics interface.
    ComPtr<IThreadPoolStatics> threadPool;
    HRESULT hr =
GetActivationFactory(HStringReference(RuntimeClass_Windows_System_Threading_ThreadPool).Get(), &threadPool);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Create an event that is set after the timer callback completes. We later use this event to wait for
the timer to complete.
    // This event is for demonstration only in a console app. In most apps, you typically don't wait for
async operations to complete.
    Event threadCompleted(CreateEventEx(nullptr, nullptr, CREATE_EVENT_MANUAL_RESET, WRITE_OWNER |
EVENT_ALL_ACCESS));
    hr = threadCompleted.IsValid() ? S_OK : HRESULT_FROM_WIN32(GetLastError());
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }


    wprintf_s(L"Starting thread...\n");

    // Create a thread that computes prime numbers.
    ComPtr<IAsyncAction> asyncAction;
    hr = threadPool->RunAsync(Callback<IWorkItemHandler>([&threadCompleted](IAsyncAction* asyncAction) ->
HRESULT
    {
        // Print a message.
        const unsigned int start = 0;
        const unsigned int end = 100000;
        unsigned int primeCount = 0;
        for (int n = start; n < end; n++)
        {
            if (IsPrime(n))
            {
                printf_s("%d ", n);
```

```
                primeCount++;
            }
        }

        wprintf_s(L"There are %u prime numbers from %u to %u.\n", primeCount, start, end);

        // Set the completion event and return.
        SetEvent(threadCompleted.Get());
        return S_OK;

    }).Get(), &asyncAction);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Print a message and wait for the thread to complete.
    wprintf_s(L"Waiting for thread...\n");

    // Wait for the thread to complete.
    WaitForSingleObjectEx(threadCompleted.Get(), INFINITE, FALSE);

    wprintf_s(L"Finished.\n");

    // All smart pointers and RAII objects go out of scope here.
}
/*
Output:
Starting thread...
Waiting for thread...
There are 9592 prime numbers from 0 to 100000.
Finished.
*/
```

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 `wrl-consume-asyncOp.cpp` and then run the following command in a **Visual Studio Command Prompt** window.

```
cl.exe wrl-consume-asyncOp.cpp runtimeobject.lib
```

See also

Windows Runtime C++ Template Library (WRL)

How to: Handle Events Using WRL

9/21/2022 • 7 minutes to read • Edit Online

This document shows how to use the Windows Runtime C++ Template Library (WRL)to subscribe to and handle the events of a Windows Runtime object.

For a more basic example that creates an instance of that component and retrieves a property value, see How to: Activate and Use a Windows Runtime Component.

Subscribing to and Handling Events

The following steps start an `ABI::Windows::System::Threading::IDeviceWatcher` object and use event handlers to monitor progress. The `IDeviceWatcher` interface enables you to enumerate devices asynchronously, or in the background, and receive notification when devices are added, removed, or changed. The Callback function is an important part of this example because it enables it to specify event handlers that process the results of the background operation. The complete example follows.

> **WARNING**
> Although you typically use the Windows Runtime C++ Template Library in a Universal Windows Platform app, this example uses a console app for illustration. Functions such as `wprintf_s` are not available from a Universal Windows Platform app. For more information about the types and functions that you can use in a Universal Windows Platform app, see CRT functions not supported in Universal Windows Platform apps and Win32 and COM for UWP apps.

1. Include (`#include`) any required Windows Runtime, Windows Runtime C++ Template Library, or C++ Standard Library headers.

```
#include <Windows.Devices.Enumeration.h>
#include <wrl/event.h>
#include <stdio.h>

using namespace ABI::Windows::Devices::Enumeration;
using namespace ABI::Windows::Foundation;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;
```

   `Windows.Devices.Enumeration.h` declares the types that are required to enumerate devices.

   We recommend that you utilize the `using namespace` directive in your .cpp file to make the code more readable.

2. Declare the local variables for the app. This example holds count of the number of enumerated devices and registration tokens that enable it to later unsubscribe from events.

```
// Counts the number of enumerated devices.
unsigned int deviceCount = 0;

// Event registration tokens that enable us to later unsubscribe from events.
EventRegistrationToken addedToken;
EventRegistrationToken stoppedToken;
EventRegistrationToken enumCompletedToken;
```

3. Initialize the Windows Runtime.

```
// Initialize the Windows Runtime.
RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
if (FAILED(initialize))
{
    return PrintError(__LINE__, initialize);
}
```

4. Create an Event object that synchronizes the completion of the enumeration process to the main app.

```
// Create an event that is set after device enumeration completes. We later use this event to wait
for the timer to complete.
// This event is for demonstration only in a console app. In most apps, you typically don't wait for
async operations to complete.
Event enumerationCompleted(CreateEventEx(nullptr, nullptr, CREATE_EVENT_MANUAL_RESET, WRITE_OWNER |
EVENT_ALL_ACCESS));
HRESULT hr = enumerationCompleted.IsValid() ? S_OK : HRESULT_FROM_WIN32(GetLastError());
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

> **NOTE**
>
> This event is for demonstration only as part of a console app. This example uses the event to ensure that an async operation completes before the app exits. In most apps, you typically don't wait for async operations to complete.

5. Create an activation factory for the `IDeviceWatcher` interface.

```
// Get the activation factory for the IDeviceWatcher interface.
ComPtr<IDeviceInformationStatics> watcherFactory;
hr =
ABI::Windows::Foundation::GetActivationFactory(HStringReference(RuntimeClass_Windows_Devices_Enumerat
ion_DeviceInformation).Get(), &watcherFactory);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

The Windows Runtime uses fully-qualified names to identify types. The `RuntimeClass_Windows_Devices_Enumeration_DeviceInformation` parameter is a string that's provided by the Windows Runtime and contains the required runtime class name.

6. Create the `IDeviceWatcher` object.

```
// Create a IDeviceWatcher object from the factory.
ComPtr<IDeviceWatcher> watcher;
hr = watcherFactory->CreateWatcher(&watcher);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

7. Use the `Callback` function to subscribe to the `Added`, `EnumerationCompleted`, and `Stopped` events.

```
// Subscribe to the Added event.
hr = watcher->add_Added(Callback<AddedHandler>([&deviceCount](IDeviceWatcher* watcher,
IDeviceInformation*) -> HRESULT
{
    // Print a message and increment the device count.
    // When we reach 10 devices, stop enumerating devices.
    wprintf_s(L"Added device...\n");
    deviceCount++;
    if (deviceCount == 10)
    {
        return watcher->Stop();
    }
    return S_OK;

}).Get(), &addedToken);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}

hr = watcher->add_Stopped(Callback<StoppedHandler>([=, &enumerationCompleted](IDeviceWatcher*
watcher, IInspectable*) -> HRESULT
{
    wprintf_s(L"Device enumeration stopped.\nRemoving event handlers...");

    // Unsubscribe from the events. This is shown for demonstration.
    // The need to remove event handlers depends on the requirements of
    // your app. For instance, if you only need to handle an event for
    // a short period of time, you might remove the event handler when you
    // no longer need it. If you handle an event for the duration of the app,
    // you might not need to explicitly remove it.
    HRESULT hr1 = watcher->remove_Added(addedToken);
    HRESULT hr2 = watcher->remove_Stopped(stoppedToken);
    HRESULT hr3 = watcher->remove_EnumerationCompleted(enumCompletedToken);

    // Set the completion event and return.
    SetEvent(enumerationCompleted.Get());

    return FAILED(hr1) ? hr1 : FAILED(hr2) ? hr2 : hr3;

}).Get(), &stoppedToken);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}

// Subscribe to the EnumerationCompleted event.
hr = watcher->add_EnumerationCompleted(Callback<EnumerationCompletedHandler>([](IDeviceWatcher*
watcher, IInspectable*) -> HRESULT
{
    wprintf_s(L"Enumeration completed.\n");

    return watcher->Stop();

}).Get(), &enumCompletedToken);
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

The `Added` event handler increments the count of enumerated devices. It stops the enumeration process after ten devices are found.

The `Stopped` event handler removes the event handlers and sets the completion event.

The `EnumerationCompleted` event handler stops the enumeration process. We handle this event in case

there are fewer than ten devices.

> **TIP**
> This example uses a lambda expression to define the callbacks. You can also use function objects (functors), function pointers, or std::function objects. For more information about lambda expressions, see Lambda Expressions.

8. Start the enumeration process.

```
wprintf_s(L"Starting device enumeration...\n");
hr = watcher->Start();
if (FAILED(hr))
{
    return PrintError(__LINE__, hr);
}
```

9. Wait for the enumeration process to complete and then print a message. All `ComPtr` and RAII objects leave scope and are released automatically.

```
// Wait for the operation to complete.
WaitForSingleObjectEx(enumerationCompleted.Get(), INFINITE, FALSE);

wprintf_s(L"Enumerated %u devices.\n", deviceCount);

// All smart pointers and RAII objects go out of scope here.
```

Here is the complete example:

```
// wrl-consume-events.cpp
// compile with: runtimeobject.lib
#include <Windows.Devices.Enumeration.h>
#include <wrl/event.h>
#include <stdio.h>

using namespace ABI::Windows::Devices::Enumeration;
using namespace ABI::Windows::Foundation;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;

// Prints an error string for the provided source code line and HRESULT
// value and returns the HRESULT value as an int.
int PrintError(unsigned int line, HRESULT hr)
{
    wprintf_s(L"ERROR: Line:%d HRESULT: 0x%X\n", line, hr);
    return hr;
}

int wmain()
{
    // Type define the event handler types to make the code more readable.
    typedef
__FITypedEventHandler_2_Windows__CDevices__CEnumeration__CDeviceWatcher_Windows__CDevices__CEnumeration__CDe
viceInformation AddedHandler;
    typedef __FITypedEventHandler_2_Windows__CDevices__CEnumeration__CDeviceWatcher_IInspectable
EnumerationCompletedHandler;
    typedef __FITypedEventHandler_2_Windows__CDevices__CEnumeration__CDeviceWatcher_IInspectable
StoppedHandler;

    // Counts the number of enumerated devices.
    unsigned int deviceCount = 0;
```

```
    // Event registration tokens that enable us to later unsubscribe from events.
    EventRegistrationToken addedToken;
    EventRegistrationToken stoppedToken;
    EventRegistrationToken enumCompletedToken;

    // Initialize the Windows Runtime.
    RoInitializeWrapper initialize(RO_INIT_MULTITHREADED);
    if (FAILED(initialize))
    {
        return PrintError(__LINE__, initialize);
    }

    // Create an event that is set after device enumeration completes. We later use this event to wait for
the timer to complete.
    // This event is for demonstration only in a console app. In most apps, you typically don't wait for
async operations to complete.
    Event enumerationCompleted(CreateEventEx(nullptr, nullptr, CREATE_EVENT_MANUAL_RESET, WRITE_OWNER |
EVENT_ALL_ACCESS));
    HRESULT hr = enumerationCompleted.IsValid() ? S_OK : HRESULT_FROM_WIN32(GetLastError());
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Get the activation factory for the IDeviceWatcher interface.
    ComPtr<IDeviceInformationStatics> watcherFactory;
    hr =
ABI::Windows::Foundation::GetActivationFactory(HStringReference(RuntimeClass_Windows_Devices_Enumeration_Dev
iceInformation).Get(), &watcherFactory);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Create a IDeviceWatcher object from the factory.
    ComPtr<IDeviceWatcher> watcher;
    hr = watcherFactory->CreateWatcher(&watcher);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Subscribe to the Added event.
    hr = watcher->add_Added(Callback<AddedHandler>([&deviceCount](IDeviceWatcher* watcher,
IDeviceInformation*) -> HRESULT
    {
        // Print a message and increment the device count.
        // When we reach 10 devices, stop enumerating devices.
        wprintf_s(L"Added device...\n");
        deviceCount++;
        if (deviceCount == 10)
        {
            return watcher->Stop();
        }
        return S_OK;

    }).Get(), &addedToken);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    hr = watcher->add_Stopped(Callback<StoppedHandler>([=, &enumerationCompleted](IDeviceWatcher* watcher,
IInspectable*) -> HRESULT
    {
        wprintf_s(L"Device enumeration stopped.\nRemoving event handlers...");

        // Unsubscribe from the events. This is shown for demonstration.
```

```
        // The need to remove event handlers depends on the requirements of
        // your app. For instance, if you only need to handle an event for
        // a short period of time, you might remove the event handler when you
        // no longer need it. If you handle an event for the duration of the app,
        // you might not need to explicitly remove it.
        HRESULT hr1 = watcher->remove_Added(addedToken);
        HRESULT hr2 = watcher->remove_Stopped(stoppedToken);
        HRESULT hr3 = watcher->remove_EnumerationCompleted(enumCompletedToken);

        // Set the completion event and return.
        SetEvent(enumerationCompleted.Get());

        return FAILED(hr1) ? hr1 : FAILED(hr2) ? hr2 : hr3;

    }).Get(), &stoppedToken);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Subscribe to the EnumerationCompleted event.
    hr = watcher->add_EnumerationCompleted(Callback<EnumerationCompletedHandler>([](IDeviceWatcher* watcher,
IInspectable*) -> HRESULT
    {
        wprintf_s(L"Enumeration completed.\n");

        return watcher->Stop();

    }).Get(), &enumCompletedToken);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    wprintf_s(L"Starting device enumeration...\n");
    hr = watcher->Start();
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    // Wait for the operation to complete.
    WaitForSingleObjectEx(enumerationCompleted.Get(), INFINITE, FALSE);

    wprintf_s(L"Enumerated %u devices.\n", deviceCount);

    // All smart pointers and RAII objects go out of scope here.
}
/*
Sample output:
Starting device enumeration...
Added device...
Added device...
Added device...
Added device...
Added device...
Added device...
Added device...
Added device...
Added device...
Added device...
Device enumeration stopped.
Removing event handlers...
Enumerated 10 devices.
*/
```

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 `wrl-consume-events.cpp` and then run the following command in a **Visual Studio Command Prompt** window.

```
cl.exe wrl-consume-events.cpp runtimeobject.lib
```

See also

Windows Runtime C++ Template Library (WRL)

Walkthrough: Creating a UWP app using WRL and Media Foundation

9/21/2022 • 34 minutes to read • Edit Online

> **NOTE**
> For new UWP apps and components, we recommend that you use C++/WinRT, a new standard C++17 language projection for Windows Runtime APIs. C++/WinRT is available in the Windows SDK from version 1803 (10.0.17134.0) onward. C++/WinRT is implemented entirely in header files, and is designed to provide you with first-class access to the modern Windows API.

In this tutorial, you'll learn how to use the Windows Runtime C++ Template Library (WRL) to create a Universal Windows Platform (UWP) app that uses Microsoft Media Foundation.

This example creates a custom Media Foundation transform. It applies a grayscale effect to images that are captured from a webcam. The app uses C++ to define the custom transform and C# to use the component to transform the captured images.

> **NOTE**
> Instead of C#, you can also use JavaScript, Visual Basic, or C++ to consume the custom transform component.

You can usually use C++/CX to create Windows Runtime components. However, sometimes you have to use the WRL. For example, when you create a media extension for Microsoft Media Foundation, you must create a component that implements both COM and Windows Runtime interfaces. Because C++/CX can only create Windows Runtime objects, to create a media extension you must use the WRL. That's because it enables the implementation of both COM and Windows Runtime interfaces.

> **NOTE**
> Although this code example is long, it demonstrates the minimum that's required to create a useful Media Foundation transform. You can use it as a starting point for your own custom transform. This example is adapted from the Media extensions sample, which uses media extensions to apply effects to video, decode video, and create scheme handlers that produce media streams.

Prerequisites

- In Visual Studio 2017 and later, UWP support is an optional component. To install it, open the Visual Studio Installer from the Windows Start menu and find your version of Visual Studio. Choose **Modify** and then make sure the **Universal Windows Platform Development** tile is checked. Under **Optional Components** check **C++ Tools for UWP (v141)** for Visual Studio 2017, or **C++ Tools for UWP (v142)** for Visual Studio 2019. Then check the version of the Windows SDK that you want to use.
- Experience with the Windows Runtime.
- Experience with COM.
- A webcam.

Key points

- To create a custom Media Foundation component, use a Microsoft Interface Definition Language (MIDL) definition file to define an interface, implement that interface, and then make it activatable from other components.
- The `namespace` and `runtimeclass` attributes, and the `NTDDI_WIN8` version attribute value are important parts of the MIDL definition for a Media Foundation component that uses WRL.
- `Microsoft::WRL::RuntimeClass` is the base class for the custom Media Foundation component. The [`Microsoft::WRL::RuntimeClassType::WinRtClassicComMix]` (runtimeclasstype-enumeration.md) enum value, which is provided as a template argument, marks the class for use both as a Windows Runtime class and as a classic COM runtime class.
- The `InspectableClass` macro implements basic COM functionality such as reference counting and the `QueryInterface` method, and sets the runtime class name and trust level.
- Use the `Microsoft::WRL::Module` class to implement DLL entry-point functions such as `DllGetActivationFactory`, `DllCanUnloadNow`, and `DllGetClassObject`.
- Link your component DLL to *`runtimeobject.lib`*. Also specify `/WINMD` on the linker line to generate Windows metadata.
- Use project references to make WRL components accessible to UWP apps.

To use the WRL to create the Media Foundation grayscale transform component

1. In Visual Studio, create a **Blank Solution** project. Name the project, for example, *MediaCapture*.
2. Add a **DLL (Universal Windows)** project to the solution. Name the project, for example, *GrayscaleTransform*.
3. Add a **MIDL File (.idl)** file to the project. Name the file, for example, *`GrayscaleTransform.idl`*.
4. Add this code to GrayscaleTransform.idl:

```
import "Windows.Media.idl";

#include <sdkddkver.h>

namespace GrayscaleTransform
{
    [version(NTDDI_WIN8), activatable(NTDDI_WIN8)]
    runtimeclass GrayscaleEffect
    {
        [default] interface Windows.Media.IMediaExtension;
    }
}
```

5. Use the following code to replace the contents of *`pch.h`*:

```
#pragma once

#include "targetver.h"

#include <new>
#include <mfapi.h>
#include <mftransform.h>
#include <mfidl.h>
#include <mferror.h>
#include <strsafe.h>
#include <assert.h>

// Note: The Direct2D helper library is included for its 2D matrix operations.
#include <D2d1helper.h>

#include <wrl\implements.h>
#include <wrl\module.h>
#include <windows.media.h>
```

6. Add a new header file to the project, name it `BufferLock.h`, and then replace the contents with this code:

```
#pragma once

//  Locks a video buffer that might or might not support IMF2DBuffer.

class VideoBufferLock
{
public:
    VideoBufferLock(IMFMediaBuffer *pBuffer) : m_p2DBuffer(nullptr)
    {
        m_pBuffer = pBuffer;
        m_pBuffer->AddRef();

        // Query for the 2-D buffer interface. OK if this fails.
        m_pBuffer->QueryInterface(IID_PPV_ARGS(&m_p2DBuffer));
    }

    ~VideoBufferLock()
    {
        UnlockBuffer();
        m_pBuffer->Release();
        if (m_p2DBuffer)
        {
            m_p2DBuffer->Release();
        }
    }

    // LockBuffer:
    // Locks the buffer. Returns a pointer to scan line 0 and returns the stride.

    // The caller must provide the default stride as an input parameter, in case
    // the buffer does not expose IMF2DBuffer. You can calculate the default stride
    // from the media type.

    HRESULT LockBuffer(
        LONG  lDefaultStride,    // Minimum stride (with no padding).
        DWORD dwHeightInPixels,  // Height of the image, in pixels.
        BYTE  **ppbScanLine0,    // Receives a pointer to the start of scan line 0.
        LONG  *plStride          // Receives the actual stride.
        )
    {
        HRESULT hr = S_OK;

        // Use the 2-D version if available.
        if (m_p2DBuffer)
        {
```

```
            hr = m_p2DBuffer->Lock2D(ppbScanLine0, plStride);
        }
        else
        {
            // Use non-2D version.
            BYTE *pData = nullptr;

            hr = m_pBuffer->Lock(&pData, nullptr, nullptr);
            if (SUCCEEDED(hr))
            {
                *plStride = lDefaultStride;
                if (lDefaultStride < 0)
                {
                    // Bottom-up orientation. Return a pointer to the start of the
                    // last row *in memory* which is the top row of the image.
                    *ppbScanLine0 = pData + abs(lDefaultStride) * (dwHeightInPixels - 1);
                }
                else
                {
                    // Top-down orientation. Return a pointer to the start of the
                    // buffer.
                    *ppbScanLine0 = pData;
                }
            }
        }
        return hr;
    }

    HRESULT UnlockBuffer()
    {
        if (m_p2DBuffer)
        {
            return m_p2DBuffer->Unlock2D();
        }
        else
        {
            return m_pBuffer->Unlock();
        }
    }

private:
    IMFMediaBuffer  *m_pBuffer;
    IMF2DBuffer     *m_p2DBuffer;
};
```

7. `GrayscaleTransform.h` isn't used in this example. You can remove it from the project if you want to.

8. Use the following code to replace the contents of `GrayscaleTransform.cpp`:

```
#include "pch.h"

#include "GrayscaleTransform_h.h"
#include "BufferLock.h"

using namespace Microsoft::WRL;

//
// * IMPORTANT: If you implement your own MFT, create a new GUID for the CLSID. *
//

// Configuration attributes

// {7BBBB051-133B-41F5-B6AA-5AFF9B33A2CB}
GUID const MFT_GRAYSCALE_DESTINATION_RECT = {0x7bbbb051, 0x133b, 0x41f5, 0xb6, 0xaa, 0x5a, 0xff,
0x9b, 0x33, 0xa2, 0xcb};

// {14782342-93E8-4565-872C-D9A2973D5CBF}
```

```
GUID const MFT_GRAYSCALE_SATURATION = {0x14782342, 0x93e8, 0x4565, 0x87, 0x2c, 0xd9, 0xa2, 0x97,
0x3d, 0x5c, 0xbf};

// {E0BADE5D-E4B9-4689-9DBA-E2F00D9CED0E}
GUID const MFT_GRAYSCALE_CHROMA_ROTATION = {0xe0bade5d, 0xe4b9, 0x4689, 0x9d, 0xba, 0xe2, 0xf0, 0xd,
0x9c, 0xed, 0xe};

template <class T> void SafeRelease(T **ppT)
{
    if (*ppT)
    {
        (*ppT)->Release();
        *ppT = nullptr;
    }
}

// Function pointer for the function that transforms the image.
typedef void (*IMAGE_TRANSFORM_FN)(
    const D2D1::Matrix3x2F& mat,             // Chroma transform matrix.
    const D2D_RECT_U&       rcDest,          // Destination rectangle for the transformation.
    BYTE*                   pDest,           // Destination buffer.
    LONG                    lDestStride,     // Destination stride.
    const BYTE*             pSrc,            // Source buffer.
    LONG                    lSrcStride,      // Source stride.
    DWORD                   dwWidthInPixels, // Image width in pixels.
    DWORD                   dwHeightInPixels // Image height in pixels.
    );

// Implements a grayscale video effect.
class CGrayscale
    : public RuntimeClass<
           RuntimeClassFlags<RuntimeClassType::WinRtClassicComMix>,
           ABI::Windows::Media::IMediaExtension,
           IMFTransform>
{
    InspectableClass(RuntimeClass_GrayscaleTransform_GrayscaleEffect, BaseTrust)

public:
    CGrayscale();

    STDMETHOD(RuntimeClassInitialize)();

    // IMediaExtension
    STDMETHODIMP SetProperties(ABI::Windows::Foundation::Collections::IPropertySet *pConfiguration);

    // IMFTransform
    STDMETHODIMP GetStreamLimits(
        DWORD   *pdwInputMinimum,
        DWORD   *pdwInputMaximum,
        DWORD   *pdwOutputMinimum,
        DWORD   *pdwOutputMaximum
    );

    STDMETHODIMP GetStreamCount(
        DWORD   *pcInputStreams,
        DWORD   *pcOutputStreams
    );

    STDMETHODIMP GetStreamIDs(
        DWORD   dwInputIDArraySize,
        DWORD   *pdwInputIDs,
        DWORD   dwOutputIDArraySize,
        DWORD   *pdwOutputIDs
    );

    STDMETHODIMP GetInputStreamInfo(
        DWORD                     dwInputStreamID,
        MFT_INPUT_STREAM_INFO *   pStreamInfo
    );
```

```
);

STDMETHODIMP GetOutputStreamInfo(
    DWORD                    dwOutputStreamID,
    MFT_OUTPUT_STREAM_INFO *  pStreamInfo
);

STDMETHODIMP GetAttributes(IMFAttributes** pAttributes);

STDMETHODIMP GetInputStreamAttributes(
    DWORD           dwInputStreamID,
    IMFAttributes   **ppAttributes
);

STDMETHODIMP GetOutputStreamAttributes(
    DWORD           dwOutputStreamID,
    IMFAttributes   **ppAttributes
);

STDMETHODIMP DeleteInputStream(DWORD dwStreamID);

STDMETHODIMP AddInputStreams(
    DWORD   cStreams,
    DWORD   *adwStreamIDs
);

STDMETHODIMP GetInputAvailableType(
    DWORD           dwInputStreamID,
    DWORD           dwTypeIndex, // 0-based
    IMFMediaType    **ppType
);

STDMETHODIMP GetOutputAvailableType(
    DWORD           dwOutputStreamID,
    DWORD           dwTypeIndex, // 0-based
    IMFMediaType    **ppType
);

STDMETHODIMP SetInputType(
    DWORD           dwInputStreamID,
    IMFMediaType    *pType,
    DWORD           dwFlags
);

STDMETHODIMP SetOutputType(
    DWORD           dwOutputStreamID,
    IMFMediaType    *pType,
    DWORD           dwFlags
);

STDMETHODIMP GetInputCurrentType(
    DWORD           dwInputStreamID,
    IMFMediaType    **ppType
);

STDMETHODIMP GetOutputCurrentType(
    DWORD           dwOutputStreamID,
    IMFMediaType    **ppType
);

STDMETHODIMP GetInputStatus(
    DWORD           dwInputStreamID,
    DWORD           *pdwFlags
);

STDMETHODIMP GetOutputStatus(DWORD *pdwFlags);

STDMETHODIMP SetOutputBounds(
    LONGLONG        hnsLowerBound,
    LONGLONG        hnsUpperBound
```

```
        LONGLONG        *pmfsUpperBound
    );

    STDMETHODIMP ProcessEvent(
        DWORD              dwInputStreamID,
        IMFMediaEvent      *pEvent
    );

    STDMETHODIMP ProcessMessage(
        MFT_MESSAGE_TYPE    eMessage,
        ULONG_PTR           ulParam
    );

    STDMETHODIMP ProcessInput(
        DWORD               dwInputStreamID,
        IMFSample           *pSample,
        DWORD               dwFlags
    );

    STDMETHODIMP ProcessOutput(
        DWORD                   dwFlags,
        DWORD                   cOutputBufferCount,
        MFT_OUTPUT_DATA_BUFFER  *pOutputSamples, // one per stream
        DWORD                   *pdwStatus
    );

private:
    ~CGrayscale();

    // HasPendingOutput: Returns TRUE if the MFT is holding an input sample.
    BOOL HasPendingOutput() const { return m_pSample != nullptr; }

    // IsValidInputStream: Returns TRUE if dwInputStreamID is a valid input stream identifier.
    BOOL IsValidInputStream(DWORD dwInputStreamID) const
    {
        return dwInputStreamID == 0;
    }

    // IsValidOutputStream: Returns TRUE if dwOutputStreamID is a valid output stream identifier.
    BOOL IsValidOutputStream(DWORD dwOutputStreamID) const
    {
        return dwOutputStreamID == 0;
    }

    HRESULT OnGetPartialType(DWORD dwTypeIndex, IMFMediaType **ppmt);
    HRESULT OnCheckInputType(IMFMediaType *pmt);
    HRESULT OnCheckOutputType(IMFMediaType *pmt);
    HRESULT OnCheckMediaType(IMFMediaType *pmt);
    void    OnSetInputType(IMFMediaType *pmt);
    void    OnSetOutputType(IMFMediaType *pmt);
    HRESULT BeginStreaming();
    HRESULT EndStreaming();
    HRESULT OnProcessOutput(IMFMediaBuffer *pIn, IMFMediaBuffer *pOut);
    HRESULT OnFlush();
    HRESULT UpdateFormatInfo();

    CRITICAL_SECTION            m_critSec;

    // Transformation parameters
    D2D1::Matrix3x2F            m_transform;                // Chroma transform matrix.
    D2D_RECT_U                  m_rcDest;                   // Destination rectangle for the effect.

    // Streaming
    bool                        m_bStreamingInitialized;
    IMFSample                   *m_pSample;                 // Input sample.
    IMFMediaType                *m_pInputType;              // Input media type.
    IMFMediaType                *m_pOutputType;             // Output media type.

    // Fomat information
    UINT32                      m_imageWidthInPixels;
```

```
    UINT32                      m_imageWidthInPixels;
    UINT32                      m_imageHeightInPixels;
    DWORD                       m_cbImageSize;              // Image size, in bytes.

    IMFAttributes               *m_pAttributes;

    // Image transform function. (Changes based on the media type.)
    IMAGE_TRANSFORM_FN          m_pTransformFn;
};
ActivatableClass(CGrayscale);

#pragma comment(lib, "d2d1")

/*

This sample implements a video effect as a Media Foundation transform (MFT).

The video effect manipulates chroma values in a YUV image. In the default setting,
the entire image is converted to grayscale. Optionally, the application may set any
of the following attributes:

MFT_GRAYSCALE_DESTINATION_RECT (type = blob, UINT32[4] array)

    Sets the destination rectangle for the effect. Pixels outside the destination
    rectangle are not altered.

MFT_GRAYSCALE_SATURATION (type = double)

    Sets the saturation level. The nominal range is [0...1]. Values beyond 1.0f
    result in supersaturated colors. Values below 0.0f create inverted colors.

MFT_GRAYSCALE_CHROMA_ROTATION (type = double)

    Rotates the chroma values of each pixel. The attribue value is the angle of
    rotation in degrees. The result is a shift in hue.

The effect is implemented by treating the chroma value of each pixel as a vector [u,v],
and applying a transformation matrix to the vector. The saturation parameter is applied
as a scaling transform.


NOTES ON THE MFT IMPLEMENTATION

1. The MFT has fixed streams: One input stream and one output stream.

2. The MFT supports the following formats: UYVY, YUY2, NV12.

3. If the MFT is holding an input sample, SetInputType and SetOutputType both fail.

4. The input and output types must be identical.

5. If both types are set, no type can be set until the current type is cleared.

6. Preferred input types:

   (a) If the output type is set, that's the preferred type.
   (b) Otherwise, the preferred types are partial types, constructed from the
         list of supported subtypes.

7. Preferred output types: As above.

8. Streaming:

    The private BeingStreaming() method is called in response to the
    MFT_MESSAGE_NOTIFY_BEGIN_STREAMING message.

    If the client does not send MFT_MESSAGE_NOTIFY_BEGIN_STREAMING, the MFT calls
    BeginStreaming inside the first call to ProcessInput or ProcessOutput.
```

```
    This is a good approach for allocating resources that your MFT requires for
    streaming.

9. The configuration attributes are applied in the BeginStreaming method. If the
   client changes the attributes during streaming, the change is ignored until
   streaming is stopped (either by changing the media types or by sending the
   MFT_MESSAGE_NOTIFY_END_STREAMING message) and then restarted.

*/


// Video FOURCC codes.
const DWORD FOURCC_YUY2 = '2YUY';
const DWORD FOURCC_UYVY = 'YVYU';
const DWORD FOURCC_NV12 = '21VN';

// Static array of media types (preferred and accepted).
const GUID g_MediaSubtypes[] =
{
    MFVideoFormat_NV12,
    MFVideoFormat_YUY2,
    MFVideoFormat_UYVY
};

HRESULT GetImageSize(DWORD fcc, UINT32 width, UINT32 height, DWORD* pcbImage);
HRESULT GetDefaultStride(IMFMediaType *pType, LONG *plStride);
bool ValidateRect(const RECT& rc);

template <typename T>
inline T clamp(const T& val, const T& minVal, const T& maxVal)
{
    return (val < minVal ? minVal : (val > maxVal ? maxVal : val));
}


// TransformChroma:
// Apply the transforms to calculate the output chroma values.

void TransformChroma(const D2D1::Matrix3x2F& mat, BYTE *pu, BYTE *pv)
{
    // Normalize the chroma values to [-112, 112] range

    D2D1_POINT_2F pt = { static_cast<float>(*pu) - 128, static_cast<float>(*pv) - 128 };

    pt = mat.TransformPoint(pt);

    // Clamp to valid range.
    clamp(pt.x, -112.0f, 112.0f);
    clamp(pt.y, -112.0f, 112.0f);

    // Map back to [16...240] range.
    *pu = static_cast<BYTE>(pt.x + 128.0f);
    *pv = static_cast<BYTE>(pt.y + 128.0f);
}

//-------------------------------------------------------------------
// Functions to convert a YUV images to grayscale.
//
// In all cases, the same transformation is applied to the 8-bit
// chroma values, but the pixel layout in memory differs.
//
// The image conversion functions take the following parameters:
//
// mat               Transfomation matrix for chroma values.
// rcDest            Destination rectangle.
// pDest             Pointer to the destination buffer.
// lDestStride       Stride of the destination buffer, in bytes.
// pSrc              Pointer to the source buffer.
// lSrcStride        Stride of the source buffer, in bytes.
```

```
// dwWidthInPixels   Frame width in pixels.
// dwHeightInPixels  Frame height, in pixels.
//-------------------------------------------------------------------

// Convert UYVY image.

void TransformImage_UYVY(
    const D2D1::Matrix3x2F& mat,
    const D2D_RECT_U& rcDest,
    _Inout_updates_(_Inexpressible_(lDestStride * dwHeightInPixels)) BYTE *pDest,
    _In_ LONG lDestStride,
    _In_reads_(_Inexpressible_(lSrcStride * dwHeightInPixels)) const BYTE* pSrc,
    _In_ LONG lSrcStride,
    _In_ DWORD dwWidthInPixels,
    _In_ DWORD dwHeightInPixels)
{
    DWORD y = 0;
    const DWORD y0 = min(rcDest.bottom, dwHeightInPixels);

    // Lines above the destination rectangle.
    for ( ; y < rcDest.top; y++)
    {
        memcpy(pDest, pSrc, dwWidthInPixels * 2);
        pSrc += lSrcStride;
        pDest += lDestStride;
    }

    // Lines within the destination rectangle.
    for ( ; y < y0; y++)
    {
        WORD *pSrc_Pixel = (WORD*)pSrc;
        WORD *pDest_Pixel = (WORD*)pDest;

        for (DWORD x = 0; (x + 1) < dwWidthInPixels; x += 2)
        {
            // Byte order is U0 Y0 V0 Y1
            // Each WORD is a byte pair (U/V, Y)
            // Windows is little-endian so the order appears reversed.

            if (x >= rcDest.left && x < rcDest.right)
            {
                BYTE u = pSrc_Pixel[x] & 0x00FF;
                BYTE v = pSrc_Pixel[x+1] & 0x00FF;

                TransformChroma(mat, &u, &v);

                pDest_Pixel[x] = (pSrc_Pixel[x] & 0xFF00) | u;
                pDest_Pixel[x+1] = (pSrc_Pixel[x+1] & 0xFF00) | v;
            }
            else
            {
#pragma warning(push)
#pragma warning(disable: 6385)
#pragma warning(disable: 6386)
                pDest_Pixel[x] = pSrc_Pixel[x];
                pDest_Pixel[x+1] = pSrc_Pixel[x+1];
#pragma warning(pop)
            }
        }

        pDest += lDestStride;
        pSrc += lSrcStride;
    }

    // Lines below the destination rectangle.
    for ( ; y < dwHeightInPixels; y++)
    {
        memcpy(pDest, pSrc, dwWidthInPixels * 2);
        pSrc += lSrcStride;
```

```
        pDest += lDestStride;
    }
}


// Convert YUY2 image.

void TransformImage_YUY2(
    const D2D1::Matrix3x2F& mat,
    const D2D_RECT_U& rcDest,
    _Inout_updates_(_Inexpressible_(lDestStride * dwHeightInPixels)) BYTE *pDest,
    _In_ LONG lDestStride,
    _In_reads_(_Inexpressible_(lSrcStride * dwHeightInPixels)) const BYTE* pSrc,
    _In_ LONG lSrcStride,
    _In_ DWORD dwWidthInPixels,
    _In_ DWORD dwHeightInPixels)
{
    DWORD y = 0;
    const DWORD y0 = min(rcDest.bottom, dwHeightInPixels);

    // Lines above the destination rectangle.
    for ( ; y < rcDest.top; y++)
    {
        memcpy(pDest, pSrc, dwWidthInPixels * 2);
        pSrc += lSrcStride;
        pDest += lDestStride;
    }

    // Lines within the destination rectangle.
    for ( ; y < y0; y++)
    {
        WORD *pSrc_Pixel = (WORD*)pSrc;
        WORD *pDest_Pixel = (WORD*)pDest;

        for (DWORD x = 0; (x + 1) < dwWidthInPixels; x += 2)
        {
            // Byte order is Y0 U0 Y1 V0
            // Each WORD is a byte pair (Y, U/V)
            // Windows is little-endian so the order appears reversed.

            if (x >= rcDest.left && x < rcDest.right)
            {
                BYTE u = pSrc_Pixel[x] >> 8;
                BYTE v = pSrc_Pixel[x+1] >> 8;

                TransformChroma(mat, &u, &v);

                pDest_Pixel[x] = (pSrc_Pixel[x] & 0x00FF) | (u<<8);
                pDest_Pixel[x+1] = (pSrc_Pixel[x+1] & 0x00FF) | (v<<8);
            }
            else
            {
#pragma warning(push)
#pragma warning(disable: 6385)
#pragma warning(disable: 6386)
                pDest_Pixel[x] = pSrc_Pixel[x];
                pDest_Pixel[x+1] = pSrc_Pixel[x+1];
#pragma warning(pop)
            }
        }
        pDest += lDestStride;
        pSrc += lSrcStride;
    }

    // Lines below the destination rectangle.
    for ( ; y < dwHeightInPixels; y++)
    {
        memcpy(pDest, pSrc, dwWidthInPixels * 2);
        pSrc += lSrcStride;
```

```
        pDest += lDestStride;
    }
}

// Convert NV12 image

void TransformImage_NV12(
    const D2D1::Matrix3x2F& mat,
    const D2D_RECT_U& rcDest,
    _Inout_updates_(_Inexpressible_(2 * lDestStride * dwHeightInPixels)) BYTE *pDest,
    _In_ LONG lDestStride,
    _In_reads_(_Inexpressible_(2 * lSrcStride * dwHeightInPixels)) const BYTE* pSrc,
    _In_ LONG lSrcStride,
    _In_ DWORD dwWidthInPixels,
    _In_ DWORD dwHeightInPixels)
{
    // NV12 is planar: Y plane, followed by packed U-V plane.

    // Y plane
    for (DWORD y = 0; y < dwHeightInPixels; y++)
    {
        CopyMemory(pDest, pSrc, dwWidthInPixels);
        pDest += lDestStride;
        pSrc += lSrcStride;
    }

    // U-V plane

    // NOTE: The U-V plane has 1/2 the number of lines as the Y plane.

    // Lines above the destination rectangle.
    DWORD y = 0;
    const DWORD y0 = min(rcDest.bottom, dwHeightInPixels);

    for ( ; y < rcDest.top/2; y++)
    {
        memcpy(pDest, pSrc, dwWidthInPixels);
        pSrc += lSrcStride;
        pDest += lDestStride;
    }

    // Lines within the destination rectangle.
    for ( ; y < y0/2; y++)
    {
        for (DWORD x = 0; (x + 1) < dwWidthInPixels; x += 2)
        {
            if (x >= rcDest.left && x < rcDest.right)
            {
                BYTE u = pSrc[x];
                BYTE v = pSrc[x+1];

                TransformChroma(mat, &u, &v);

                pDest[x] = u;
                pDest[x+1] = v;
            }
            else
            {
                pDest[x] = pSrc[x];
                pDest[x+1] = pSrc[x+1];
            }
        }
        pDest += lDestStride;
        pSrc += lSrcStride;
    }

    // Lines below the destination rectangle.
    for ( ; y < dwHeightInPixels/2; y++)
    {
```

```
    {
        memcpy(pDest, pSrc, dwWidthInPixels);
        pSrc += lSrcStride;
        pDest += lDestStride;
    }
}

CGrayscale::CGrayscale() :
    m_pSample(nullptr), m_pInputType(nullptr), m_pOutputType(nullptr), m_pTransformFn(nullptr),
    m_imageWidthInPixels(0), m_imageHeightInPixels(0), m_cbImageSize(0),
    m_transform(D2D1::Matrix3x2F::Identity()), m_rcDest(D2D1::RectU()),
m_bStreamingInitialized(false),
    m_pAttributes(nullptr)
{
    InitializeCriticalSectionEx(&m_critSec, 3000, 0);
}

CGrayscale::~CGrayscale()
{
    SafeRelease(&m_pInputType);
    SafeRelease(&m_pOutputType);
    SafeRelease(&m_pSample);
    SafeRelease(&m_pAttributes);
    DeleteCriticalSection(&m_critSec);
}

// Initialize the instance.
STDMETHODIMP CGrayscale::RuntimeClassInitialize()
{
    // Create the attribute store.
    return MFCreateAttributes(&m_pAttributes, 3);
}

// IMediaExtension methods

//-------------------------------------------------------------------
// SetProperties
// Sets the configuration of the effect
//-------------------------------------------------------------------
HRESULT CGrayscale::SetProperties(ABI::Windows::Foundation::Collections::IPropertySet
*pConfiguration)
{
    return S_OK;
}

// IMFTransform methods. Refer to the Media Foundation SDK documentation for details.

//-------------------------------------------------------------------
// GetStreamLimits
// Returns the minimum and maximum number of streams.
//-------------------------------------------------------------------

HRESULT CGrayscale::GetStreamLimits(
    DWORD   *pdwInputMinimum,
    DWORD   *pdwInputMaximum,
    DWORD   *pdwOutputMinimum,
    DWORD   *pdwOutputMaximum
)
{
    // This MFT has a fixed number of streams.
    *pdwInputMinimum = 1;
    *pdwInputMaximum = 1;
    *pdwOutputMinimum = 1;
    *pdwOutputMaximum = 1;
    return S_OK;
}

// Returns the actual number of streams.

HRESULT CGrayscale::GetStreamCount(
```

```
HRESULT CGrayscale::GetStreamCount(
    DWORD   *pcInputStreams,
    DWORD   *pcOutputStreams
)
{
    // This MFT has a fixed number of streams.
    *pcInputStreams = 1;
    *pcOutputStreams = 1;
    return S_OK;
}



//------------------------------------------------------------------
// GetStreamIDs
// Returns stream IDs for the input and output streams.
//------------------------------------------------------------------

HRESULT CGrayscale::GetStreamIDs(
    DWORD   dwInputIDArraySize,
    DWORD   *pdwInputIDs,
    DWORD   dwOutputIDArraySize,
    DWORD   *pdwOutputIDs
)
{
    // It is not required to implement this method if the MFT has a fixed number of
    // streams AND the stream IDs are numbered sequentially from zero (that is, the
    // stream IDs match the stream indexes).

    // In that case, it is OK to return E_NOTIMPL.
    return E_NOTIMPL;
}


//------------------------------------------------------------------
// GetInputStreamInfo
// Returns information about an input stream.
//------------------------------------------------------------------

HRESULT CGrayscale::GetInputStreamInfo(
    DWORD                     dwInputStreamID,
    MFT_INPUT_STREAM_INFO *   pStreamInfo
)
{
    EnterCriticalSection(&m_critSec);

    if (!IsValidInputStream(dwInputStreamID))
    {
        LeaveCriticalSection(&m_critSec);
        return MF_E_INVALIDSTREAMNUMBER;
    }

    // NOTE: This method should succeed even when there is no media type on the
    //       stream. If there is no media type, we only need to fill in the dwFlags
    //       member of MFT_INPUT_STREAM_INFO. The other members depend on having a
    //       a valid media type.

    pStreamInfo->hnsMaxLatency = 0;
    pStreamInfo->dwFlags = MFT_INPUT_STREAM_WHOLE_SAMPLES |
MFT_INPUT_STREAM_SINGLE_SAMPLE_PER_BUFFER;

    if (m_pInputType == nullptr)
    {
        pStreamInfo->cbSize = 0;
    }
    else
    {
        pStreamInfo->cbSize = m_cbImageSize;
    }
```

```
    pStreamInfo->cbMaxLookahead = 0;
    pStreamInfo->cbAlignment = 0;

    LeaveCriticalSection(&m_critSec);
    return S_OK;
}

//-------------------------------------------------------------------
// GetOutputStreamInfo
// Returns information about an output stream.
//-------------------------------------------------------------------

HRESULT CGrayscale::GetOutputStreamInfo(
    DWORD                    dwOutputStreamID,
    MFT_OUTPUT_STREAM_INFO *  pStreamInfo
)
{
    EnterCriticalSection(&m_critSec);

    if (!IsValidOutputStream(dwOutputStreamID))
    {
        LeaveCriticalSection(&m_critSec);
        return MF_E_INVALIDSTREAMNUMBER;
    }

    // NOTE: This method should succeed even when there is no media type on the
    //       stream. If there is no media type, we only need to fill in the dwFlags
    //       member of MFT_OUTPUT_STREAM_INFO. The other members depend on having a
    //       a valid media type.

    pStreamInfo->dwFlags =
        MFT_OUTPUT_STREAM_WHOLE_SAMPLES |
        MFT_OUTPUT_STREAM_SINGLE_SAMPLE_PER_BUFFER |
        MFT_OUTPUT_STREAM_FIXED_SAMPLE_SIZE ;

    if (m_pOutputType == nullptr)
    {
        pStreamInfo->cbSize = 0;
    }
    else
    {
        pStreamInfo->cbSize = m_cbImageSize;
    }

    pStreamInfo->cbAlignment = 0;

    LeaveCriticalSection(&m_critSec);
    return S_OK;
}

// Returns the attributes for the MFT.
HRESULT CGrayscale::GetAttributes(IMFAttributes** ppAttributes)
{
    EnterCriticalSection(&m_critSec);

    *ppAttributes = m_pAttributes;
    (*ppAttributes)->AddRef();

    LeaveCriticalSection(&m_critSec);
    return S_OK;
}

// Returns stream-level attributes for an input stream.

HRESULT CGrayscale::GetInputStreamAttributes(
    DWORD           dwInputStreamID,
    IMFAttributes   **ppAttributes
)
```

```
{
    // This MFT does not support any stream-level attributes, so the method is not implemented.
    return E_NOTIMPL;
}


//-------------------------------------------------------------------
// GetOutputStreamAttributes
// Returns stream-level attributes for an output stream.
//-------------------------------------------------------------------

HRESULT CGrayscale::GetOutputStreamAttributes(
    DWORD           dwOutputStreamID,
    IMFAttributes   **ppAttributes
)
{
    // This MFT does not support any stream-level attributes, so the method is not implemented.
    return E_NOTIMPL;
}


//-------------------------------------------------------------------
// DeleteInputStream
//-------------------------------------------------------------------

HRESULT CGrayscale::DeleteInputStream(DWORD dwStreamID)
{
    // This MFT has a fixed number of input streams, so the method is not supported.
    return E_NOTIMPL;
}


//-------------------------------------------------------------------
// AddInputStreams
//-------------------------------------------------------------------

HRESULT CGrayscale::AddInputStreams(
    DWORD   cStreams,
    DWORD   *adwStreamIDs
)
{
    // This MFT has a fixed number of output streams, so the method is not supported.
    return E_NOTIMPL;
}


//-------------------------------------------------------------------
// GetInputAvailableType
// Returns a preferred input type.
//-------------------------------------------------------------------

HRESULT CGrayscale::GetInputAvailableType(
    DWORD           dwInputStreamID,
    DWORD           dwTypeIndex, // 0-based
    IMFMediaType    **ppType
)
{
    EnterCriticalSection(&m_critSec);

    if (!IsValidInputStream(dwInputStreamID))
    {
        LeaveCriticalSection(&m_critSec);
        return MF_E_INVALIDSTREAMNUMBER;
    }

    HRESULT hr = S_OK;

    // If the output type is set, return that type as our preferred input type.
    if (m_pOutputType == nullptr)
```

```
    {
        // The output type is not set. Create a partial media type.
        hr = OnGetPartialType(dwTypeIndex, ppType);
    }
    else if (dwTypeIndex > 0)
    {
        hr = MF_E_NO_MORE_TYPES;
    }
    else
    {
        *ppType = m_pOutputType;
        (*ppType)->AddRef();
    }

    LeaveCriticalSection(&m_critSec);
    return hr;
}

// Returns a preferred output type.

HRESULT CGrayscale::GetOutputAvailableType(
    DWORD           dwOutputStreamID,
    DWORD           dwTypeIndex, // 0-based
    IMFMediaType    **ppType
)
{
    EnterCriticalSection(&m_critSec);

    if (!IsValidOutputStream(dwOutputStreamID))
    {
        LeaveCriticalSection(&m_critSec);
        return MF_E_INVALIDSTREAMNUMBER;
    }

    HRESULT hr = S_OK;

    if (m_pInputType == nullptr)
    {
        // The input type is not set. Create a partial media type.
        hr = OnGetPartialType(dwTypeIndex, ppType);
    }
    else if (dwTypeIndex > 0)
    {
        hr = MF_E_NO_MORE_TYPES;
    }
    else
    {
        *ppType = m_pInputType;
        (*ppType)->AddRef();
    }

    LeaveCriticalSection(&m_critSec);
    return hr;
}

HRESULT CGrayscale::SetInputType(
    DWORD           dwInputStreamID,
    IMFMediaType    *pType, // Can be nullptr to clear the input type.
    DWORD           dwFlags
)
{
    // Validate flags.
    if (dwFlags & ~MFT_SET_TYPE_TEST_ONLY)
    {
        return E_INVALIDARG;
    }

    EnterCriticalSection(&m_critSec);
```

```
    if (!IsValidInputStream(dwInputStreamID))
    {
        LeaveCriticalSection(&m_critSec);
        return MF_E_INVALIDSTREAMNUMBER;
    }

    HRESULT hr = S_OK;

    // Does the caller want us to set the type, or just test it?
    BOOL bReallySet = ((dwFlags & MFT_SET_TYPE_TEST_ONLY) == 0);

    // If we have an input sample, the client cannot change the type now.
    if (HasPendingOutput())
    {
        hr = MF_E_TRANSFORM_CANNOT_CHANGE_MEDIATYPE_WHILE_PROCESSING;
        goto done;
    }

    // Validate the type, if non-nullptr.
    if (pType)
    {
        hr = OnCheckInputType(pType);
        if (FAILED(hr))
        {
            goto done;
        }
    }

    // The type is OK. Set the type, unless the caller was just testing.
    if (bReallySet)
    {
        OnSetInputType(pType);

        // When the type changes, end streaming.
        hr = EndStreaming();
    }

done:
    LeaveCriticalSection(&m_critSec);
    return hr;
}

HRESULT CGrayscale::SetOutputType(
    DWORD           dwOutputStreamID,
    IMFMediaType    *pType, // Can be nullptr to clear the output type.
    DWORD           dwFlags
)
{
    // Validate flags.
    if (dwFlags & ~MFT_SET_TYPE_TEST_ONLY)
    {
        return E_INVALIDARG;
    }

    EnterCriticalSection(&m_critSec);

    if (!IsValidOutputStream(dwOutputStreamID))
    {
        LeaveCriticalSection(&m_critSec);
        return MF_E_INVALIDSTREAMNUMBER;
    }

    HRESULT hr = S_OK;

    // Does the caller want us to set the type, or just test it?
    BOOL bReallySet = ((dwFlags & MFT_SET_TYPE_TEST_ONLY) == 0);

    // If we have an input sample, the client cannot change the type now.
    if (HasPendingOutput())
```

```
    if (HasPendingOutput())
    {
        hr = MF_E_TRANSFORM_CANNOT_CHANGE_MEDIATYPE_WHILE_PROCESSING;
        goto done;
    }

    // Validate the type, if non-nullptr.
    if (pType)
    {
        hr = OnCheckOutputType(pType);
        if (FAILED(hr))
        {
            goto done;
        }
    }

    // The type is OK. Set the type, unless the caller was just testing.
    if (bReallySet)
    {
        OnSetOutputType(pType);

        // When the type changes, end streaming.
        hr = EndStreaming();
    }

done:
    LeaveCriticalSection(&m_critSec);
    return hr;
}

// Returns the current input type.

HRESULT CGrayscale::GetInputCurrentType(
    DWORD           dwInputStreamID,
    IMFMediaType    **ppType
)
{
    HRESULT hr = S_OK;

    EnterCriticalSection(&m_critSec);

    if (!IsValidInputStream(dwInputStreamID))
    {
        hr = MF_E_INVALIDSTREAMNUMBER;
    }
    else if (!m_pInputType)
    {
        hr = MF_E_TRANSFORM_TYPE_NOT_SET;
    }
    else
    {
        *ppType = m_pInputType;
        (*ppType)->AddRef();
    }
    LeaveCriticalSection(&m_critSec);
    return hr;
}

// Returns the current output type.

HRESULT CGrayscale::GetOutputCurrentType(
    DWORD           dwOutputStreamID,
    IMFMediaType    **ppType
)
{
    HRESULT hr = S_OK;

    EnterCriticalSection(&m_critSec);

    if (!IsValidOutputStream(dwOutputStreamID))
```

```
    if (!IsValidOutputStream(dwOutputStreamID))
    {
        hr = MF_E_INVALIDSTREAMNUMBER;
    }
    else if (!m_pOutputType)
    {
        hr = MF_E_TRANSFORM_TYPE_NOT_SET;
    }
    else
    {
        *ppType = m_pOutputType;
        (*ppType)->AddRef();
    }

    LeaveCriticalSection(&m_critSec);
    return hr;
}

// Query if the MFT is accepting more input.

HRESULT CGrayscale::GetInputStatus(
    DWORD           dwInputStreamID,
    DWORD           *pdwFlags
)
{
    EnterCriticalSection(&m_critSec);

    if (!IsValidInputStream(dwInputStreamID))
    {
        LeaveCriticalSection(&m_critSec);
        return MF_E_INVALIDSTREAMNUMBER;
    }

    // If an input sample is already queued, do not accept another sample until the
    // client calls ProcessOutput or Flush.

    // NOTE: It is possible for an MFT to accept more than one input sample. For
    // example, this might be required in a video decoder if the frames do not
    // arrive in temporal order. In the case, the decoder must hold a queue of
    // samples. For the video effect, each sample is transformed independently, so
    // there is no reason to queue multiple input samples.

    if (m_pSample == nullptr)
    {
        *pdwFlags = MFT_INPUT_STATUS_ACCEPT_DATA;
    }
    else
    {
        *pdwFlags = 0;
    }

    LeaveCriticalSection(&m_critSec);
    return S_OK;
}

// Query if the MFT can produce output.

HRESULT CGrayscale::GetOutputStatus(DWORD *pdwFlags)
{
    EnterCriticalSection(&m_critSec);

    // The MFT can produce an output sample if (and only if) there an input sample.
    if (m_pSample != nullptr)
    {
        *pdwFlags = MFT_OUTPUT_STATUS_SAMPLE_READY;
    }
    else
    {
        *pdwFlags = 0;
    }
```

```
    }

    LeaveCriticalSection(&m_critSec);
    return S_OK;
}


//-------------------------------------------------------------------
// SetOutputBounds
// Sets the range of time stamps that the MFT will output.
//-------------------------------------------------------------------

HRESULT CGrayscale::SetOutputBounds(
    LONGLONG        hnsLowerBound,
    LONGLONG        hnsUpperBound
)
{
    // Implementation of this method is optional.
    return E_NOTIMPL;
}


//-------------------------------------------------------------------
// ProcessEvent
// Sends an event to an input stream.
//-------------------------------------------------------------------

HRESULT CGrayscale::ProcessEvent(
    DWORD              dwInputStreamID,
    IMFMediaEvent      *pEvent
)
{
    // This MFT does not handle any stream events, so the method can
    // return E_NOTIMPL. This tells the pipeline that it can stop
    // sending any more events to this MFT.
    return E_NOTIMPL;
}


//-------------------------------------------------------------------
// ProcessMessage
//-------------------------------------------------------------------

HRESULT CGrayscale::ProcessMessage(
    MFT_MESSAGE_TYPE    eMessage,
    ULONG_PTR           ulParam
)
{
    EnterCriticalSection(&m_critSec);

    HRESULT hr = S_OK;

    switch (eMessage)
    {
    case MFT_MESSAGE_COMMAND_FLUSH:
        // Flush the MFT.
        hr = OnFlush();
        break;

    case MFT_MESSAGE_COMMAND_DRAIN:
        // Drain: Tells the MFT to reject further input until all pending samples are
        // processed. That is our default behavior already, so there is nothing to do.
        //
        // For a decoder that accepts a queue of samples, the MFT might need to drain
        // the queue in response to this command.
    break;

    case MFT_MESSAGE_SET_D3D_MANAGER:
        // Sets a pointer to the IDirect3DDeviceManager9 interface.
```

```
        // The pipeline should never send this message unless the MFT sets the MF_SA_D3D_AWARE
        // attribute set to TRUE. Because this MFT does not set MF_SA_D3D_AWARE, it is an error
        // to send the MFT_MESSAGE_SET_D3D_MANAGER message to the MFT. Return an error code in
        // this case.

        // NOTE: If this MFT were D3D-enabled, it would cache the IDirect3DDeviceManager9
        // pointer for use during streaming.

        hr = E_NOTIMPL;
        break;

    case MFT_MESSAGE_NOTIFY_BEGIN_STREAMING:
        hr = BeginStreaming();
        break;

    case MFT_MESSAGE_NOTIFY_END_STREAMING:
        hr = EndStreaming();
        break;

    // The next two messages do not require any action from this MFT.

    case MFT_MESSAGE_NOTIFY_END_OF_STREAM:
        break;

    case MFT_MESSAGE_NOTIFY_START_OF_STREAM:
        break;
    }

    LeaveCriticalSection(&m_critSec);
    return hr;
}

// Process an input sample.

HRESULT CGrayscale::ProcessInput(
    DWORD               dwInputStreamID,
    IMFSample           *pSample,
    DWORD               dwFlags
)
{
    if (dwFlags != 0)
    {
        return E_INVALIDARG; // dwFlags is reserved and must be zero.
    }

    HRESULT hr = S_OK;

    EnterCriticalSection(&m_critSec);

    // Validate the input stream number.
    if (!IsValidInputStream(dwInputStreamID))
    {
        hr = MF_E_INVALIDSTREAMNUMBER;
        goto done;
    }

    // Check for valid media types.
    // The client must set input and output types before calling ProcessInput.
    if (!m_pInputType || !m_pOutputType)
    {
        hr = MF_E_NOTACCEPTING;
        goto done;
    }

    // Check if an input sample is already queued.
    if (m_pSample != nullptr)
    {
        hr = MF_E_NOTACCEPTING;   // We already have an input sample.
```

```
        goto done;
    }

    // Initialize streaming.
    hr = BeginStreaming();
    if (FAILED(hr))
    {
        goto done;
    }

    // Cache the sample. We do the actual work in ProcessOutput.
    m_pSample = pSample;
    pSample->AddRef();  // Hold a reference count on the sample.

done:
    LeaveCriticalSection(&m_critSec);
    return hr;
}


//------------------------------------------------------------------
// ProcessOutput
// Process an output sample.
//------------------------------------------------------------------

HRESULT CGrayscale::ProcessOutput(
    DWORD                   dwFlags,
    DWORD                   cOutputBufferCount,
    MFT_OUTPUT_DATA_BUFFER  *pOutputSamples, // one per stream
    DWORD                   *pdwStatus
)
{
    // Check input parameters...

    // This MFT does not accept any flags for the dwFlags parameter.

    // The only defined flag is MFT_PROCESS_OUTPUT_DISCARD_WHEN_NO_BUFFER. This flag
    // applies only when the MFT marks an output stream as lazy or optional. But this
    // MFT has no lazy or optional streams, so the flag is not valid.

    if (dwFlags != 0)
    {
        return E_INVALIDARG;
    }

    // There must be exactly one output buffer.
    if (cOutputBufferCount != 1)
    {
        return E_INVALIDARG;
    }

    // It must contain a sample.
    if (pOutputSamples[0].pSample == nullptr)
    {
        return E_INVALIDARG;
    }

    HRESULT hr = S_OK;

    IMFMediaBuffer *pInput = nullptr;
    IMFMediaBuffer *pOutput = nullptr;

    EnterCriticalSection(&m_critSec);

    // There must be an input sample available for processing.
    if (m_pSample == nullptr)
    {
        hr = MF_E_TRANSFORM_NEED_MORE_INPUT;
        goto done;
```

```
    }

    // Initialize streaming.

    hr = BeginStreaming();
    if (FAILED(hr))
    {
        goto done;
    }

    // Get the input buffer.
    hr = m_pSample->ConvertToContiguousBuffer(&pInput);
    if (FAILED(hr))
    {
        goto done;
    }

    // Get the output buffer.
    hr = pOutputSamples[0].pSample->ConvertToContiguousBuffer(&pOutput);
    if (FAILED(hr))
    {
        goto done;
    }

    hr = OnProcessOutput(pInput, pOutput);
    if (FAILED(hr))
    {
        goto done;
    }

    // Set status flags.
    pOutputSamples[0].dwStatus = 0;
    *pdwStatus = 0;


    // Copy the duration and time stamp from the input sample, if present.

    LONGLONG hnsDuration = 0;
    LONGLONG hnsTime = 0;

    if (SUCCEEDED(m_pSample->GetSampleDuration(&hnsDuration)))
    {
        hr = pOutputSamples[0].pSample->SetSampleDuration(hnsDuration);
        if (FAILED(hr))
        {
            goto done;
        }
    }

    if (SUCCEEDED(m_pSample->GetSampleTime(&hnsTime)))
    {
        hr = pOutputSamples[0].pSample->SetSampleTime(hnsTime);
    }

done:
    SafeRelease(&m_pSample);   // Release our input sample.
    SafeRelease(&pInput);
    SafeRelease(&pOutput);
    LeaveCriticalSection(&m_critSec);
    return hr;
}

// PRIVATE METHODS

// All methods that follow are private to this MFT and are not part of the IMFTransform interface.

// Create a partial media type from our list.
//
// dwTypeIndex: Index into the list of peferred media types.
```

```
// dwTypeIndex: Index into the list of preferred media types.
// ppmt:        Receives a pointer to the media type.

HRESULT CGrayscale::OnGetPartialType(DWORD dwTypeIndex, IMFMediaType **ppmt)
{
    if (dwTypeIndex >= ARRAYSIZE(g_MediaSubtypes))
    {
        return MF_E_NO_MORE_TYPES;
    }

    IMFMediaType *pmt = nullptr;

    HRESULT hr = MFCreateMediaType(&pmt);
    if (FAILED(hr))
    {
        goto done;
    }

    hr = pmt->SetGUID(MF_MT_MAJOR_TYPE, MFMediaType_Video);
    if (FAILED(hr))
    {
        goto done;
    }

    hr = pmt->SetGUID(MF_MT_SUBTYPE, g_MediaSubtypes[dwTypeIndex]);
    if (FAILED(hr))
    {
        goto done;
    }

    *ppmt = pmt;
    (*ppmt)->AddRef();

done:
    SafeRelease(&pmt);
    return hr;
}

// Validate an input media type.

HRESULT CGrayscale::OnCheckInputType(IMFMediaType *pmt)
{
    assert(pmt != nullptr);

    HRESULT hr = S_OK;

    // If the output type is set, see if they match.
    if (m_pOutputType != nullptr)
    {
        DWORD flags = 0;
        hr = pmt->IsEqual(m_pOutputType, &flags);

        // IsEqual can return S_FALSE. Treat this as failure.
        if (hr != S_OK)
        {
            hr = MF_E_INVALIDMEDIATYPE;
        }
    }
    else
    {
        // Output type is not set. Just check this type.
        hr = OnCheckMediaType(pmt);
    }
    return hr;
}

// Validate an output media type.

HRESULT CGrayscale::OnCheckOutputType(IMFMediaType *pmt)
{
```

```
{
    assert(pmt != nullptr);

    HRESULT hr = S_OK;

    // If the input type is set, see if they match.
    if (m_pInputType != nullptr)
    {
        DWORD flags = 0;
        hr = pmt->IsEqual(m_pInputType, &flags);

        // IsEqual can return S_FALSE. Treat this as failure.
        if (hr != S_OK)
        {
            hr = MF_E_INVALIDMEDIATYPE;
        }

    }
    else
    {
        // Input type is not set. Just check this type.
        hr = OnCheckMediaType(pmt);
    }
    return hr;
}


// Validate a media type (input or output)

HRESULT CGrayscale::OnCheckMediaType(IMFMediaType *pmt)
{
    BOOL bFoundMatchingSubtype = FALSE;

    // Major type must be video.
    GUID major_type;
    HRESULT hr = pmt->GetGUID(MF_MT_MAJOR_TYPE, &major_type);
    if (FAILED(hr))
    {
        goto done;
    }

    if (major_type != MFMediaType_Video)
    {
        hr = MF_E_INVALIDMEDIATYPE;
        goto done;
    }

    // Subtype must be one of the subtypes in our global list.

    // Get the subtype GUID.
    GUID subtype;
    hr = pmt->GetGUID(MF_MT_SUBTYPE, &subtype);
    if (FAILED(hr))
    {
        goto done;
    }

    // Look for the subtype in our list of accepted types.
    for (DWORD i = 0; i < ARRAYSIZE(g_MediaSubtypes); i++)
    {
        if (subtype == g_MediaSubtypes[i])
        {
            bFoundMatchingSubtype = TRUE;
            break;
        }
    }

    if (!bFoundMatchingSubtype)
    {
        hr = MF_E_INVALIDMEDIATYPE; // The MFT does not support this subtype.
```

```
        hr = MF_E_INVALIDMEDIATYPE; // The MFT does not support this subtype.
        goto done;
    }

    // Reject single-field media types.
    UINT32 interlace = MFGetAttributeUINT32(pmt, MF_MT_INTERLACE_MODE, MFVideoInterlace_Progressive);
    if (interlace == MFVideoInterlace_FieldSingleUpper  || interlace ==
MFVideoInterlace_FieldSingleLower)
    {
        hr = MF_E_INVALIDMEDIATYPE;
    }

done:
    return hr;
}


// Set or clear the input media type.
//
// Prerequisite: The input type was already validated.

void CGrayscale::OnSetInputType(IMFMediaType *pmt)
{
    // if pmt is nullptr, clear the type.
    // if pmt is non-nullptr, set the type.

    SafeRelease(&m_pInputType);
    m_pInputType = pmt;
    if (m_pInputType)
    {
        m_pInputType->AddRef();
    }

    // Update the format information.
    UpdateFormatInfo();
}


// Set or clears the output media type.
//
// Prerequisite: The output type was already validated.

void CGrayscale::OnSetOutputType(IMFMediaType *pmt)
{
    // If pmt is nullptr, clear the type. Otherwise, set the type.

    SafeRelease(&m_pOutputType);
    m_pOutputType = pmt;
    if (m_pOutputType)
    {
        m_pOutputType->AddRef();
    }
}


// Initialize streaming parameters.
//
// This method is called if the client sends the MFT_MESSAGE_NOTIFY_BEGIN_STREAMING
// message, or when the client processes a sample, whichever happens first.

HRESULT CGrayscale::BeginStreaming()
{
    HRESULT hr = S_OK;

    if (!m_bStreamingInitialized)
    {
        // Get the configuration attributes.

        // Get the destination rectangle.
```

```
        RECT rcDest;
        hr = m_pAttributes->GetBlob(MFT_GRAYSCALE_DESTINATION_RECT, (UINT8*)&rcDest, sizeof(rcDest),
nullptr);
        if (hr == MF_E_ATTRIBUTENOTFOUND || !ValidateRect(rcDest))
        {
            // The client did not set this attribute, or the client provided an invalid rectangle.
            // Default to the entire image.

            m_rcDest = D2D1::RectU(0, 0, m_imageWidthInPixels, m_imageHeightInPixels);
            hr = S_OK;
        }
        else if (SUCCEEDED(hr))
        {
            m_rcDest = D2D1::RectU(rcDest.left, rcDest.top, rcDest.right, rcDest.bottom);
        }
        else
        {
            goto done;
        }

        // Get the chroma transformations.

        float scale = (float)MFGetAttributeDouble(m_pAttributes, MFT_GRAYSCALE_SATURATION, 0.0f);
        float angle = (float)MFGetAttributeDouble(m_pAttributes, MFT_GRAYSCALE_CHROMA_ROTATION,
0.0f);

        m_transform = D2D1::Matrix3x2F::Scale(scale, scale) * D2D1::Matrix3x2F::Rotation(angle);

        m_bStreamingInitialized = true;
    }

done:
    return hr;
}


// End streaming.

// This method is called if the client sends an MFT_MESSAGE_NOTIFY_END_STREAMING
// message, or when the media type changes. In general, it should be called whenever
// the streaming parameters need to be reset.

HRESULT CGrayscale::EndStreaming()
{
    m_bStreamingInitialized = false;
    return S_OK;
}



// Generate output data.

HRESULT CGrayscale::OnProcessOutput(IMFMediaBuffer *pIn, IMFMediaBuffer *pOut)
{
    BYTE *pDest = nullptr;         // Destination buffer.
    LONG lDestStride = 0;       // Destination stride.

    BYTE *pSrc = nullptr;          // Source buffer.
    LONG lSrcStride = 0;        // Source stride.

    // Helper objects to lock the buffers.
    VideoBufferLock inputLock(pIn);
    VideoBufferLock outputLock(pOut);

    // Stride if the buffer does not support IMF2DBuffer
    LONG lDefaultStride = 0;

    HRESULT hr = GetDefaultStride(m_pInputType, &lDefaultStride);
```

```
    if (FAILED(hr))
    {
        goto done;
    }

    // Lock the input buffer.
    hr = inputLock.LockBuffer(lDefaultStride, m_imageHeightInPixels, &pSrc, &lSrcStride);
    if (FAILED(hr))
    {
        goto done;
    }

    // Lock the output buffer.
    hr = outputLock.LockBuffer(lDefaultStride, m_imageHeightInPixels, &pDest, &lDestStride);
    if (FAILED(hr))
    {
        goto done;
    }

    // Invoke the image transform function.
    assert (m_pTransformFn != nullptr);
    if (m_pTransformFn)
    {
        (*m_pTransformFn)(m_transform, m_rcDest, pDest, lDestStride, pSrc, lSrcStride,
            m_imageWidthInPixels, m_imageHeightInPixels);
    }
    else
    {
        hr = E_UNEXPECTED;
        goto done;
    }


    // Set the data size on the output buffer.
    hr = pOut->SetCurrentLength(m_cbImageSize);

    // The VideoBufferLock class automatically unlocks the buffers.
done:
    return hr;
}


// Flush the MFT.

HRESULT CGrayscale::OnFlush()
{
    // For this MFT, flushing just means releasing the input sample.
    SafeRelease(&m_pSample);
    return S_OK;
}

// Update the format information. This method is called whenever the
// input type is set.

HRESULT CGrayscale::UpdateFormatInfo()
{
    HRESULT hr = S_OK;

    GUID subtype = GUID_NULL;

    m_imageWidthInPixels = 0;
    m_imageHeightInPixels = 0;
    m_cbImageSize = 0;

    m_pTransformFn = nullptr;

    if (m_pInputType != nullptr)
    {
        hr = m_pInputType->GetGUID(MF_MT_SUBTYPE, &subtype);
```

```
        if (FAILED(hr))
        {
            goto done;
        }
        if (subtype == MFVideoFormat_YUY2)
        {
            m_pTransformFn = TransformImage_YUY2;
        }
        else if (subtype == MFVideoFormat_UYVY)
        {
            m_pTransformFn = TransformImage_UYVY;
        }
        else if (subtype == MFVideoFormat_NV12)
        {
            m_pTransformFn = TransformImage_NV12;
        }
        else
        {
            hr = E_UNEXPECTED;
            goto done;
        }

        hr = MFGetAttributeSize(m_pInputType, MF_MT_FRAME_SIZE, &m_imageWidthInPixels,
&m_imageHeightInPixels);
        if (FAILED(hr))
        {
            goto done;
        }

        // Calculate the image size (not including padding)
        hr = GetImageSize(subtype.Data1, m_imageWidthInPixels, m_imageHeightInPixels,
&m_cbImageSize);
    }

done:
    return hr;
}


// Calculate the size of the buffer needed to store the image.

// fcc: The FOURCC code of the video format.

HRESULT GetImageSize(DWORD fcc, UINT32 width, UINT32 height, DWORD* pcbImage)
{
    HRESULT hr = S_OK;

    switch (fcc)
    {
    case FOURCC_YUY2:
    case FOURCC_UYVY:
        // check overflow
        if ((width > MAXDWORD / 2) || (width * 2 > MAXDWORD / height))
        {
            hr = E_INVALIDARG;
        }
        else
        {
            // 16 bpp
            *pcbImage = width * height * 2;
        }
        break;

    case FOURCC_NV12:
        // check overflow
        if ((height/2 > MAXDWORD - height) || ((height + height/2) > MAXDWORD / width))
        {
            hr = E_INVALIDARG;
        }
```

```
        else
        {
            // 12 bpp
            *pcbImage = width * (height + (height/2));
        }
        break;

    default:
        hr = E_FAIL;    // Unsupported type.
    }
    return hr;
}

// Get the default stride for a video format.
HRESULT GetDefaultStride(IMFMediaType *pType, LONG *plStride)
{
    LONG lStride = 0;

    // Try to get the default stride from the media type.
    HRESULT hr = pType->GetUINT32(MF_MT_DEFAULT_STRIDE, (UINT32*)&lStride);
    if (FAILED(hr))
    {
        // Attribute not set. Try to calculate the default stride.
        GUID subtype = GUID_NULL;

        UINT32 width = 0;
        UINT32 height = 0;

        // Get the subtype and the image size.
        hr = pType->GetGUID(MF_MT_SUBTYPE, &subtype);
        if (SUCCEEDED(hr))
        {
            hr = MFGetAttributeSize(pType, MF_MT_FRAME_SIZE, &width, &height);
        }
        if (SUCCEEDED(hr))
        {
            if (subtype == MFVideoFormat_NV12)
            {
                lStride = width;
            }
            else if (subtype == MFVideoFormat_YUY2 || subtype == MFVideoFormat_UYVY)
            {
                lStride = ((width * 2) + 3) & ~3;
            }
            else
            {
                hr = E_INVALIDARG;
            }
        }

        // Set the attribute for later reference.
        if (SUCCEEDED(hr))
        {
            (void)pType->SetUINT32(MF_MT_DEFAULT_STRIDE, UINT32(lStride));
        }
    }
    if (SUCCEEDED(hr))
    {
        *plStride = lStride;
    }
    return hr;
}


// Validate that a rectangle meets the following criteria:
//
//  - All coordinates are non-negative.
//  - The rectangle is not flipped (top > bottom, left > right)
//
```

```
//
// These are the requirements for the destination rectangle.

bool ValidateRect(const RECT& rc)
{
    if (rc.left < 0 || rc.top < 0)
    {
        return false;
    }
    if (rc.left > rc.right || rc.top > rc.bottom)
    {
        return false;
    }
    return true;
}
```

9. Add a new module-definition file to the project, name it `GrayscaleTransform.def`, and then add this code:

```
EXPORTS
    DllCanUnloadNow                     PRIVATE
    DllGetActivationFactory             PRIVATE
    DllGetClassObject                   PRIVATE
```

10. Use the following code to replace the contents of `dllmain.cpp`:

```
#include "pch.h"
#include <initguid.h>
#include <wrl\module.h>

using namespace Microsoft::WRL;

STDAPI_(BOOL) DllMain(_In_ HINSTANCE hInstance, _In_ DWORD reason, _In_opt_ void *reserved)
{
    if (DLL_PROCESS_ATTACH == reason)
    {
        DisableThreadLibraryCalls(hInstance);
    }
    return TRUE;
}

STDAPI DllGetActivationFactory(_In_ HSTRING activatibleClassId, _COM_Outptr_ IActivationFactory
**factory)
{
    return Module<InProc>::GetModule().GetActivationFactory(activatibleClassId, factory);
}

STDAPI DllCanUnloadNow()
{
    return Module<InProc>::GetModule().Terminate() ? S_OK : S_FALSE;
}

STDAPI DllGetClassObject(_In_ REFCLSID rclsid, _In_ REFIID riid, _COM_Outptr_ void **ppv)
{
    return Module<InProc>::GetModule().GetClassObject(rclsid, riid, ppv);
}
```

11. In the project's **Property Pages** dialog box, set the following **Linker** properties.

    a. Under **Input**, for the **Module Definition File**, specify `GrayScaleTransform.def`.

    b. Also under **Input**, add `runtimeobject.lib`, `mfuuid.lib`, and `mfplat.lib` to the **Additional Dependencies** property.

    c. Under **Windows Metadata**, set **Generate Windows Metadata** to **Yes (/WINMD)**.

To use the WRL the custom Media Foundation component from a C# app

1. Add a new **C# Blank App (Universal Windows)** project to the `MediaCapture` solution. Name the project, for example, *MediaCapture*.

2. In the **MediaCapture** project, add a reference to the `GrayscaleTransform` project. To learn how, see How to: Add or remove references by using the reference manager.

3. In `Package.appxmanifest`, on the **Capabilities** tab, select **Microphone** and **Webcam**. Both capabilities are required to capture photos from the webcam.

4. In `MainPage.xaml`, add this code to the root `Grid` element:

```
<StackPanel>
    <TextBlock x:Name="StatusBlock" Margin="10,10,0,0"/>
    <StackPanel Orientation="Horizontal" Grid.Row="1" Margin="0,10,0,0">
        <Button x:Name="StartDevice" Click="StartDevice_Click" IsEnabled="true"
Margin="10,0,10,0">StartDevice</Button>
        <Button x:Name="TakePhoto" Click="TakePhoto_Click" IsEnabled="false"
Margin="0,0,10,0">TakePhoto</Button>
    </StackPanel>
    <StackPanel Orientation="Horizontal" Grid.Row="2" Margin="0,10,0,0">
        <CheckBox x:Name="AddRemoveEffect" Margin="10,0,10,0" Content="Grayscale effect"
IsEnabled="False" Checked="AddRemoveEffect_Checked" Unchecked="AddRemoveEffect_Unchecked"/>
    </StackPanel>
    <Image x:Name="CapturedImage" Width="320" Height="240" Margin="10,10,0,0"
HorizontalAlignment="Left"/>
</StackPanel>
```

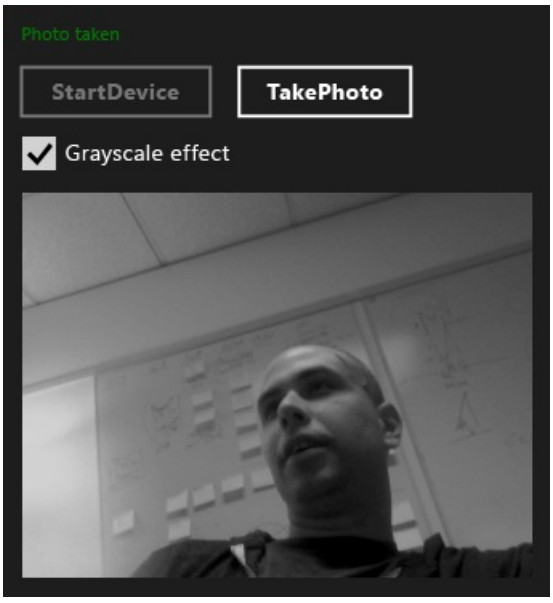
5. Use the following code to replace the contents of `MainPage.xaml.cs`:

```
using System;
using Windows.Devices.Enumeration;
using Windows.Media.Capture;
using Windows.Media.Effects;
using Windows.Media.MediaProperties;
using Windows.Storage.Streams;
using Windows.UI;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Media.Imaging;
using Windows.UI.Xaml.Navigation;

namespace MediaCapture
{
    public sealed partial class MainPage : Page
    {
        // Captures photos from the webcam.
        private Windows.Media.Capture.MediaCapture mediaCapture;

        // Used to display status messages.
        private Brush statusBrush = new SolidColorBrush(Colors.Green);
        // Used to display error messages.
        private Brush exceptionBrush = new SolidColorBrush(Colors.Red);

        public MainPage()
        {
            this.InitializeComponent();
        }

        // Shows a status message.
        private void ShowStatusMessage(string text)
        {
            StatusBlock.Foreground = statusBrush;
```

```
            StatusBlock.Text = text;
        }

        // Shows an error message.
        private void ShowExceptionMessage(Exception ex)
        {
            StatusBlock.Foreground = exceptionBrush;
            StatusBlock.Text = ex.Message;
        }

        // Click event handler for the "Start Device" button.
        private async void StartDevice_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                StartDevice.IsEnabled = false;

                // Enumerate webcams.
                ShowStatusMessage("Enumerating webcams...");
                var devInfoCollection = await
DeviceInformation.FindAllAsync(DeviceClass.VideoCapture);
                if (devInfoCollection.Count == 0)
                {
                    ShowStatusMessage("No webcams found");
                    return;
                }

                // Initialize the MediaCapture object, choosing the first found webcam.
                mediaCapture = new Windows.Media.Capture.MediaCapture();
                var settings = new Windows.Media.Capture.MediaCaptureInitializationSettings();
                settings.VideoDeviceId = devInfoCollection[0].Id;
                await mediaCapture.InitializeAsync(settings);

                // We can now take photos and enable the grayscale effect.
                TakePhoto.IsEnabled = true;
                AddRemoveEffect.IsEnabled = true;

                ShowStatusMessage("Device initialized successfully");
            }
            catch (Exception ex)
            {
                ShowExceptionMessage(ex);
            }
        }

        // Takes a photo from the webcam and displays it.
        private async void TakePhoto_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                ShowStatusMessage("Taking photo...");
                TakePhoto.IsEnabled = false;

                // Capture the photo to an in-memory stream.
                var photoStream = new InMemoryRandomAccessStream();
                await mediaCapture.CapturePhotoToStreamAsync(ImageEncodingProperties.CreateJpeg(),
photoStream);
                ShowStatusMessage("Create photo file successful");

                // Display the photo.
                var bmpimg = new BitmapImage();
                photoStream.Seek(0);
                await bmpimg.SetSourceAsync(photoStream);
                CapturedImage.Source = bmpimg;

                TakePhoto.IsEnabled = true;
                ShowStatusMessage("Photo taken");
            }
            catch (Exception ex)
```

```
            {
                ShowExceptionMessage(ex);
                TakePhoto.IsEnabled = true;
            }
        }

        // Enables the grayscale effect.
        private async void AddRemoveEffect_Checked(object sender, RoutedEventArgs e)
        {
            try
            {
                AddRemoveEffect.IsEnabled = false;
                VideoEffectDefinition def = new
VideoEffectDefinition("GrayscaleTransform.GrayscaleEffect");
                await mediaCapture.AddVideoEffectAsync(def, MediaStreamType.Photo);
                ShowStatusMessage("Add effect to video preview successful");
                AddRemoveEffect.IsEnabled = true;
            }
            catch (Exception ex)
            {
                ShowExceptionMessage(ex);
            }
        }

        // Removes the grayscale effect.
        private async void AddRemoveEffect_Unchecked(object sender, RoutedEventArgs e)
        {
            try
            {
                AddRemoveEffect.IsEnabled = false;
                await mediaCapture.ClearEffectsAsync(Windows.Media.Capture.MediaStreamType.Photo);
                ShowStatusMessage("Remove effect from preview successful");
                AddRemoveEffect.IsEnabled = true;
            }
            catch (Exception ex)
            {
                ShowExceptionMessage(ex);
            }
        }
    }
}
```

The following illustration shows the `MediaCapture app`.



Next Steps

The example shows how to capture photos from the default webcam one at a time. The Media extensions sample does more. It demonstrates how to enumerate webcam devices and work with local scheme handlers. The sample also demonstrates other media effects that work on both individual photos and streams of video.

See also

Windows Runtime C++ Template Library (WRL)
Microsoft Media Foundation

How to: Create a Classic COM Component Using WRL

9/21/2022 • 5 minutes to read • Edit Online

You can use the Windows Runtime C++ Template Library (WRL) to create basic classic COM components for use in desktop apps, in addition to using it for Universal Windows Platform (UWP) apps. For the creation of COM components, the Windows Runtime C++ Template Library may require less code than the ATL. For information about the subset of COM that the Windows Runtime C++ Template Library supports, see Windows Runtime C++ Template Library (WRL).

This document shows how to use the Windows Runtime C++ Template Library to create a basic COM component. Although you can use the deployment mechanism that best fits your needs, this document also shows a basic way to register and consume the COM component from a desktop app.

To use the Windows Runtime C++ Template Library to create a basic classic COM component

1. In Visual Studio, create a **Blank Solution** project. Name the project, for example, `WRLClassicCOM`.
2. Add a **Win32 Project** to the solution. Name the project, for example, `CalculatorComponent`. On the **Application Settings** tab, select **DLL**.
3. Add a **Midl File (.idl)** file to the project. Name the file, for example, `CalculatorComponent.idl`.
4. Add this code to CalculatorComponent.idl:

```
import "ocidl.idl";

[uuid(0DBABB94-CE99-42F7-ACBD-E698B2332C60), version(1.0)]
interface ICalculatorComponent : IUnknown
{
    HRESULT Add([in] int a, [in] int b, [out, retval] int* value);
}

[uuid(9D3E6826-CB8E-4D86-8B14-89F0D7EFCD01), version(1.0)]
library CalculatorComponentLib
{
    [uuid(E68F5EDD-6257-4E72-A10B-4067ED8E85F2), version(1.0)]
    coclass CalculatorComponent
    {
        [default] interface ICalculatorComponent;
    }
};
```

5. In CalculatorComponent.cpp, define the `CalculatorComponent` class. The `CalculatorComponent` class inherits from Microsoft::WRL::RuntimeClass. Microsoft::WRL::RuntimeClassFlags<ClassicCom> specifies that the class derives from IUnknown and not IInspectable. (`IInspectable` is available only to Windows Runtime app components.) `CoCreatableClass` creates a factory for the class that can be used with functions such as CoCreateInstance.

```
#include "pch.h" // Use stdafx.h in Visual Studio 2017 and earlier

#include "CalculatorComponent_h.h"
#include <wrl.h>

using namespace Microsoft::WRL;

class CalculatorComponent: public RuntimeClass<RuntimeClassFlags<ClassicCom>, ICalculatorComponent>
{
public:
    CalculatorComponent()
    {
    }

    STDMETHODIMP Add(_In_ int a, _In_ int b, _Out_ int* value)
    {
        *value = a + b;
        return S_OK;
    }
};

CoCreatableClass(CalculatorComponent);
```

6. Use the following code to replace the code in `dllmain.cpp`. This file defines the DLL export functions. These functions use the Microsoft::WRL::Module class to manage the class factories for the module.

```
#include "pch.h" // Use stdafx.h in Visual Studio 2017 and earlier
#include <wrl\module.h>

using namespace Microsoft::WRL;

#if !defined(__WRL_CLASSIC_COM__)
STDAPI DllGetActivationFactory(_In_ HSTRING activatibleClassId, _COM_Outptr_ IActivationFactory**
factory)
{
    return Module<InProc>::GetModule().GetActivationFactory(activatibleClassId, factory);
}
#endif

#if !defined(__WRL_WINRT_STRICT__)
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, _COM_Outptr_ void** ppv)
{
    return Module<InProc>::GetModule().GetClassObject(rclsid, riid, ppv);
}
#endif

STDAPI DllCanUnloadNow()
{
    return Module<InProc>::GetModule().Terminate() ? S_OK : S_FALSE;
}

STDAPI_(BOOL) DllMain(_In_opt_ HINSTANCE hinst, DWORD reason, _In_opt_ void*)
{
    if (reason == DLL_PROCESS_ATTACH)
    {
        DisableThreadLibraryCalls(hinst);
    }
    return TRUE;
}
```

7. Add a **Module-Definition File (.def)** file to the project. Name the file, for example, `CalculatorComponent.def`. This file gives the linker the names of the functions to be exported. Open the **Property Pages** dialog for your project, then under **Configuration Properties** > **Linker** > **Input**, set

the **Module Definition File** property to your DEF file.

8. Add this code to CalculatorComponent.def:

```
LIBRARY

EXPORTS
    DllGetActivationFactory PRIVATE
    DllGetClassObject       PRIVATE
    DllCanUnloadNow         PRIVATE
```

9. Add runtimeobject.lib to the linker line. To learn how, see `.Lib` Files as Linker Input.

To consume the COM component from a desktop app

1. Register the COM component with the Windows Registry. To do so, create a registration entries file, name it `RegScript.reg`, and add the following text. Replace *<dll-path>* with the path of your DLL—for example, `C:\temp\WRLClassicCOM\Debug\CalculatorComponent.dll`.

```
Windows Registry Editor Version 5.00

[HKEY_CLASSES_ROOT\Wow6432Node\CLSID\{E68F5EDD-6257-4E72-A10B-4067ED8E85F2}]
@="CalculatorComponent Class"

[HKEY_CLASSES_ROOT\Wow6432Node\CLSID\{E68F5EDD-6257-4E72-A10B-4067ED8E85F2}\InprocServer32]
@="<dll-path>"
"ThreadingModel"="Apartment"

[HKEY_CLASSES_ROOT\Wow6432Node\CLSID\{E68F5EDD-6257-4E72-A10B-4067ED8E85F2}\Programmable]

[HKEY_CLASSES_ROOT\Wow6432Node\CLSID\{E68F5EDD-6257-4E72-A10B-4067ED8E85F2}\TypeLib]
@="{9D3E6826-CB8E-4D86-8B14-89F0D7EFCD01}"

[HKEY_CLASSES_ROOT\Wow6432Node\CLSID\{E68F5EDD-6257-4E72-A10B-4067ED8E85F2}\Version]
@="1.0"
```

2. Run RegScript.reg or add it to your project's **Post-Build Event**. For more information, see Pre-build Event/Post-build Event Command Line Dialog Box.

3. Add a **Win32 Console Application** project to the solution. Name the project, for example, `Calculator`.

4. Use this code to replace the contents of `Calculator.cpp`:

```
#include "pch.h" // Use stdafx.h in Visual Studio 2017 and earlier

#include "..\CalculatorComponent\CalculatorComponent_h.h"

const IID IID_ICalculatorComponent =
{0x0DBABB94,0xCE99,0x42F7,0xAC,0xBD,0xE6,0x98,0xB2,0x33,0x2C,0x60};
const CLSID CLSID_CalculatorComponent =
{0xE68F5EDD,0x6257,0x4E72,0xA1,0x0B,0x40,0x67,0xED,0x8E,0x85,0xF2};

// Prints an error string for the provided source code line and HRESULT
// value and returns the HRESULT value as an int.
int PrintError(unsigned int line, HRESULT hr)
{
    wprintf_s(L"ERROR: Line:%d HRESULT: 0x%X\n", line, hr);
    return hr;
}

int wmain()
{
    HRESULT hr;

    // Initialize the COM library.
    hr = CoInitializeEx(nullptr, COINIT_APARTMENTTHREADED);
    if (FAILED(hr))
    {
        return PrintError(__LINE__, hr);
    }

    ICalculatorComponent* calc = nullptr; // Interface to COM component.

    // Create the CalculatorComponent object.
    hr = CoCreateInstance(CLSID_CalculatorComponent, nullptr, CLSCTX_INPROC_SERVER,
IID_PPV_ARGS(&calc));
    if (SUCCEEDED(hr))
    {
        // Test the component by adding two numbers.
        int result;
        hr = calc->Add(4, 5, &result);
        if (FAILED(hr))
        {
            PrintError(__LINE__, hr);
        }
        else
        {
            wprintf_s(L"result = %d\n", result);
        }

        // Free the CalculatorComponent object.
        calc->Release();
    }
    else
    {
        // Object creation failed. Print a message.
        PrintError(__LINE__, hr);
    }

    // Free the COM library.
    CoUninitialize();

    return hr;
}
/* Output:
result = 9
*/
```

Robust Programming

This document uses standard COM functions to demonstrate that you can use the Windows Runtime C++ Template Library to author a COM component and make it available to any COM-enabled technology. You can also use Windows Runtime C++ Template Library types such as Microsoft::WRL::ComPtr in your desktop app to manage the lifetime of COM and other objects. The following code uses the Windows Runtime C++ Template Library to manage the lifetime of the `ICalculatorComponent` pointer. The `CoInitializeWrapper` class is an RAII wrapper that guarantees that the COM library is freed and also guarantees that the lifetime of the COM library outlives the `ComPtr` smart pointer object.

```
#include "pch.h" // Use stdafx.h in Visual Studio 2017 and earlier
#include <wrl.h>

#include "..\CalculatorComponent\CalculatorComponent_h.h"

using namespace Microsoft::WRL;

const IID IID_ICalculatorComponent = {0x0DBABB94,0xCE99,0x42F7,0xAC,0xBD,0xE6,0x98,0xB2,0x33,0x2C,0x60};
const CLSID CLSID_CalculatorComponent = {0xE68F5EDD,0x6257,0x4E72,0xA1,0x0B,0x40,0x67,0xED,0x8E,0x85,0xF2};

// Prints an error string for the provided source code line and HRESULT
// value and returns the HRESULT value as an int.
int PrintError(unsigned int line, HRESULT hr)
{
    wprintf_s(L"ERROR: Line:%d HRESULT: 0x%X\n", line, hr);
    return hr;
}

int wmain()
{
    HRESULT hr;

    // RAII wrapper for managing the lifetime of the COM library.
    class CoInitializeWrapper
    {
        HRESULT _hr;
    public:
        CoInitializeWrapper(DWORD flags)
        {
            _hr = CoInitializeEx(nullptr, flags);
        }
        ~CoInitializeWrapper()
        {
            if (SUCCEEDED(_hr))
            {
                CoUninitialize();
            }
        }
        operator HRESULT()
        {
            return _hr;
        }

    };

    // Initialize the COM library.
    CoInitializeWrapper initialize(COINIT_APARTMENTTHREADED);
    if (FAILED(initialize))
    {
        return PrintError(__LINE__, initialize);
    }

    ComPtr<ICalculatorComponent> calc; // Interface to COM component.

    // Create the CalculatorComponent object.
```

```
    hr = CoCreateInstance(CLSID_CalculatorComponent, nullptr, CLSCTX_INPROC_SERVER,
IID_PPV_ARGS(calc.GetAddressOf()));
    if (SUCCEEDED(hr))
    {
        // Test the component by adding two numbers.
        int result;
        hr = calc->Add(4, 5, &result);
        if (FAILED(hr))
        {
            return PrintError(__LINE__, hr);
        }
        wprintf_s(L"result = %d\n", result);
    }
    else
    {
        // Object creation failed. Print a message.
        return PrintError(__LINE__, hr);
    }

    return 0;
}
```

See also

Windows Runtime C++ Template Library (WRL)

How to: Instantiate WRL Components Directly

9/21/2022 • 2 minutes to read • Edit Online

Learn how to use the Windows Runtime C++ Template Library (WRL)Microsoft::WRL::Make and Microsoft::WRL::Details::MakeAndInitialize functions to instantiate a component from the module that defines it.

By instantiating components directly, you can reduce overhead when you don't need class factories or other mechanisms. You can instantiate a component directly in both Universal Windows Platform apps and in desktop apps.

To learn how to use Windows Runtime C++ Template Library to create a classic COM component and instantiate it from an external desktop app, see How to: Create a Classic COM Component.

This document shows two examples. The first example uses the `Make` function to instantiate a component. The second example uses the `MakeAndInitialize` function to instantiate a component that can fail during construction. (Because COM typically uses HRESULT values, instead of exceptions, to indicate errors, a COM type typically does not throw from its constructor. `MakeAndInitialize` enables a component to validate its construction arguments through the `RuntimeClassInitialize` method.) Both examples define a basic logger interface and implement that interface by defining a class that writes messages to the console.

> **IMPORTANT**
> You can't use the `new` operator to instantiate Windows Runtime C++ Template Library components. Therefore, we recommend that you always use `Make` or `MakeAndInitialize` to instantiate a component directly.

To create and instantiate a basic logger component

1. In Visual Studio, create a **Win32 Console Application** project. Name the project, for example, *WRLLogger*.

2. Add a **Midl File (.idl)** file to the project, name the file `ILogger.idl`, and then add this code:

```
import "ocidl.idl";

// Prints text to the console.
[uuid(AFDB9683-F18A-4B85-90D1-B6158DAFA46C)]
interface ILogger : IUnknown
{
    HRESULT Log([in] LPCWSTR text);
}
```

3. Use the following code to replace the contents of `WRLLogger.cpp`.

```
#include "pch.h" // Use stdafx.h in Visual Studio 2017 and earlier
#include <wrl\implements.h>
#include <comutil.h>

#include "ILogger_h.h"

using namespace Microsoft::WRL;

// Writes logging messages to the console.
class CConsoleWriter : public RuntimeClass<RuntimeClassFlags<ClassicCom>, ILogger>
{
public:
    STDMETHODIMP Log(_In_ PCWSTR text)
    {
        wprintf_s(L"%s\n", text);
        return S_OK;
    }

private:
    // Make destroyable only through Release.
    ~CConsoleWriter()
    {
    }
};

int wmain()
{
    ComPtr<CConsoleWriter> writer = Make<CConsoleWriter>();
    HRESULT hr = writer->Log(L"Logger ready.");
    return hr;
}

/* Output:
Logger ready.
*/
```

To handle construction failure for the basic logger component

1. Use the following code to replace the definition of the `CConsoleWriter` class. This version holds a private string member variable and overrides the `RuntimeClass::RuntimeClassInitialize` method. `RuntimeClassInitialize` fails if the call to `SHStrDup` fails.

```
// Writes logging messages to the console.
class CConsoleWriter : public RuntimeClass<RuntimeClassFlags<ClassicCom>, ILogger>
{
public:
    // Initializes the CConsoleWriter object.
    // Failure here causes your object to fail construction with the HRESULT you choose.
    HRESULT RuntimeClassInitialize(_In_ PCWSTR category)
    {
        return SHStrDup(category, &m_category);
    }

    STDMETHODIMP Log(_In_ PCWSTR text)
    {
        wprintf_s(L"%s: %s\n", m_category, text);
        return S_OK;
    }

private:
    PWSTR m_category;

    // Make destroyable only through Release.
    ~CConsoleWriter()
    {
        CoTaskMemFree(m_category);
    }
};
```

2. Use the following code to replace the definition of `wmain`. This version uses `MakeAndInitialize` to instantiate the `CConsoleWriter` object and checks the HRESULT result.

```
int wmain()
{
    ComPtr<CConsoleWriter> writer;
    HRESULT hr = MakeAndInitialize<CConsoleWriter>(&writer, L"INFO");
    if (FAILED(hr))
    {
        wprintf_s(L"Object creation failed. Result = 0x%x", hr);
        return hr;
    }
    hr = writer->Log(L"Logger ready.");
    return hr;
}

/* Output:
INFO: Logger ready.
*/
```

See also

Windows Runtime C++ Template Library (WRL)
Microsoft::WRL::Make
Microsoft::WRL::Details::MakeAndInitialize

How to: Use winmdidl.exe and midlrt.exe to create .h files from windows metadata

9/21/2022 • 2 minutes to read • Edit Online

Winmdidl.exe and midlrt.exe enable COM-level interaction between native C++ code and Windows Runtime components. Winmdidl.exe takes as input a .winmd file that contains metadata for a Windows Runtime component and outputs an IDL file. Midlrt.exe converts that IDL file into header files that the C++ code can consume. Both tools run on the command line.

You use these tools in two main scenarios:

- Creating custom IDL and header files so that a C++ app written by using the Windows Runtime Template Library (WRL) can consume a custom Windows Runtime component.
- Generating proxy and stub files for user-defined event types in a Windows Runtime Component. For more information, see Custom events and event accessors in Windows Runtime Components.

These tools are required only for parsing custom .winmd files. The .idl and .h files for Windows operating system components are already generated for you. By default in Windows 8.1, they are located in \Program Files (x86)\Windows Kits\8.1\Include\winrt\.

Location of the tools

By default in [Windows 8.1, winmdidl.exe and midlrt.exe are located in C:\Program Files (x86)\Windows Kits\8.1\. Versions of the tools are also available in the \bin\x86\ and \bin\x64\ folders.

Winmdidl command-line arguments

```
Winmdidl.exe [/nologo] [/suppressversioncheck] [/time] [/outdir:dir] [/banner:file] [/utf8] Winmdfile
```

/nologo
Prevents console display of the winmdidl copyright message and version number.

/suppressversioncheck
Not used.

/time
Displays the total execution time in the console output.

/outdir:*dir*
Specifies an output directory. If the path contains spaces, use quotation marks. The default output directory is *<drive>*:\Users*<username>*\AppData\Local\VirtualStore\Program Files (x86)\Microsoft Visual Studio 12.0\.

/banner:*file*
Specifies a file that contains custom text to prepend to the default copyright message and winmdidl version number at the top of the generated .idl file. If the path contains spaces, use quotation marks.

/utf8
Causes the file to be formatted as UTF-8.

Winmdfile

The name of the .winmd file to parse. If the path contains spaces, use quotation marks.

Midlrt command-line arguments

See MIDLRT and Windows Runtime components.

Examples

The following example shows a winmdidl command at a Visual Studio x86 command prompt. It specifies an output directory, and a file that contains special banner text to add to the generated .idl file.

```
C:\Program Files (x86)\Microsoft Visual Studio 12.0>winmdidl /nologo /outdir:c:\users\giraffe\documents\
/banner:c:\users\giraffe\documents\banner.txt "C:\Users\giraffe\Documents\Visual Studio
2013\Projects\Test_for_winmdidl\Debug\Test_for_winmdidl\test_for_winmdidl.winmd"
```

The next example shows the console display from winmdidl that indicates that the operation succeeded.

Generating c:\users\giraffe\documents\\Test_for_winmdidl.idl

Next, midlrt is run on the generated IDL file. Notice that the **metadata_dir** argument is specified after the name of the .idl file. The path of \WinMetadata\ is required—it's the location for windows.winmd.

```
C:\Program Files (x86)\Microsoft Visual Studio 12.0> midlrt
"c:\users\username\documents\test_for_winmdidl.idl" /metadata_dir "C:\Windows\System32\WinMetadata"
```

Remarks

The output file from a winmdidl operation has the same name as the input file but has the .idl file name extension.

If you are developing a Windows Runtime component that will be accessed from the WRL, you can specify winmdidl.exe and midlrt.exe to run as post-build steps so that the .idl and .h files are generated on each build. For an example, see Raising Events in Windows Runtime Components.

Key WRL APIs by Category

9/21/2022 • 2 minutes to read • Edit Online

The following tables list primary Windows Runtime C++ Template Library classes, structs, functions, and macros. Constructs in helper namespaces and classes are omitted. These lists augment the API documentation, which is arranged by namespace.

Classes

TITLE	DESCRIPTION
ActivationFactory Class	Enables one or more classes to be activated by the Windows Runtime.
AsyncBase Class	Implements the Windows Runtime asynchronous state machine.
ClassFactory Class	Implements the basic functionality of the `IClassFactory` interface.
ComPtr Class	Creates a *smart pointer* type that represents the interface specified by the template parameter. ComPtr automatically maintains a reference count for the underlying interface pointer and releases the interface when the reference count goes to zero.
Event Class (Windows Runtime C++ Template Library)	Represents an event.
EventSource Class	Represents an event. `EventSource` member functions add, remove, and invoke event handlers.
FtmBase Class	Represents a free-threaded marshaler object.
HandleT Class	Represents a handle to an object.
HString Class	Provides support for manipulating HSTRING handles.
HStringReference Class	Represents an HSTRING that is created from an existing string.
Module Class	Represents a collection of related objects.
Module::GenericReleaseNotifier Class	Invokes an event handler when the last object in the current module is released. The event handler is specified by on a lambda, functor, or pointer-to-function.
Module::MethodReleaseNotifier Class	Invokes an event handler when the last object in the current module is released. The event handler is specified by an object and its pointer-to-a-method member.

TITLE	DESCRIPTION
Module::ReleaseNotifier Class	Invokes an event handler when the last object in a module is released.
RoInitializeWrapper Class	Initializes the Windows Runtime.
RuntimeClass Class	Represents an instantiated class that inherits the specified number of interfaces, and provides the specified Windows Runtime, classic COM, and weak reference support.
SimpleActivationFactory Class	Provides a fundamental mechanism to create a Windows Runtime or classic COM base class.
SimpleClassFactory Class	Provides a fundamental mechanism to create a base class.
WeakRef Class	Represents a *weak reference* that can be used by only the Windows Runtime, not classic COM. A weak reference represents an object that might or might not be accessible.

Structures

TITLE	DESCRIPTION
ChainInterfaces Structure	Specifies verification and initialization functions that can be applied to a set of interface IDs.
CloakedIid Structure	Indicates to the `RuntimeClass`, `Implements` and `ChainInterfaces` templates that the specified interface is not accessible in the IID list.
Implements Structure	Implements `QueryInterface` and `GetIid` for the specified interfaces.
MixIn Structure	Ensures that a runtime class derives from Windows Runtime interfaces, if any, and then classic COM interfaces.

Functions

TITLE	DESCRIPTION
ActivateInstance Function	Registers and retrieves an instance of a specified type defined in a specified class ID.
AsWeak Function	Retrieves a weak reference to a specified instance.
Callback Function	Creates an object whose member function is a callback method.
CreateActivationFactory Function	Creates a factory that produces instances of the specified class that can be activated by the Windows Runtime.

TITLE	DESCRIPTION
CreateClassFactory Function	Creates a factory that produces instances of the specified class.
GetActivationFactory Function	Retrieves an activation factory for the type specified by the template parameter.
Make Function	Initializes the specified Windows Runtime class.

Macros

TITLE	DESCRIPTION
ActivatableClass Macros	Populates an internal cache that contains a factory that can create an instance of the specified class.
InspectableClass Macro	Sets the runtime class name and trust level.

See also

Windows Runtime C++ Template Library (WRL)

WRL Reference

9/21/2022 • 2 minutes to read • Edit Online

This section contains reference information for the Windows Runtime C++ Template Library (WRL).

> **NOTE**
> The Windows Runtime C++ Template Library defines functionality that supports the Windows Runtime C++ Template Library infrastructure and is not intended to be used directly from your code. Such functionality is noted in this documentation.

In This Section

Microsoft::WRL Namespace
Defines the fundamental types that make up the Windows Runtime C++ Template Library.

Microsoft::WRL::Wrappers Namespace
Defines Resource Acquisition Is Initialization (RAII) wrapper types that simplify the lifetime management of objects, strings, and handles.

Microsoft::WRL::Wrappers::HandleTraits Namespace
Describes characteristics of common handle-based resource types.

Windows::Foundation Namespace
Enables fundamental Windows Runtime functionality, such as object and factory creation.

Related Sections

Windows Runtime C++ Template Library (WRL)
Introduces Windows Runtime C++ Template Library, a COM-based template library that provides a low-level way to author and use Windows Runtime components.

Microsoft::WRL Namespace

9/21/2022 • 2 minutes to read • Edit Online

Defines the fundamental types that make up the Windows Runtime C++ Template Library.

Syntax

```
namespace Microsoft::WRL;
```

Members

Typedefs

NAME	DESCRIPTION
`InhibitWeakReferencePolicy`	`RuntimeClassFlags<WinRt | InhibitWeakReference>`

Classes

NAME	DESCRIPTION
ActivationFactory Class	Enables one or more classes to be activated by the Windows Runtime.
AsyncBase Class	Implements the Windows Runtime asynchronous state machine.
ClassFactory Class	Implements the basic functionality of the `IClassFactory` interface.
ComPtr Class	Creates a *smart pointer* type that represents the interface specified by the template parameter. ComPtr automatically maintains a reference count for the underlying interface pointer and releases the interface when the reference count goes to zero.
DeferrableEventArgs Class	A template class used for the event argument types for deferrals.
EventSource Class	Represents an event. `EventSource` member functions add, remove, and invoke event handlers.
FtmBase Class	Represents a free-threaded marshaler object.
Module Class	Represents a collection of related objects.
RuntimeClass Class	Represents an instantiated class that inherits the specified number of interfaces, and provides the specified Windows Runtime, classic COM, and weak reference support.

NAME	DESCRIPTION
SimpleActivationFactory Class	Provides a fundamental mechanism to create a Windows Runtime or classic COM base class.
SimpleClassFactory Class	Provides a fundamental mechanism to create a base class.
WeakRef Class	Represents a *weak reference* that can be used by only the Windows Runtime, not classic COM. A weak reference represents an object that might or might not be accessible.

Structures

NAME	DESCRIPTION
ChainInterfaces Structure	Specifies verification and initialization functions that can be applied to a set of interface IDs.
CloakedIid Structure	Indicates to the `RuntimeClass`, `Implements` and `ChainInterfaces` templates that the specified interface is not accessible in the IID list.
Implements Structure	Implements `QueryInterface` and `GetIid` for the specified interfaces.
MixIn Structure	Ensures that a runtime class derives from Windows Runtime interfaces, if any, and then classic COM interfaces.
RuntimeClassFlags Structure	Contains the type for an instance of a RuntimeClass.

Enumerations

NAME	DESCRIPTION
AsyncResultType Enumeration	Specifies the type of result returned by the `GetResults()` method.
ModuleType Enumeration	Specifies whether a module should support an in-process server or an out-of-process server.
RuntimeClassType Enumeration	Specifies the type of RuntimeClass instance that is supported.

Functions

NAME	DESCRIPTION
AsWeak Function	Retrieves a weak reference to a specified instance.
Callback Function (WRL)	Creates an object whose member function is a callback method.
CreateActivationFactory Function	Creates a factory that produces instances of the specified class that can be activated by the Windows Runtime.

NAME	DESCRIPTION
CreateClassFactory Function	Creates a factory that produces instances of the specified class.
Make Function	Initializes the specified Windows Runtime class.

Requirements

Header: async.h, client.h, corewrappers.h, event.h, ftm.h, implements.h, internal.h, module.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL::Wrappers Namespace

ActivatableClass Macros

9/21/2022 • 2 minutes to read • Edit Online

Populates an internal cache that contains a factory that can create an instance of the specified class.

Syntax

```
ActivatableClass(
   className
);

ActivatableClassWithFactory(
   className,
   factory
);

ActivatableClassWithFactoryEx(
   className,
   factory,
   serverName
);
```

Parameters

className
Name of the class to create.

factory
Factory that will create an instance of the specified class.

serverName
A name that specifies a subset of factories in the module.

Remarks

Do not use these macros with classic COM unless you use the `#undef` directive to ensure that the `__WRL_WINRT_STRICT__` macro definition is removed.

Requirements

Header: module.h

Namespace: Microsoft::WRL

See also

Module Class

ActivationFactory Class

9/21/2022 • 2 minutes to read • Edit Online

Enables one or more classes to be activated by the Windows Runtime.

Syntax

```
template <
    typename I0 = Details::Nil,
    typename I1 = Details::Nil,
    typename I2 = Details::Nil
>
class ActivationFactory :
    public Details::RuntimeClass<
        typename Details::InterfaceListHelper<
            IActivationFactory,
            I0,
            I1,
            I2,
            Details::Nil
        >::TypeT,
        RuntimeClassFlags<WinRt | InhibitWeakReference>,
        false
    >;
```

Parameters

I0

The zeroth interface.

I1

The first interface.

I2

The second interface.

Remarks

`ActivationFactory` provides registration methods and basic functionality for the `IActivationFactory` interface. `ActivationFactory` also enables you to provide a custom factory implementation.

The following code fragment symbolically illustrates how to use ActivationFactory.

```
struct MyClassFactory : public ActivationFactory<IMyAddtionalInterfaceOnFactory>
{
    STDMETHOD(ActivateInstance) (_Outptr_result_nullonfailure_ IInspectable** ppvObject)
    {
        // my custom implementation

        return S_OK;
    }
};

ActivatableClassWithFactory(MyClass, MyClassFactory);
// or if a default factory is used:
//ActivatableClassWithFactory(MyClass, SimpleActivationFactory);
```

The following code fragment shows how to use the Implements structure to specify more than three interface IDs.

```
struct MyFactory : ActivationFactory<Implements<I1, I2, I3>, I4, I5>;
```

Members

Public Constructors

NAME	DESCRIPTION
ActivationFactory::ActivationFactory	Initializes the `ActivationFactory` class.

Public Methods

NAME	DESCRIPTION
ActivationFactory::AddRef	Increments the reference count of the current `ActivationFactory` object.
ActivationFactory::GetIids	Retrieves an array of implemented interface IDs.
ActivationFactory::GetRuntimeClassName	Gets the runtime class name of the object that the current `ActivationFactory` instantiates.
ActivationFactory::GetTrustLevel	Gets the trust level of the object that the current `ActivationFactory` instantiates.
ActivationFactory::QueryInterface	Retrieves a pointer to the specified interface.
ActivationFactory::Release	Decrements the reference count of the current `ActivationFactory` object.

Inheritance Hierarchy

`I0`

`ChainInterfaces`

`I0`

`RuntimeClassBase`

`ImplementsHelper`

`DontUseNewUseMake`

`RuntimeClassFlags`

`RuntimeClassBaseT`

`RuntimeClass`

`ActivationFactory`

Requirements

Header: module.h

Namespace: Microsoft::WRL

ActivationFactory::ActivationFactory

Initializes the `ActivationFactory` class.

```
ActivationFactory();
```

ActivationFactory::AddRef

Increments the reference count of the current `ActivationFactory` object.

```
STDMETHOD_(
   ULONG,
   AddRef
)();
```

Return Value

S_OK if successful; otherwise, an HRESULT that describes the failure.

ActivationFactory::GetIids

Retrieves an array of implemented interface IDs.

```
STDMETHOD(
   GetIids
)(_Out_ ULONG *iidCount, _Deref_out_ _Deref_post_cap_(*iidCount) IID **iids);
```

Parameters

iidCount
When this operation completes, the number of interace IDs in the *iids* array.

iids
When this operation completes, an array of implemented interface IDs.

Return Value

S_OK if successful; otherwise, an HRESULT that describes the failure. E_OUTOFMEMORY is a possible failure HRESULT.

ActivationFactory::GetRuntimeClassName

Gets the runtime class name of the object that the current `ActivationFactory` instantiates.

```
STDMETHOD(
   GetRuntimeClassName
)(_Out_ HSTRING* runtimeName);
```

Parameters

runtimeName
When this operation completes, a handle to a string that contains the runtime class name of the object that the

current `ActivationFactory` instantiates.

Return Value

S_OK if successful; otherwise, an HRESULT that describes the failure.

ActivationFactory::GetTrustLevel

Gets the trust level of the object that the current `ActivationFactory` instantiates.

```
STDMETHOD(
   GetTrustLevel
)(_Out_ TrustLevel* trustLvl);
```

Parameters

trustLvl
When this operation completes, the trust level of the runtime class that the `ActivationFactory` instantiates.

Return Value

S_OK if successful; otherwise, an assertion error is emitted and *trustLvl* is set to `FullTrust`.

ActivationFactory::QueryInterface

Retrieves a pointer to the specified interface.

```
STDMETHOD(
   QueryInterface
)(REFIID riid, _Deref_out_ void **ppvObject);
```

Parameters

riid
An interface ID.

ppvObject
When this operation is complete, a pointer to the interface specified by parameter *riid*.

Return Value

S_OK if successful; otherwise, an HRESULT that describes the failure.

ActivationFactory::Release

Decrements the reference count of the current `ActivationFactory` object.

```
STDMETHOD_(
   ULONG,
   Release
)();
```

Return Value

S_OK if successful; otherwise, an HRESULT that describes the failure.

AgileActivationFactory Class

9/21/2022 • 2 minutes to read • Edit Online

Represents an apartment-friendly activation factory that implements FtmBase.

Syntax

```
template <
    typename I0 = Details::Nil,
    typename I1 = Details::Nil,
    typename I2 = Details::Nil,
    FactoryCacheFlags cacheFlagValue = FactoryCacheDefault
>
class AgileActivationFactory :
    public ActivationFactory<
        Implements<FtmBase, I0>,
        I1,
        I2,
        cacheFlagValue
    >;
```

Requirements

Header: module.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace
ActivationFactory Class

AgileEventSource Class

9/21/2022 • 2 minutes to read • Edit Online

Represents an event that is raised by a agile component, which is a component that can be accessed from any thread. Inherits from EventSource and overrides the `Add` member function with an additional type parameter for specifying options for how to invoke the agile event.

Syntax

```
template<
    typename TDelegateInterface,
    typename TEventSourceOptions = Microsoft::WRL::InvokeModeOptions<FireAll>
>
class AgileEventSource :
    public Microsoft::WRL::EventSource<
        TDelegateInterface, TEventSourceOptions>;
```

Parameters

TDelegateInterface
The interface to a delegate that represents an event handler.

TEventSourceOptions
An InvokeModeOptions structure whose invokeMode field is set to `InvokeMode::StopOnFirstError` or `InvokeMode::FireAll`.

Remarks

The vast majority of components in the Windows Runtime are agile components. For more information, see Threading and Marshaling (C++/CX).

Inheritance Hierarchy

`EventSource`

`AgileEventSource`

Requirements

Header: event.h

Namespace: Microsoft::WRL

Members

Public Methods

NAME	DESCRIPTION

NAME	DESCRIPTION
AgileEventSource::Add Method	Appends the agile event handler represented by the specified delegate interface to the set of event handlers for the current **AgileEventSource** object.

AgileEventSource::Add Method

Appends the event handler represented by the specified delegate interface to the set of event handlers for the current EventSource object.

Syntax

```
HRESULT Add(
   _In_ TDelegateInterface* delegateInterface,
   _Out_ EventRegistrationToken* token
);
```

Parameters

delegateInterface
The interface to a delegate object, which represents an event handler.

token
When this operation completes, a handle that represents the event. Use this token as the parameter to the `Remove()` method to discard the event handler.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

See also

Microsoft::WRL Namespace

AsWeak Function

9/21/2022 • 2 minutes to read • Edit Online

Retrieves a weak reference to a specified instance.

Syntax

```
template<typename T>
HRESULT AsWeak(
   _In_ T* p,
   _Out_ WeakRef* pWeak
);
```

Parameters

T

A pointer to the type of parameter *p*.

p

An instance of a type.

pWeak

When this operation completes, a pointer to a weak reference to parameter *p*.

Return Value

S_OK, if this operation is successful; otherwise, an error HRESULT that indicates the cause of the failure.

Requirements

Header: client.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

AsyncBase Class

9/21/2022 • 5 minutes to read • Edit Online

Implements the Windows Runtime asynchronous state machine.

Syntax

```
template <
    typename TComplete,
    typename TProgress = Details::Nil,
    AsyncResultType resultType = SingleResult
>
class AsyncBase : public AsyncBase<TComplete, Details::Nil, resultType>;

template <typename TComplete, AsyncResultType resultType>
class AsyncBase<TComplete, Details::Nil, resultType> :
    public Microsoft::WRL::Implements<IAsyncInfo>;
```

Parameters

TComplete
An event handler that is called when an asynchronous operation completes.

TProgress
An event handler that is called when a running asynchronous operation reports the current progress of the operation.

resultType
One of the AsyncResultType enumeration values. By default, `SingleResult`.

Members

Public Constructors

NAME	DESCRIPTION
AsyncBase::AsyncBase	Initializes an instance of the `AsyncBase` class.

Public Methods

NAME	DESCRIPTION
AsyncBase::Cancel	Cancels an asynchronous operation.
AsyncBase::Close	Closes the asynchronous operation.
AsyncBase::FireCompletion	Invokes the completion event handler, or resets the internal progress delegate.
AsyncBase::FireProgress	Invokes the current progress event handler.

NAME	DESCRIPTION
AsyncBase::get_ErrorCode	Retrieves the error code for the current asynchronous operation.
AsyncBase::get_Id	Retrieves the handle of the asynchronous operation.
AsyncBase::get_Status	Retrieves a value that indicates the status of the asynchronous operation.
AsyncBase::GetOnComplete	Copies the address of the current completion event handler to the specified variable.
AsyncBase::GetOnProgress	Copies the address of the current progress event handler to the specified variable.
AsyncBase::put_Id	Sets the handle of the asynchronous operation.
AsyncBase::PutOnComplete	Sets the address of the completion event handler to the specified value.
AsyncBase::PutOnProgress	Sets the address of the progress event handler to the specified value.

Protected Methods

NAME	DESCRIPTION
AsyncBase::CheckValidStateForDelegateCall	Tests whether delegate properties can be modified in the current asynchronous state.
AsyncBase::CheckValidStateForResultsCall	Tests whether the results of an asynchronous operation can be collected in the current asynchronous state.
AsyncBase::ContinueAsyncOperation	Determines whether the asynchronous operation should continue processing or should halt.
AsyncBase::CurrentStatus	Retrieves the status of the current asynchronous operation.
AsyncBase::ErrorCode	Retrieves the error code for the current asynchronous operation.
AsyncBase::OnCancel	When overridden in a derived class, cancels an asynchronous operation.
AsyncBase::OnClose	When overridden in a derived class, closes an asynchronous operation.
AsyncBase::OnStart	When overridden in a derived class, starts an asynchronous operation.
AsyncBase::Start	Starts the asynchronous operation.
AsyncBase::TryTransitionToCompleted	Indicates whether the current asynchronous operation has completed.

NAME	DESCRIPTION
AsyncBase::TryTransitionToError	Indicates whether the specified error code can modify the internal error state.

Inheritance Hierarchy

`AsyncBase`

`AsyncBase`

Requirements

Header: async.h

Namespace: Microsoft::WRL

AsyncBase::AsyncBase

Initializes an instance of the `AsyncBase` class.

```
AsyncBase();
```

AsyncBase::Cancel

Cancels an asynchronous operation.

```
STDMETHOD(
   Cancel
)(void);
```

Return Value

By default, always returns S_OK.

Remarks

`Cancel()` is a default implementation of `IAsyncInfo::Cancel`, and does no actual work. To actually cancel an asynchronous operation, override the `OnCancel()` pure virtual method.

AsyncBase::CheckValidStateForDelegateCall

Tests whether delegate properties can be modified in the current asynchronous state.

```
inline HRESULT CheckValidStateForDelegateCall();
```

Return Value

S_OK if delegate properties can be modified; otherwise, E_ILLEGAL_METHOD_CALL.

AsyncBase::CheckValidStateForResultsCall

Tests whether the results of an asynchronous operation can be collected in the current asynchronous state.

```
inline HRESULT CheckValidStateForResultsCall();
```

Return Value

S_OK if results can be collected; otherwise, E_ILLEGAL_METHOD_CALLE_ILLEGAL_METHOD_CALL.

AsyncBase::Close

Closes the asynchronous operation.

```
STDMETHOD(
   Close
)(void) override;
```

Return Value

S_OK if the operation closes or is already closed; otherwise, E_ILLEGAL_STATE_CHANGE.

Remarks

`Close()` is a default implementation of `IAsyncInfo::Close`, and does no actual work. To actually close an asynchronous operation, override the `OnClose()` pure virtual method.

AsyncBase::ContinueAsyncOperation

Determines whether the asynchronous operation should continue processing or should halt.

```
inline bool ContinueAsyncOperation();
```

Return Value

`true` if the current state of the asynchronous operation is *started*, which means the operation should continue. Otherwise, `false`, which means the operation should halt.

AsyncBase::CurrentStatus

Retrieves the status of the current asynchronous operation.

```
inline void CurrentStatus(
   Details::AsyncStatusInternal *status
);
```

Parameters

status
The location where this operation stores the current status.

Remarks

This operation is thread-safe.

AsyncBase::ErrorCode

Retrieves the error code for the current asynchronous operation.

```
inline void ErrorCode(
   HRESULT *error
);
```

Parameters

error
The location where this operation stores the current error code.

Remarks

This operation is thread-safe.

AsyncBase::FireCompletion

Invokes the completion event handler, or resets the internal progress delegate.

```
void FireCompletion(
   void
) override;

virtual void FireCompletion();
```

Remarks

The first version of `FireCompletion()` resets the internal progress delegate variable. The second version invokes the completion event handler if the asynchronous operation is complete.

AsyncBase::FireProgress

Invokes the current progress event handler.

```
void FireProgress(
   const typename ProgressTraits::Arg2Type arg
);
```

Parameters

arg
The event handler method to invoke.

Remarks

`ProgressTraits` is derived from ArgTraitsHelper Structure.

AsyncBase::get_ErrorCode

Retrieves the error code for the current asynchronous operation.

```
STDMETHOD(
   get_ErrorCode
)(HRESULT* errorCode) override;
```

Parameters

errorCode
The location where the current error code is stored.

Return Value

S_OK if successful; otherwise, E_ILLEGAL_METHOD_CALL if the current asynchronous operation is closed.

AsyncBase::get_Id

Retrieves the handle of the asynchronous operation.

```
STDMETHOD(
   get_Id
)(unsigned int *id) override;
```

Parameters

id
The location where the handle is to be stored.

Return Value

S_OK if successful; otherwise, E_ILLEGAL_METHOD_CALL.

Remarks

This method implements `IAsyncInfo::get_Id`.

AsyncBase::get_Status

Retrieves a value that indicates the status of the asynchronous operation.

```
STDMETHOD(
   get_Status
)(AsyncStatus *status) override;
```

Parameters

status
The location where the status is to be stored. For more information, see `Windows::Foundation::AsyncStatus` enumeration.

Return Value

S_OK if successful; otherwise, E_ILLEGAL_METHOD_CALL.

Remarks

This method implements `IAsyncInfo::get_Status`.

AsyncBase::GetOnComplete

Copies the address of the current completion event handler to the specified variable.

```
STDMETHOD(
   GetOnComplete
)(TComplete** completeHandler);
```

Parameters

completeHandler
The location where the address of the current completion event handler is stored.

Return Value

S_OK if successful; otherwise, E_ILLEGAL_METHOD_CALL.

AsyncBase::GetOnProgress

Copies the address of the current progress event handler to the specified variable.

```
STDMETHOD(
   GetOnProgress
)(TProgress** progressHandler);
```

Parameters

progressHandler
The location where the address of the current progress event handler is stored.

Return Value

S_OK if successful; otherwise, E_ILLEGAL_METHOD_CALL.

AsyncBase::OnCancel

When overridden in a derived class, cancels an asynchronous operation.

```
virtual void OnCancel(
   void
) = 0;
```

AsyncBase::OnClose

When overridden in a derived class, closes an asynchronous operation.

```
virtual void OnClose(
   void
) = 0;
```

AsyncBase::OnStart

When overridden in a derived class, starts an asynchronous operation.

```
virtual HRESULT OnStart(
   void
) = 0;
```

AsyncBase::put_Id

Sets the handle of the asynchronous operation.

```
STDMETHOD(
   put_Id
)(const unsigned int id);
```

Parameters

id
A nonzero handle.

Return Value

S_OK if successful; otherwise, E_INVALIDARG or E_ILLEGAL_METHOD_CALL.

AsyncBase::PutOnComplete

Sets the address of the completion event handler to the specified value.

```
STDMETHOD(
   PutOnComplete
)(TComplete* completeHandler);
```

Parameters

completeHandler
The address to which the completion event handler is set.

Return Value

S_OK if successful; otherwise, E_ILLEGAL_METHOD_CALL.

AsyncBase::PutOnProgress

Sets the address of the progress event handler to the specified value.

```
STDMETHOD(
   PutOnProgress
)(TProgress* progressHandler);
```

Parameters

progressHandler
The address to which the progress event handler is set.

Return Value

S_OK if successful; otherwise, E_ILLEGAL_METHOD_CALL.

AsyncBase::Start

Starts the asynchronous operation.

```
STDMETHOD(
   Start
)(void);
```

Return Value

S_OK if the operation starts or is already started; otherwise, E_ILLEGAL_STATE_CHANGE.

Remarks

`Start()` is a protected method that is not externally visible because async operations "hot start" before returning to the caller.

AsyncBase::TryTransitionToCompleted

Indicates whether the current asynchronous operation has completed.

```
bool TryTransitionToCompleted(
   void
);
```

Return Value

`true` if the asynchronous operation has completed; otherwise, `false`.

AsyncBase::TryTransitionToError

Indicates whether the specified error code can modify the internal error state.

```
bool TryTransitionToError(
   const HRESULT error
);
```

Parameters

error
An error HRESULT.

Return Value

`true` if the internal error state was changed; otherwise, `false`.

Remarks

This operation modifies the error state only if the error state is already set to S_OK. This operation has no effect if the error state is already error, cancelled, completed, or closed.

AsyncResultType Enumeration

9/21/2022 • 2 minutes to read • Edit Online

Specifies the type of result returned by the `GetResults()` method.

Syntax

```
enum AsyncResultType;
```

Members

Values

NAME	DESCRIPTION
`MultipleResults`	A set of multiple results, which are presented progressively between `Start` state and before `Close()` is called.
`SingleResult`	A single result, which is presented after the `Complete` event occurs.

Requirements

Header: async.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

Callback Function (WRL)

9/21/2022 • 2 minutes to read • Edit Online

Creates an object whose member function is a callback method.

Syntax

```
template<
   typename TDelegateInterface,
   typename TCallback
>
ComPtr<TDelegateInterface> Callback(
   TCallback callback
);
template<
   typename TDelegateInterface,
   typename TCallbackObject
>
ComPtr<TDelegateInterface> Callback(
   _In_ TCallbackObject *object,
   _In_ HRESULT (TCallbackObject::* method)()
);
template<
   typename TDelegateInterface,
   typename TCallbackObject,
   typename TArg1
>
ComPtr<TDelegateInterface> Callback(
   _In_ TCallbackObject *object,
   _In_ HRESULT (TCallbackObject::* method)(TArg1)
);
template<
   typename TDelegateInterface,
   typename TCallbackObject,
   typename TArg1,
   typename TArg2
>
ComPtr<TDelegateInterface> Callback(
   _In_ TCallbackObject *object,
   _In_ HRESULT (TCallbackObject::* method)(TArg1,
   TArg2)
);
template<
   typename TDelegateInterface,
   typename TCallbackObject,
   typename TArg1,
   typename TArg2,
   typename TArg3
>
ComPtr<TDelegateInterface> Callback(
   _In_ TCallbackObject *object,
   _In_ HRESULT (TCallbackObject::* method)(TArg1,
   TArg2,
   TArg3)
);
template<
   typename TDelegateInterface,
   typename TCallbackObject,
   typename TArg1,
   typename TArg2,
   typename TArg3,
```

```
   typename TArg4
>
ComPtr<TDelegateInterface> Callback(
   _In_ TCallbackObject *object,
   _In_ HRESULT (TCallbackObject::* method)(TArg1,
   TArg2,
   TArg3,
   TArg4)
);
template<
   typename TDelegateInterface,
   typename TCallbackObject,
   typename TArg1,
   typename TArg2,
   typename TArg3,
   typename TArg4,
   typename TArg5
>
ComPtr<TDelegateInterface> Callback(
   _In_ TCallbackObject *object,
   _In_ HRESULT (TCallbackObject::* method)(TArg1,
   TArg2,
   TArg3,
   TArg4,
   TArg5)
);
template<
   typename TDelegateInterface,
   typename TCallbackObject,
   typename TArg1,
   typename TArg2,
   typename TArg3,
   typename TArg4,
   typename TArg5,
   typename TArg6
>
ComPtr<TDelegateInterface> Callback(
   _In_ TCallbackObject *object,
   _In_ HRESULT (TCallbackObject::* method)(TArg1,
   TArg2,
   TArg3,
   TArg4,
   TArg5,
   TArg6)
);
template<
   typename TDelegateInterface,
   typename TCallbackObject,
   typename TArg1,
   typename TArg2,
   typename TArg3,
   typename TArg4,
   typename TArg5,
   typename TArg6,
   typename TArg7
>
ComPtr<TDelegateInterface> Callback(
   _In_ TCallbackObject *object,
   _In_ HRESULT (TCallbackObject::* method)(TArg1,
   TArg2,
   TArg3,
   TArg4,
   TArg5,
   TArg6,
   TArg7)
);
template<
   typename TDelegateInterface,
   typename TCallbackObject,
```

```
    typename TArg1,
    typename TArg2,
    typename TArg3,
    typename TArg4,
    typename TArg5,
    typename TArg6,
    typename TArg7,
    typename TArg8
>
ComPtr<TDelegateInterface> Callback(
    _In_ TCallbackObject *object,
    _In_ HRESULT (TCallbackObject::* method)(TArg1,
    TArg2,
    TArg3,
    TArg4,
    TArg5,
    TArg6,
    TArg7,
    TArg8)
);
template<
    typename TDelegateInterface,
    typename TCallbackObject,
    typename TArg1,
    typename TArg2,
    typename TArg3,
    typename TArg4,
    typename TArg5,
    typename TArg6,
    typename TArg7,
    typename TArg8,
    typename TArg9
>
ComPtr<TDelegateInterface> Callback(
    _In_ TCallbackObject *object,
    _In_ HRESULT (TCallbackObject::* method)(TArg1,
    TArg2,
    TArg3,
    TArg4,
    TArg5,
    TArg6,
    TArg7,
    TArg8,
    TArg9)
);
```

Parameters

TDelegateInterface
A template parameter that specifies the interface of the delegate to call when an event occurs.

TCallback
A template parameter that specifies the type of an object that represents an object and its callback member function.

TCallbackObject
A template parameter that specifies the object whose member function is the method to call when an event occurs.

TArg1
A template parameter that specifies the type of the first callback method argument.

TArg2
A template parameter that specifies the type of the second callback method argument.

TArg3
A template parameter that specifies the type of the third callback method argument.

TArg4
A template parameter that specifies the type of the fourth callback method argument.

TArg5
A template parameter that specifies the type of the fifth callback method argument.

TArg6
A template parameter that specifies the type of the sixth callback method argument.

TArg7
A template parameter that specifies the type of the seventh callback method argument.

TArg8
A template parameter that specifies the type of the eighth callback method argument.

TArg9
A template parameter that specifies the type of the ninth callback method argument.

callback
An object that represents the callback object and its member function.

object
The object whose member function is called when an event occurs.

method
The member function to call when an event occurs.

Return Value

An object whose member function is the specified callback method.

Remarks

The base of a delegate object must be `IUnknown`, not `IInspectable`.

Requirements

Header: event.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

CancelTransitionPolicy Enumeration

9/21/2022 • 2 minutes to read • Edit Online

Indicates how an asynchronous operation's attempt to transition to a terminal state of completed or error should behave with respect to a client-requested canceled state.

Syntax

```
enum CancelTransitionPolicy;
```

Members

Values

NAME	DESCRIPTION
`RemainCanceled`	If the asynchronous operation is currently in a client-requested canceled state, this indicates that it will stay in the canceled state as opposed to transitioning to a terminal completed or error state.
`TransitionFromCanceled`	If the asynchronous operation is currently in a client-requested canceled state, this indicates that state should transition from that canceled state to the terminal state of completed or error as determined by the call that utilizes this flag.

Requirements

Header: async.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

ChainInterfaces Structure

9/21/2022 • 2 minutes to read • Edit Online

Specifies verification and initialization functions that can be applied to a set of interface IDs.

Syntax

```
template <
    typename I0,
    typename I1,
    typename I2 = Details::Nil,
    typename I3 = Details::Nil,
    typename I4 = Details::Nil,
    typename I5 = Details::Nil,
    typename I6 = Details::Nil,
    typename I7 = Details::Nil,
    typename I8 = Details::Nil,
    typename I9 = Details::Nil
>
struct ChainInterfaces : I0;

template <
    typename DerivedType,
    typename BaseType,
    bool hasImplements,
    typename I1,
    typename I2,
    typename I3,
    typename I4,
    typename I5,
    typename I6,
    typename I7,
    typename I8,
    typename I9
>
struct ChainInterfaces<
    MixIn<
        DerivedType,
        BaseType,
        hasImplements
    >, I1, I2, I3, I4, I5, I6, I7, I8, I9
>;
```

Parameters

I0

(Required) Interface ID 0.

I1

(Required) Interface ID 1.

I2

(Optional) Interface ID 2.

I3

(Optional) Interface ID 3.

I4

(Optional) Interface ID 4.

I5
(Optional) Interface ID 5.

I6
(Optional) Interface ID 6.

I7
(Optional) Interface ID 7.

I8
(Optional) Interface ID 8.

I9
(Optional) Interface ID 9.

DerivedType
A derived type.

BaseType
The base type of a derived type.

hasImplements
A Boolean value that if `true`, means you can't use a MixIn structure with a class that does not derive from the Implements stucture.

Members

Protected Methods

NAME	DESCRIPTION
ChainInterfaces::CanCastTo	Indicates whether the specified interface ID can be cast to each of the specializations defined by the `ChainInterface` template parameters.
ChainInterfaces::CastToUnknown	Casts the interface pointer of the type defined by the *I0* template parameter to a pointer to `IUnknown`.
ChainInterfaces::FillArrayWithIid	Stores the interface ID defined by the *I0* template parameter into a specified location in a specified array of interface IDs.
ChainInterfaces::Verify	Verifies that each interface defined by template parameters *I0* through *I9* inherits from `IUnknown` and/or `IInspectable`, and that *I0* inherits from *I1* through *I9*.

Protected Constants

NAME	DESCRIPTION
ChainInterfaces::IidCount	The total number of interface IDs contained in the interfaces specified by template parameters *I0* through *I9*.

Inheritance Hierarchy

`I0`

`ChainInterfaces`

Requirements

Header: implements.h

Namespace: Microsoft::WRL

ChainInterfaces::CanCastTo

Indicates whether the specified interface ID can be cast to each of the specializations defined by the non-default template parameters.

```
__forceinline bool CanCastTo(
   REFIID riid,
   _Deref_out_ void **ppv
);
```

Parameters

riid
An interface ID.

ppv
A pointer to the last interface ID that was cast successfully.

Return Value

`true` if all the cast operations succeeded; otherwise, `false`.

ChainInterfaces::CastToUnknown

Casts the interface pointer of the type defined by the *I0* template parameter to a pointer to `IUnknown`.

```
__forceinline IUnknown* CastToUnknown();
```

Return Value

A pointer to `IUnknown`.

ChainInterfaces::FillArrayWithIid

Stores the interface ID defined by the *I0* template parameter into a specified location in a specified array of interface IDs.

```
__forceinline static void FillArrayWithIid(
   _Inout_ unsigned long &index,
   _In_ IID* iids
);
```

Parameters

index
Pointer to an index value into the *iids* array.

iids
An array of interface IDs.

ChainInterfaces::IidCount

The total number of interface IDs contained in the interfaces specified by template parameters *I0* through *I9*.

```
static const unsigned long IidCount = Details::InterfaceTraits<I0>::IidCount +
Details::InterfaceTraits<I1>::IidCount + Details::InterfaceTraits<I2>::IidCount +
Details::InterfaceTraits<I3>::IidCount + Details::InterfaceTraits<I4>::IidCount +
Details::InterfaceTraits<I5>::IidCount + Details::InterfaceTraits<I6>::IidCount +
Details::InterfaceTraits<I7>::IidCount + Details::InterfaceTraits<I8>::IidCount +
Details::InterfaceTraits<I9>::IidCount;
```

Return Value

The total number of interface IDs.

Remarks

Template parameters *I0* and *I1* are required, and parameters *I2* through *I9* are optional. The IID count of each interface is typically 1.

ChainInterfaces::Verify

Verifies that each interface defined by template parameters *I0* through *I9* inherits from `IUnknown` and/or `IInspectable`, and that *I0* inherits from *I1* through *I9*.

```
WRL_NOTHROW __forceinline static void Verify();
```

Remarks

If the verification operation fails, a `static_assert` emits an error message describing the failure.

Template parameters *I0* and *I1* are required, and parameters *I2* through *I9* are optional.

ClassFactory Class

9/21/2022 • 2 minutes to read • Edit Online

Implements the basic functionality of the `IClassFactory` interface.

Syntax

```
template <
    typename I0 = Details::Nil,
    typename I1 = Details::Nil,
    typename I2 = Details::Nil
>
class ClassFactory :
    public Details::RuntimeClass<
        typename Details::InterfaceListHelper<
            IClassFactory,
            I0,
            I1,
            I2,
            Details::Nil
        >::TypeT,
        RuntimeClassFlags<ClassicCom | InhibitWeakReference>,
        false
    >;
```

Parameters

I0

The zeroth interface.

I1

The first interface.

I2

The second interface.

Remarks

Utilize `ClassFactory` to provide a user-defined factory implementation.

The following programming pattern demonstrates how to use the Implements structure to specify more than three interfaces on a class factory.

```
struct MyFactory : ClassFactory<Implements<I1, I2, I3>, I4, I5>
```

Members

Public Constructors

NAME	DESCRIPTION
ClassFactory::ClassFactory	

Public Methods

NAME	DESCRIPTION
ClassFactory::AddRef	Increments the reference count for the current `ClassFactory` object.
ClassFactory::LockServer	Increments or decrements the number of underlying objects that are tracked by the current `ClassFactory` object.
ClassFactory::QueryInterface	Retrieves a pointer to the interface specified by parameter.
ClassFactory::Release	Decrements the reference count for the current `ClassFactory` object.

Inheritance Hierarchy

`I0`

`ChainInterfaces`

`I0`

`RuntimeClassBase`

`ImplementsHelper`

`DontUseNewUseMake`

`RuntimeClassFlags`

`RuntimeClassBaseT`

`RuntimeClass`

`ClassFactory`

Requirements

Header: module.h

Namespace: Microsoft::WRL

ClassFactory::AddRef

Increments the reference count for the current `ClassFactory` object.

```
STDMETHOD_(
   ULONG,
   AddRef
)();
```

Return Value

S_OK if successful; otherwise, an HRESULT that describes the failure.

ClassFactory::ClassFactory

```
WRL_NOTHROW ClassFactory();
```

ClassFactory::LockServer

Increments or decrements the number of underlying objects that are tracked by the current `ClassFactory` object.

```
STDMETHOD(
   LockServer
)(BOOL fLock);
```

Parameters

fLock

`true` to increment the number of tracked objects. `false` to decrement the number of tracked objects.

Return Value

S_OK if successful; otherwise, E_FAIL.

Remarks

`ClassFactory` keeps track of objects in an underlying instance of the Module class.

ClassFactory::QueryInterface

Retrieves a pointer to the interface specified by parameter.

```
STDMETHOD(
   QueryInterface
)(REFIID riid, _Deref_out_ void **ppvObject);
```

Parameters

riid

An interface ID.

ppvObject

When this operation completes, a pointer to the interface specified by parameter *riid*.

Return Value

S_OK if successful; otherwise, an HRESULT that describes the failure.

ClassFactory::Release

Decrements the reference count for the current `ClassFactory` object.

```
STDMETHOD_(
   ULONG,
   Release
)();
```

Return Value

S_OK if successful; otherwise, an HRESULT that describes the failure.

CloakedIid Structure

9/21/2022 • 2 minutes to read • Edit Online

Indicates to the `RuntimeClass`, `Implements` and `ChainInterfaces` templates that the specified interface is not accessible in the IID list.

Syntax

```
template<typename T>
struct CloakedIid : T;
```

Parameters

T

The interface that is hidden (cloaked).

Remarks

The following is an example of how **CloakedIid** is used:

`struct MyRuntimeClass : RuntimeClass<CloakedIid<IMyCloakedInterface>> {}`.

Inheritance Hierarchy

`T`

`CloakedIid`

Requirements

Header: implements.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

ComposableBase Class

9/21/2022 • 2 minutes to read • Edit Online

Syntax

```
template<typename FactoryInterface = IInspectable>
class ComposableBase;
```

Parameters

FactoryInterface

Requirements

Header: implements.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

ComPtr Class

9/21/2022 • 9 minutes to read • Edit Online

Creates a *smart pointer* type that represents the interface specified by the template parameter. `ComPtr` automatically maintains a reference count for the underlying interface pointer and releases the interface when the reference count goes to zero.

Syntax

```
template <typename T>
class ComPtr;

template<class U>
friend class ComPtr;
```

Parameters

`T`
The interface that the `ComPtr` represents.

`U`
A class to which the current `ComPtr` is a friend. (The template that uses this parameter is protected.)

Remarks

`ComPtr<>` declares a type that represents the underlying interface pointer. Use `ComPtr<>` to declare a variable and then use the arrow member-access operator (`->`) to access an interface member function.

For more information about smart pointers, see the "COM Smart Pointers" subsection of the COM Coding Practices article.

Members

Public Typedefs

NAME	DESCRIPTION
`InterfaceType`	A synonym for the type specified by the `T` template parameter.

Public Constructors

NAME	DESCRIPTION
`ComPtr::ComPtr`	Initializes a new instance of the `ComPtr` class. Overloads provide default, copy, move, and conversion constructors.
`ComPtr::~ComPtr`	Deinitializes an instance of `ComPtr`.

Public Methods

NAME	DESCRIPTION
`ComPtr::As`	Returns a `ComPtr` object that represents the interface identified by the specified template parameter.
`ComPtr::AsIID`	Returns a `ComPtr` object that represents the interface identified by the specified interface ID.
`ComPtr::AsWeak`	Retrieves a weak reference to the current object.
`ComPtr::Attach`	Associates this `ComPtr` with the interface type specified by the current template type parameter.
`ComPtr::CopyTo`	Copies the current or specified interface associated with this `ComPtr` to the specified output pointer.
`ComPtr::Detach`	Disassociates this `ComPtr` from the interface that it represents.
`ComPtr::Get`	Retrieves a pointer to the interface that is associated with this `ComPtr`.
`ComPtr::GetAddressOf`	Retrieves the address of the `ptr_` data member, which contains a pointer to the interface represented by this `ComPtr`.
`ComPtr::ReleaseAndGetAddressOf`	Releases the interface associated with this `ComPtr` and then retrieves the address of the `ptr_` data member, which contains a pointer to the interface that was released.
`ComPtr::Reset`	Releases the interface associated with this `ComPtr` and returns the new reference count.
`ComPtr::Swap`	Exchanges the interface managed by the current `ComPtr` with the interface managed by the specified `ComPtr`.

Protected Methods

NAME	DESCRIPTION
`ComPtr::InternalAddRef`	Increments the reference count of the interface associated with this `ComPtr`.
`ComPtr::InternalRelease`	Performs a COM Release operation on the interface associated with this `ComPtr`.

Public Operators

NAME	DESCRIPTION
`ComPtr::operator&`	Retrieves the address of the current `ComPtr`.
`ComPtr::operator->`	Retrieves a pointer to the type specified by the current template parameter.

NAME	DESCRIPTION
`ComPtr::operator=`	Assigns a value to the current `ComPtr`.
`ComPtr::operator==`	Indicates whether two `ComPtr` objects are equal.
`ComPtr::operator!=`	Indicates whether two `ComPtr` objects aren't equal.
`ComPtr::operator Microsoft::WRL::Details::BoolType`	Indicates whether a `ComPtr` is managing the object lifetime of an interface.

Protected Data Members

NAME	DESCRIPTION
`ComPtr::ptr_`	Contains a pointer to the interface that is associated with, and managed by this `ComPtr`.

Inheritance Hierarchy

`ComPtr`

Requirements

Header: *`client.h`*

Namespace: `Microsoft::WRL`

`ComPtr::~ComPtr`

Deinitializes an instance of `ComPtr`.

```
WRL_NOTHROW ~ComPtr();
```

`ComPtr::As`

Returns a `ComPtr` object that represents the interface identified by the specified template parameter.

```
template<typename U>
HRESULT As(
   _Out_ ComPtr<U>* p
) const;

template<typename U>
HRESULT As(
   _Out_ Details::ComPtrRef<ComPtr<U>> p
) const;
```

Parameters

`U`

The interface to be represented by parameter *`p`*.

`p`

A `ComPtr` object that represents the interface specified by parameter `U`. Parameter `p` must not refer to the current `ComPtr` object.

Remarks

The first template is the form that you should use in your code. The second template is an internal, helper specialization. It supports C++ language features such as the `auto` type deduction keyword.

Return Value

`S_OK` if successful; otherwise, an `HRESULT` that indicates the error.

ComPtr::AsIID

Returns a `ComPtr` object that represents the interface identified by the specified interface ID.

```
WRL_NOTHROW HRESULT AsIID(
   REFIID riid,
   _Out_ ComPtr<IUnknown>* p
) const;
```

Parameters

riid
An interface ID.

p
If the object has an interface whose ID equals *riid*, a doubly indirect pointer to the interface specified by the *riid* parameter. Otherwise, a pointer to `IUnknown`.

Return Value

`S_OK` if successful; otherwise, an `HRESULT` that indicates the error.

ComPtr::AsWeak

Retrieves a weak reference to the current object.

```
HRESULT AsWeak(
   _Out_ WeakRef* pWeakRef
);
```

Parameters

pWeakRef
When this operation completes, a pointer to a weak reference object.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

ComPtr::Attach

Associates this `ComPtr` with the interface type specified by the current template type parameter.

```
void Attach(
   _In_opt_ InterfaceType* other
);
```

Parameters

other
An interface type.

ComPtr::ComPtr

Initializes a new instance of the ComPtr class. Overloads provide default, copy, move, and conversion constructors.

```
WRL_NOTHROW ComPtr();

WRL_NOTHROW ComPtr(
   decltype(__nullptr)
);

template<class U>
WRL_NOTHROW ComPtr(
   _In_opt_ U *other
);

WRL_NOTHROW ComPtr(
   const ComPtr& other
);

template<class U>
WRL_NOTHROW ComPtr(
   const ComPtr<U> &other,
   typename ENABLE_IF<__is_convertible_to(U*, T*), void *>
);

WRL_NOTHROW ComPtr(
   _Inout_ ComPtr &&other
);

template<class U>
WRL_NOTHROW ComPtr(
   _Inout_ ComPtr<U>&& other, typename ENABLE_IF<__is_convertible_to(U*, T*), void *>
);
```

Parameters

U
The type of the *other* parameter.

other
An object of type *U*.

Return Value

Remarks

The first constructor is the default constructor, which implicitly creates an empty object. The second constructor specifies __nullptr, which explicitly creates an empty object.

The third constructor creates an object from the object specified by a pointer. The ComPtr now owns the pointed-to memory and maintains a reference count to it.

The fourth and fifth constructors are copy constructors. The fifth constructor copies an object if it's convertible to the current type.

The sixth and seventh constructors are move constructors. The seventh constructor moves an object if it's convertible to the current type.

`ComPtr::CopyTo`

Copies the current or specified interface associated with this `ComPtr` to the specified pointer.

```
HRESULT CopyTo(
   _Deref_out_ InterfaceType** ptr
);

HRESULT CopyTo(
   REFIID riid,
   _Deref_out_ void** ptr
) const;

template<typename U>
HRESULT CopyTo(
   _Deref_out_ U** ptr
) const;
```

Parameters

`U`
A type name.

`ptr`
When this operation completes, a pointer to the requested interface.

`riid`
An interface ID.

Return Value

`S_OK` if successful; otherwise, an `HRESULT` that indicates why the implicit `QueryInterface` operation failed.

Remarks

The first function returns a copy of a pointer to the interface associated with this `ComPtr`. This function always returns `S_OK`.

The second function performs a `QueryInterface` operation on the interface associated with this `ComPtr` for the interface specified by the *`riid`* parameter.

The third function performs a `QueryInterface` operation on the interface associated with this `ComPtr` for the underlying interface of the *`U`* parameter.

`ComPtr::Detach`

Disassociates this `ComPtr` object from the interface that it represents.

```
T* Detach();
```

Return Value

A pointer to the interface that was represented by this `ComPtr` object.

`ComPtr::Get`

Retrieves a pointer to the interface that is associated with this `ComPtr`.

```
T* Get() const;
```

Return Value

Pointer to the interface that is associated with this `ComPtr`.

`ComPtr::GetAddressOf`

Retrieves the address of the `ptr_` data member, which contains a pointer to the interface represented by this `ComPtr`.

```
T* const* GetAddressOf() const;
T** GetAddressOf();
```

Return Value

The address of a variable.

`ComPtr::InternalAddRef`

Increments the reference count of the interface associated with this `ComPtr`.

```
void InternalAddRef() const;
```

Remarks

This method is protected.

`ComPtr::InternalRelease`

Performs a COM Release operation on the interface associated with this `ComPtr`.

```
unsigned long InternalRelease();
```

Remarks

This method is protected.

`ComPtr::operator&`

Releases the interface associated with this `ComPtr` object and then retrieves the address of the `ComPtr` object.

```
Details::ComPtrRef<WeakRef> operator&()

const Details::ComPtrRef<const WeakRef> operator&() const
```

Return Value

A weak reference to the current `ComPtr`.

Remarks

This method differs from `ComPtr::GetAddressOf` in that this method releases a reference to the interface pointer. Use `ComPtr::GetAddressOf` when you require the address of the interface pointer but don't want to release that interface.

`ComPtr::operator->`

Retrieves a pointer to the type specified by the current template parameter.

```
WRL_NOTHROW Microsoft::WRL::Details::RemoveIUnknown<InterfaceType>* operator->() const;
```

Return Value

Pointer to the type specified by the current template type name.

Remarks

This helper function removes unnecessary overhead caused by using the STDMETHOD macro. This function makes `IUnknown` types `private` instead of `virtual`.

`ComPtr::operator=`

Assigns a value to the current `ComPtr`.

```
WRL_NOTHROW ComPtr& operator=(
   decltype(__nullptr)
);
WRL_NOTHROW ComPtr& operator=(
   _In_opt_ T *other
);
template <typename U>
WRL_NOTHROW ComPtr& operator=(
   _In_opt_ U *other
);
WRL_NOTHROW ComPtr& operator=(
   const ComPtr &other
);
template<class U>
WRL_NOTHROW ComPtr& operator=(
   const ComPtr<U>& other
);
WRL_NOTHROW ComPtr& operator=(
   _Inout_ ComPtr &&other
);
template<class U>
WRL_NOTHROW ComPtr& operator=(
   _Inout_ ComPtr<U>&& other
);
```

Parameters

`U`
A class.

`other`
A pointer, reference, or rvalue reference to a type or another `ComPtr`.

Return Value

A reference to the current `ComPtr`.

Remarks

The first version of this operator assigns an empty value to the current `ComPtr`.

In the second version, if the assigning interface pointer isn't the same as the current `ComPtr` interface pointer,

the second interface pointer is assigned to the current `ComPtr`.

In the third version, the assigning interface pointer is assigned to the current `ComPtr`.

In the fourth version, if the interface pointer of the assigning value isn't the same as the current `ComPtr` interface pointer, the second interface pointer is assigned to the current `ComPtr`.

The fifth version is a copy operator; a reference to a `ComPtr` is assigned to the current `ComPtr`.

The sixth version is a copy operator that uses move semantics; an rvalue reference to a `ComPtr` if any type is static cast and then assigned to the current `ComPtr`.

The seventh version is a copy operator that uses move semantics; an rvalue reference to a `ComPtr` of type `U` is static cast then and assigned to the current `ComPtr`.

`ComPtr::operator==`

Indicates whether two `ComPtr` objects are equal.

```
bool operator==(
   const ComPtr<T>& a,
   const ComPtr<U>& b
);

bool operator==(
   const ComPtr<T>& a,
   decltype(__nullptr)
);

bool operator==(
   decltype(__nullptr),
   const ComPtr<T>& a
);
```

Parameters

a
A reference to a `ComPtr` object.

b
A reference to another `ComPtr` object.

Return Value

The first operator yields `true` if object *a* is equal to object *b*; otherwise, `false`.

The second and third operators yield `true` if object *a* is equal to `nullptr`; otherwise, `false`.

`ComPtr::operator!=`

Indicates whether two `ComPtr` objects aren't equal.

```
bool operator!=(
   const ComPtr<T>& a,
   const ComPtr<U>& b
);

bool operator!=(
   const ComPtr<T>& a,
   decltype(__nullptr)
);

bool operator!=(
   decltype(__nullptr),
   const ComPtr<T>& a
);
```

Parameters

a
A reference to a `ComPtr` object.

b
A reference to another `ComPtr` object.

Return Value

The first operator yields `true` if object *a* isn't equal to object *b*; otherwise, `false`.

The second and third operators yield `true` if object *a* isn't equal to `nullptr`; otherwise, `false`.

ComPtr::operator Microsoft::WRL::Details::BoolType

Indicates whether a `ComPtr` is managing the object lifetime of an interface.

```
WRL_NOTHROW operator Microsoft::WRL::Details::BoolType() const;
```

Return Value

If an interface is associated with this `ComPtr`, the address of the `BoolStruct::Member` data member; otherwise, `nullptr`.

ComPtr::ptr_

Contains a pointer to the interface that is associated with, and managed by this `ComPtr`.

```
InterfaceType *ptr_;
```

Remarks

`ptr_` is an internal, protected data member.

ComPtr::ReleaseAndGetAddressOf

Releases the interface associated with this `ComPtr` and then retrieves the address of the `ptr_` data member, which contains a pointer to the interface that was released.

```
T** ReleaseAndGetAddressOf();
```

Return Value

The address of the `ptr_` data member of this `ComPtr`.

`ComPtr::Reset`

Releases the interface associated with this `ComPtr` and returns the new reference count.

```
unsigned long Reset();
```

Return Value

The number of references remaining to the underlying interface, if any.

`ComPtr::Swap`

Exchanges the interface managed by the current `ComPtr` with the interface managed by the specified `ComPtr`.

```
void Swap(
   _Inout_ ComPtr&& r
);

void Swap(
   _Inout_ ComPtr& r
);
```

Parameters

r
A `ComPtr`.

CreateActivationFactory Function

9/21/2022 • 2 minutes to read • Edit Online

Creates a factory that produces instances of the specified class that can be activated by the Windows Runtime.

Syntax

```
template<typename Factory>
   inline HRESULT STDMETHODCALLTYPE CreateActivationFactory(
      _In_ unsigned int *flags,        _In_ const CreatorMap* entry,
      REFIID riid,
   _Outptr_ IUnknown **ppFactory) throw();
```

Parameters

flags

A combination of one or more RuntimeClassType enumeration values.

entry

Pointer to a CreatorMap that contains initialization and registration information about parameter *riid*.

riid

Reference to an interface ID.

ppFactory

If this operation completes successfully, a pointer to an activation factory.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

Remarks

An assert error is emitted if template parameter *Factory* doesn't derive from interface `IActivationFactory`.

Requirements

Header: module.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL::Wrappers::Details Namespace

CreateClassFactory Function

9/21/2022 • 2 minutes to read • Edit Online

Creates a factory that produces instances of the specified class.

Syntax

```
template<typename Factory>
inline HRESULT STDMETHODCALLTYPE CreateClassFactory(
   _In_ unsigned int *flags,
   _In_ const CreatorMap* entry,
   REFIID riid,
   _Outptr_ IUnknown **ppFactory
) throw();
```

Parameters

flags
A combination of one or more RuntimeClassType enumeration values.

entry
Pointer to a CreatorMap that contains initialization and registration information about parameter *riid*.

riid
Reference to an interface ID.

ppFactory
If this operation completes successfully, a pointer to a class factory.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

Remarks

An assert error is emitted if template parameter *Factory* doesn't derive from interface `IClassFactory`.

Requirements

Header: module.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL::Wrappers::Details Namespace

DeferrableEventArgs Class

9/21/2022 • 2 minutes to read • Edit Online

A template class used for the event argument types for deferrals.

Syntax

```
template <typename TEventArgsInterface, typename TEventArgsClass>
class DeferrableEventArgs : public TEventArgsInterface;
```

Parameters

TEventArgsInterface
The interface type that declares the arguments for a deferred event.

TEventArgsClass
The class that implements *TEventArgsInterface*.

Members

Public Methods

NAME	DESCRIPTION
DeferrableEventArgs::GetDeferral	Gets a reference to the Deferral object which represents a deferred event.
DeferrableEventArgs::InvokeAllFinished	Called to indicate that all processing to handle a deferred event is complete.

Remarks

Instances of this class are passed to event handlers for deferred events. The template parameters represent an interface that defines the details of the event arguments for a specific type of deferred event, and a class that implements that interface.

The class appears as the first argument to an event handler for a deferred event. You can call the GetDeferral method to get the Deferral object from which you can get all the information about the deferred event. After completing the event handling, you should call Complete on the Deferral object. You should then call InvokeAllFinished at the end of the event handler method, which ensures that the completion of all deferred events is communicated properly.

Requirements

Header: event.h

Namespace: Microsoft::WRL

DeferrableEventArgs::GetDeferral

Gets a reference to the Deferral object which represents a deferred event.

```
HRESULT GetDeferral([out, retval] Windows::Foundation::IDeferral** result)
```

Parameters

result
A pointer that will reference the Deferral object when the call completes.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

DeferrableEventArgs::InvokeAllFinished

Called to indicate that all processing to handle a deferred event is complete.

```
void InvokeAllFinished()
```

Remarks

You should call this method after the event source calls InvokeAll. Calling this method prevents further deferrals from being taken and forces the completion handler to execute if no deferrals were taken.

EventSource Class

9/21/2022 • 4 minutes to read • Edit Online

Represents a non-agile event. `EventSource` member functions add, remove, and invoke event handlers. For agile events, use AgileEventSource.

Syntax

```
template<typename TDelegateInterface>
class EventSource;
```

Parameters

TDelegateInterface
The interface to a delegate that represents an event handler.

Members

Public Constructors

NAME	DESCRIPTION
EventSource::EventSource	Initializes a new instance of the `EventSource` class.

Public Methods

NAME	DESCRIPTION
EventSource::Add	Appends the event handler represented by the specified delegate interface to the set of event handlers for the current `EventSource` object.
EventSource::GetSize	Retrieves the number of event handlers associated with the current `EventSource` object.
EventSource::InvokeAll	Calls each event handler associated with the current `EventSource` object using the specified argument types and arguments.
EventSource::Remove	Deletes the event handler represented by the specified event registration token from the set of event handlers associated with the current `EventSource` object.

Protected Data Members

NAME	DESCRIPTION
EventSource::addRemoveLock_	Synchronizes access to the targets_ array when adding, removing, or invoking event handlers.
EventSource::targets_	An array of one or more event handlers.

NAME	DESCRIPTION
EventSource::targetsPointerLock_	Synchronizes access to internal data members even while event handlers for this EventSource are being added, removed, or invoked.

Inheritance Hierarchy

`EventSource`

Requirements

Header: event.h

Namespace: Microsoft::WRL

EventSource::Add

Appends the event handler represented by the specified delegate interface to the set of event handlers for the current `EventSource` object.

```
HRESULT Add(
   _In_ TDelegateInterface* delegateInterface,
   _Out_ EventRegistrationToken* token
);
```

Parameters

delegateInterface
The interface to a delegate object, which represents an event handler.

token
When this operation completes, a handle that represents the event. Use this token as the parameter to the Remove() method to discard the event handler.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

EventSource::addRemoveLock_

Synchronizes access to the targets_ array when adding, removing, or invoking event handlers.

```
Wrappers::SRWLock addRemoveLock_;
```

EventSource::EventSource

Initializes a new instance of the `EventSource` class.

```
EventSource();
```

EventSource::GetSize

Retrieves the number of event handlers associated with the current `EventSource` object.

```
size_t GetSize() const;
```

Return Value

The number of event handlers in targets_.

EventSource::InvokeAll

Calls each event handler associated with the current `EventSource` object using the specified argument types and arguments.

```
void InvokeAll();
template <
   typename T0
>
void InvokeAll(
   T0arg0
);
template <
   typename T0,
   typename T1
>
void InvokeAll(
   T0arg0,
   T1arg1
);
template <
   typename T0,
   typename T1,
   typename T2
>
void InvokeAll(
   T0arg0,
   T1arg1,
   T2arg2
);
template <
   typename T0,
   typename T1,
   typename T2,
   typename T3
>
void InvokeAll(
   T0arg0,
   T1arg1,
   T2arg2,
   T3arg3
);
template <
   typename T0,
   typename T1,
   typename T2,
   typename T3,
   typename T4
>
void InvokeAll(
   T0arg0,
   T1arg1,
   T2arg2,
   T3arg3,
   T4arg4
);
template <
   typename T0,
   typename T1,
```

```
   typename T2,
   typename T3,
   typename T4,
   typename T5
>
void InvokeAll(
   T0arg0,
   T1arg1,
   T2arg2,
   T3arg3,
   T4arg4,
   T5arg5
);
template <
   typename T0,
   typename T1,
   typename T2,
   typename T3,
   typename T4,
   typename T5,
   typename T6
>
void InvokeAll(
   T0arg0,
   T1arg1,
   T2arg2,
   T3arg3,
   T4arg4,
   T5arg5,
   T6arg6
);
template <
   typename T0,
   typename T1,
   typename T2,
   typename T3,
   typename T4,
   typename T5,
   typename T6,
   typename T7
>
void InvokeAll(
   T0arg0,
   T1arg1,
   T2arg2,
   T3arg3,
   T4arg4,
   T5arg5,
   T6arg6,
   T7arg7
);
template <
   typename T0,
   typename T1,
   typename T2,
   typename T3,
   typename T4,
   typename T5,
   typename T6,
   typename T7,
   typename T8
>
void InvokeAll(
   T0arg0,
   T1arg1,
   T2arg2,
   T3arg3,
   T4arg4,
   T5arg5,
```

```
    T6arg6,
    T7arg7,
    T8arg8
);
template <
    typename T0,
    typename T1,
    typename T2,
    typename T3,
    typename T4,
    typename T5,
    typename T6,
    typename T7,
    typename T8,
    typename T9
>
void InvokeAll(
    T0arg0,
    T1arg1,
    T2arg2,
    T3arg3,
    T4arg4,
    T5arg5,
    T6arg6,
    T7arg7,
    T8arg8,
    T9arg9
);
```

Parameters

T0
The type of the zeroth event handler argument.

T1
The type of the first event handler argument.

T2
The type of the second event handler argument.

T3
The type of the third event handler argument.

T4
The type of the fourth event handler argument.

T5
The type of the fifth event handler argument.

T6
The type of the sixth event handler argument.

T7
The type of the seventh event handler argument.

T8
The type of the eighth event handler argument.

T9
The type of the ninth event handler argument.

arg0
The zeroth event handler argument.

arg1
The first event handler argument.

arg2
The second event handler argument.

arg3
The third event handler argument.

arg4
The fourth event handler argument.

arg5
The fifth event handler argument.

arg6
The sixth event handler argument.

arg7
The seventh event handler argument.

arg8
The eighth event handler argument.

arg9
The ninth event handler argument.

EventSource::Remove

Deletes the event handler represented by the specified event registration token from the set of event handlers associated with the current `EventSource` object.

```
HRESULT Remove(
   EventRegistrationToken token
);
```

Parameters

token
A handle that represents an event handler. This token was returned when the event handler was registered by the Add() method.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

Remarks

For more information about the `EventRegistrationToken` structure, see the **Windows::Foundation::EventRegistrationToken Structure** topic in the **Windows Runtime** reference documentation.

EventSource::targets_

An array of one or more event handlers.

```
ComPtr<Details::EventTargetArray> targets_;
```

Remarks

When the event that is represented by the current `EventSource` object occurs, the event handlers are called.

EventSource::targetsPointerLock_

Synchronizes access to internal data members even while event handlers for this `EventSource` are being added, removed, or invoked.

```
Wrappers::SRWLock targetsPointerLock_;
```

FactoryCacheFlags Enumeration

9/21/2022 • 2 minutes to read • Edit Online

Determines whether factory objects are cached.

Syntax

```
enum FactoryCacheFlags;
```

Remarks

By default, the factory caching policy is specified as the ModuleType template parameter when you create a Module object. To override this policy, specify a **FactoryCacheFlags** value when you create a factory object.

POLICY	DESCRIPTION
`FactoryCacheDefault`	The caching policy of the `Module` object is used.
`FactoryCacheEnabled`	Enables factory caching regardless of the `ModuleType` template parameter that is used to create a `Module` object.
`FactoryCacheDisabled`	Disables factory caching regardless of the `ModuleType` template parameter that is used to create a `Module` object.

Requirements

Header: implements.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

FtmBase Class

9/21/2022 • 4 minutes to read • Edit Online

Represents a free-threaded marshaler object.

Syntax

```
class FtmBase :
    public Microsoft::WRL::Implements<
        Microsoft::WRL::RuntimeClassFlags<WinRtClassicComMix>,
        Microsoft::WRL::CloakedIid<IMarshal>
    >;
```

Remarks

For more information, see RuntimeClass Class.

Members

Public Constructors

NAME	DESCRIPTION
FtmBase::FtmBase	Initializes a new instance of the `FtmBase` class.

Public Methods

NAME	DESCRIPTION
FtmBase::CreateGlobalInterfaceTable	Creates a global interface table (GIT).
FtmBase::DisconnectObject	Forcibly releases all external connections to an object. The object's server calls the object's implementation of this method prior to shutting down.
FtmBase::GetMarshalSizeMax	Get the upper bound on the number of bytes needed to marshal the specified interface pointer on the specified object.
FtmBase::GetUnmarshalClass	Gets the CLSID that COM uses to locate the DLL containing the code for the corresponding proxy. COM loads this DLL to create an uninitialized instance of the proxy.
FtmBase::MarshalInterface	Writes into a stream the data required to initialize a proxy object in some client process.
FtmBase::ReleaseMarshalData	Destroys a marshaled data packet.
FtmBase::UnmarshalInterface	Initializes a newly created proxy and returns an interface pointer to that proxy.

Public Data Members

NAME	DESCRIPTION
FtmBase::marshaller_	Holds a reference to the free threaded marshaler.

Inheritance Hierarchy

`FtmBase`

Requirements

Header: ftm.h

Namespace: Microsoft::WRL

FtmBase::CreateGlobalInterfaceTable

Creates a global interface table (GIT).

```
static HRESULT CreateGlobalInterfaceTable(
   __out IGlobalInterfaceTable **git
);
```

Parameters

git
When this operation completes, a pointer to a global interface table.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

Remarks

For more information, see `IGlobalInterfaceTable`.

FtmBase::DisconnectObject

Forcibly releases all external connections to an object. The object's server calls the object's implementation of this method prior to shutting down.

```
STDMETHODIMP DisconnectObject(
   __in DWORD dwReserved
) override;
```

Parameters

dwReserved
Reserved for future use; must be zero.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

FtmBase::FtmBase

Initializes a new instance of the `FtmBase` class.

```
FtmBase();
```

FtmBase::GetMarshalSizeMax

Get the upper bound on the number of bytes needed to marshal the specified interface pointer on the specified object.

```
STDMETHODIMP GetMarshalSizeMax(
   __in REFIID riid,
   __in_opt void *pv,
   __in DWORD dwDestContext,
   __reserved void *pvDestContext,
   __in DWORD mshlflags,
   __out DWORD *pSize
) override;
```

Parameters

riid
Reference to the identifier of the interface to be marshaled.

pv
Interface pointer to be marshaled; can be NULL.

dwDestContext
Destination context where the specified interface is to be unmarshaled.

Specify one or more MSHCTX enumeration values.

Currently, unmarshaling can occur either in another apartment of the current process (MSHCTX_INPROC) or in another process on the same computer as the current process (MSHCTX_LOCAL).

pvDestContext
Reserved for future use; must be NULL.

mshlflags
Flag indicating whether the data to be marshaled is to be transmitted back to the client process — the typical case — or written to a global table, where it can be retrieved by multiple clients. Specify one or more MSHLFLAGS enumeration values.

pSize
When this operation completes, pointer to the upper bound on the amount of data to be written to the marshaling stream.

Return Value

S_OK if successful; otherwise, E_FAIL or E_NOINTERFACE.

FtmBase::GetUnmarshalClass

Gets the CLSID that COM uses to locate the DLL containing the code for the corresponding proxy. COM loads this DLL to create an uninitialized instance of the proxy.

```
STDMETHODIMP GetUnmarshalClass(
   __in REFIID riid,
   __in_opt void *pv,
   __in DWORD dwDestContext,
   __reserved void *pvDestContext,
   __in DWORD mshlflags,
   __out CLSID *pCid
) override;
```

Parameters

riid
Reference to the identifier of the interface to be marshaled.

pv
Pointer to the interface to be marshaled; can be NULL if the caller does not have a pointer to the desired interface.

dwDestContext
Destination context where the specified interface is to be unmarshaled.

Specify one or more MSHCTX enumeration values.

Unmarshaling can occur either in another apartment of the current process (MSHCTX_INPROC) or in another process on the same computer as the current process (MSHCTX_LOCAL).

pvDestContext
Reserved for future use; must be NULL.

mshlflags
When this operation completes, pointer to the CLSID to be used to create a proxy in the client process.

pCid

Return Value

S_OK if successful; otherwise, S_FALSE.

FtmBase::MarshalInterface

Writes into a stream the data required to initialize a proxy object in some client process.

```
STDMETHODIMP MarshalInterface(
   __in IStream *pStm,
   __in REFIID riid,
   __in_opt void *pv,
   __in DWORD dwDestContext,
   __reserved void *pvDestContext,
   __in DWORD mshlflags
) override;
```

Parameters

pStm
Pointer to the stream to be used during marshaling.

riid
Reference to the identifier of the interface to be marshaled. This interface must be derived from the `IUnknown` interface.

pv

Pointer to the interface pointer to be marshaled; can be NULL if the caller does not have a pointer to the desired interface.

dwDestContext
Destination context where the specified interface is to be unmarshaled.

Specify one or more MSHCTX enumeration values.

Unmarshaling can occur in another apartment of the current process (MSHCTX_INPROC) or in another process on the same computer as the current process (MSHCTX_LOCAL).

pvDestContext
Reserved for future use; must be zero.

mshlflags
Specifies whether the data to be marshaled is to be transmitted back to the client process — the typical case — or written to a global table, where it can be retrieved by multiple clients.

Return Value

S_OK The interface pointer was marshaled successfully.

E_NOINTERFACE The specified interface is not supported.

STG_E_MEDIUMFULL The stream is full.

E_FAIL The operation failed.

FtmBase::marshaller_

Holds a reference to the free threaded marshaler.

```
Microsoft::WRL::ComPtr<IMarshal> marshaller_; ;
```

FtmBase::ReleaseMarshalData

Destroys a marshaled data packet.

```
STDMETHODIMP ReleaseMarshalData(
   __in IStream *pStm
) override;
```

Parameters

pStm
Pointer to a stream that contains the data packet to be destroyed.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

FtmBase::UnmarshalInterface

Initializes a newly created proxy and returns an interface pointer to that proxy.

```
STDMETHODIMP UnmarshalInterface(
   __in IStream *pStm,
   __in REFIID riid,
   __deref_out void **ppv
) override;
```

Parameters

pStm
Pointer to the stream from which the interface pointer is to be unmarshaled.

riid
Reference to the identifier of the interface to be unmarshaled.

ppv
When this operation completes, the address of a pointer variable that receives the interface pointer requested in *riid*. If this operation is successful, **ppv* contains the requested interface pointer of the interface to be unmarshaled.

Return Value

S_OK if successful; otherwise, E_NOINTERFACE or E_FAIL.

GetModuleBase function

9/21/2022 • 2 minutes to read • Edit Online

Retrieves a ModuleBase pointer that allows for incrementing and decrementing the reference count of a RuntimeClass object.

Syntax

```
inline Details::ModuleBase* GetModuleBase() throw()
```

Return value

A pointer to a `ModuleBase` object.

Remarks

This function is used internally to increment and decrement object reference counts.

You can use this function to control reference counts by calling ModuleBase::IncrementObjectCount and ModuleBase::DecrementObjectCount.

Requirements

Header: implements.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

Implements Structure

9/21/2022 • 2 minutes to read • Edit Online

Implements `QueryInterface` and `GetIid` for the specified interfaces.

Syntax

```
template <
    typename I0,
    typename I1 = Details::Nil,
    typename I2 = Details::Nil,
    typename I3 = Details::Nil,
    typename I4 = Details::Nil,
    typename I5 = Details::Nil,
    typename I6 = Details::Nil,
    typename I7 = Details::Nil,
    typename I8 = Details::Nil,
    typename I9 = Details::Nil
>
struct __declspec(novtable) Implements :
    Details::ImplementsHelper<
        RuntimeClassFlags<WinRt>,
        typename Details::InterfaceListHelper<
            I0, I1, I2, I3, I4, I5, I6, I7, I8, I9
        >::TypeT
    >,
    Details::ImplementsBase;

template <
    int flags,
    typename I0,
    typename I1,
    typename I2,
    typename I3,
    typename I4,
    typename I5,
    typename I6,
    typename I7,
    typename I8
>
struct __declspec(novtable) Implements<
        RuntimeClassFlags<flags>,
        I0, I1, I2, I3, I4, I5, I6, I7, I8> :
    Details::ImplementsHelper<
        RuntimeClassFlags<flags>,
        typename Details::InterfaceListHelper<
            I0, I1, I2, I3, I4, I5, I6, I7, I8
        >::TypeT
    >,
    Details::ImplementsBase;
```

Parameters

I0

The zeroth interface ID. (Mandatory)

I1

The first interface ID. (Optional)

I2
The second interface ID. (Optional)

I3
The third interface ID. (Optional)

I4
The fourth interface ID. (Optional)

I5
The fifth interface ID. (Optional)

I6
The sixth interface ID. (Optional)

I7
The seventh interface ID. (Optional)

I8
The eighth interface ID. (Optional)

I9
The ninth interface ID. (Optional)

flags
Configuration flags for the class. One or more RuntimeClassType enumerations that are specified in a RuntimeClassFlags structure.

Remarks

Derives from the list of specified interfaces and implements helper templates for `QueryInterface` and `GetIid`.

Each *I0* through *I9* interface parameter must derive from either `IUnknown`, `IInspectable`, or the ChainInterfaces template. The *flags* parameter determines whether support is generated for `IUnknown` or `IInspectable`.

Members

Public Typedefs

NAME	DESCRIPTION
`ClassFlags`	A synonym for `RuntimeClassFlags<WinRt>`.

Protected Methods

NAME	DESCRIPTION
Implements::CanCastTo	Gets a pointer to the specified interface.
Implements::CastToUnknown	Gets a pointer to the underlying `IUnknown` interface.
Implements::FillArrayWithIid	Inserts the interface ID specified by the current zeroth template parameter into the specified array element.

Protected Constants

NAME	DESCRIPTION
Implements::IidCount	Holds the number of implemented interface IDs.

Inheritance Hierarchy

`I0`

`ChainInterfaces`

`I0`

`ImplementsBase`

`ImplementsHelper`

`Implements`

Requirements

Header: implements.h

Namespace: Microsoft::WRL

Implements::CanCastTo

Gets a pointer to the specified interface.

```
__forceinline HRESULT CanCastTo(
   REFIID riid,
   _Deref_out_ void **ppv
);
```

Parameters

riid
A reference to an interface ID.

ppv
If successful, a pointer to the interface specified by *riid*.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error, such as E_NOINTERFACE.

Remarks

This is an internal helper function that performs a QueryInterface operation.

Implements::CastToUnknown

Gets a pointer to the underlying `IUnknown` interface.

```
__forceinline IUnknown* CastToUnknown();
```

Return Value

This operation always succeeds and returns the `IUnknown` pointer.

Remarks

Internal helper function.

Implements::FillArrayWithIid

Inserts the interface ID specified by the current zeroth template parameter into the specified array element.

```
__forceinline static void FillArrayWithIid(
   unsigned long &index,
   _In_ IID* iids
);
```

Parameters

index
A zero-based index that indicates the starting array element for this operation. When this operation completes, *index* is incremented by 1.

iids
An array of type IID.

Remarks

Internal helper function.

Implements::IidCount

Holds the number of implemented interface IDs.

```
static const unsigned long IidCount;
```

InspectableClass Macro

9/21/2022 • 2 minutes to read • Edit Online

Sets the runtime class name and trust level.

Syntax

```
InspectableClass(
   runtimeClassName,
   trustLevel)
```

Parameters

runtimeClassName
The full textual name of the runtime class.

trustLevel
One of the TrustLevel enumerated values.

Remarks

The **InspectableClass** macro can be used only with Windows Runtime types.

Requirements

Header: implements.h

Namespace: Microsoft::WRL

See also

RuntimeClass Class

InvokeModeOptions Structure

9/21/2022 • 2 minutes to read • Edit Online

Specifies whether to fire all events in the delegate queue, or to stop firing after an error is raised. The allowable values are specified in the `InvokeMode` enum.

Syntax

```
enum InvokeMode
{
   StopOnFirstError = 1,
   FireAll = 2,
};

struct InvokeModeOptions
{
   static const InvokeMode invokeMode = invokeModeValue;
};
```

Requirements

Header: event.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace
Microsoft::WRL::AgileEventSource Class

Make Function

9/21/2022 • 2 minutes to read • Edit Online

Initializes the specified Windows Runtime class. Use this function to instantiate a component that is defined in the same module.

Syntax

```
template <
   typename T,
   typename TArg1,
   typename TArg2,
   typename TArg3,
   typename TArg4,
   typename TArg5,
   typename TArg6,
   typename TArg7,
   typename TArg8,
   typename TArg9
>
ComPtr<T> Make(
   TArg1 &&arg1,
   TArg2 &&arg2,
   TArg3 &&arg3,
   TArg4 &&arg4,
   TArg5 &&arg5,
   TArg6 &&arg6,
   TArg7 &&arg7,
   TArg8 &&arg8,
   TArg9 &&arg9
);
template <
   typename T,
   typename TArg1,
   typename TArg2,
   typename TArg3,
   typename TArg4,
   typename TArg5,
   typename TArg6,
   typename TArg7,
   typename TArg8
>
ComPtr<T> Make(
   TArg1 &&arg1,
   TArg2 &&arg2,
   TArg3 &&arg3,
   TArg4 &&arg4,
   TArg5 &&arg5,
   TArg6 &&arg6,
   TArg7 &&arg7,
   TArg8 &&arg8
);
template <
   typename T,
   typename TArg1,
   typename TArg2,
   typename TArg3,
   typename TArg4,
   typename TArg5,
   typename TArg6,
   typename TArg7
```

```
>
ComPtr<T> Make(
   TArg1 &&arg1,
   TArg2 &&arg2,
   TArg3 &&arg3,
   TArg4 &&arg4,
   TArg5 &&arg5,
   TArg6 &&arg6,
   TArg7 &&arg7
);
template <
   typename T,
   typename TArg1,
   typename TArg2,
   typename TArg3,
   typename TArg4,
   typename TArg5,
   typename TArg6
>
ComPtr<T> Make(
   TArg1 &&arg1,
   TArg2 &&arg2,
   TArg3 &&arg3,
   TArg4 &&arg4,
   TArg5 &&arg5,
   TArg6 &&arg6
);
template <
   typename T,
   typename TArg1,
   typename TArg2,
   typename TArg3,
   typename TArg4,
   typename TArg5
>
ComPtr<T> Make(
   TArg1 &&arg1,
   TArg2 &&arg2,
   TArg3 &&arg3,
   TArg4 &&arg4,
   TArg5 &&arg5
);
template <
   typename T,
   typename TArg1,
   typename TArg2,
   typename TArg3,
   typename TArg4
>
ComPtr<T> Make(
   TArg1 &&arg1,
   TArg2 &&arg2,
   TArg3 &&arg3,
   TArg4 &&arg4
);
template <
   typename T,
   typename TArg1,
   typename TArg2,
   typename TArg3
>
ComPtr<T> Make(
   TArg1 &&arg1,
   TArg2 &&arg2,
   TArg3 &&arg3
);
template <
   typename T,
   typename TArg1,
```

```
   typename TArg2
>
ComPtr<T> Make(
   TArg1 &&arg1,
   TArg2 &&arg2
);
template <
   typename T,
   typename TArg1
>
ComPtr<T> Make(
   TArg1 &&arg1
);
template <
   typename T
>
ComPtr<T> Make();
```

Parameters

T

A user-specified class that inherits from `WRL::RuntimeClass`.

TArg1

Type of argument 1 that is passed to the specified runtime class.

TArg2

Type of argument 2 that is passed to the specified runtime class.

TArg3

Type of argument 3 that is passed to the specified runtime class.

TArg4

Type of argument 4 that is passed to the specified runtime class.

TArg5

Type of argument 5 that is passed to the specified runtime class.

TArg6

Type of argument 6 that is passed to the specified runtime class.

TArg7

Type of argument 7 that is passed to the specified runtime class.

TArg8

Type of argument 8 that is passed to the specified runtime class.

TArg9

Type of argument 9 that is passed to the specified runtime class.

arg1

Argument 1 that is passed to the specified runtime class.

arg2

Argument 2 that is passed to the specified runtime class.

arg3

Argument 3 that is passed to the specified runtime class.

arg4

Argument 4 that is passed to the specified runtime class.

arg5
Argument 5 that is passed to the specified runtime class.

arg6
Argument 6 that is passed to the specified runtime class.

arg7
Argument 7 that is passed to the specified runtime class.

arg8
Argument 8 that is passed to the specified runtime class.

arg9
Argument 9 that is passed to the specified runtime class.

Return Value

A `ComPtr<T>` object if successful; otherwise, `nullptr`.

Remarks

See How to: Instantiate WRL Components Directly to learn the differences between this function and Microsoft::WRL::Details::MakeAndInitialize, and for an example.

Requirements

Header: implements.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

MixIn Structure

9/21/2022 • 2 minutes to read • Edit Online

Ensures that a runtime class derives from Windows Runtime interfaces, if any, and then classic COM interfaces.

Syntax

```
template<
    typename Derived,
    typename MixInType,
    bool hasImplements = __is_base_of(Details::ImplementsBase, MixInType)
>
struct MixIn;
```

Parameters

Derived
A type derived from the Implements structure.

MixInType
A base type.

hasImplements
`true` if *MixInType* is derived from the current implementation the base type; `false` otherwise.

Remarks

If a class is derived from both Windows Runtime and class COM interfaces, the class declaration list must first list any Windows Runtime interfaces and then any classic COM interfaces. **MixIn** ensures that the interfaces are specified in the correct order.

Inheritance Hierarchy

`MixIn`

Requirements

Header: implements.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

Module Class

9/21/2022 • 6 minutes to read • Edit Online

Represents a collection of related objects.

Syntax

```
template<ModuleType moduleType>
class Module;

template<>
class Module<InProc> : public Details::ModuleBase;

template<>
class Module<OutOfProc> : public Module<InProc>;
```

Parameters

moduleType
A combination of one or more ModuleType enumeration values.

Members

Protected Classes

NAME	DESCRIPTION
Module::GenericReleaseNotifier	Invokes an event handler when the last object in the current module is released. The event handler is specified by on a lambda, functor, or pointer-to-function.
Module::MethodReleaseNotifier	Invokes an event handler when the last object in the current module is released. The event handler is specified by an object and its pointer-to-a-method member.
Module::ReleaseNotifier	Invokes an event handler when the last object in a module is released.

Public Constructors

NAME	DESCRIPTION
Module::~Module	Deinitializes the current instance of the `Module` class.

Protected Constructors

NAME	DESCRIPTION
Module::Module	Initializes a new instance of the `Module` class.

Public Methods

NAME	DESCRIPTION
Module::Create	Creates an instance of a module.
Module::DecrementObjectCount	Decrements the number of objects tracked by the module.
Module::GetActivationFactory	Gets an activation factory for the module.
Module::GetClassObject	Retrieves a cache of class factories.
Module::GetModule	Creates an instance of a module.
Module::GetObjectCount	Retrieves the number of objects managed by this module.
Module::IncrementObjectCount	Increments the number of objects tracked by the module.
Module::RegisterCOMObject	Registers one or more COM objects so other applications can connect to them.
Module::RegisterObjects	Registers COM or Windows Runtime objects so other applications can connect to them.
Module::RegisterWinRTObject	Registers one or more Windows Runtime objects so other applications can connect to them.
Module::Terminate	Causes all factories instantiated by the module to shut down.
Module::UnregisterCOMObject	Unregisters one or more COM objects, which prevents other applications from connecting to them.
Module::UnregisterObjects	Unregisters the objects in the specified module so that other applications cannot connect to them.
Module::UnregisterWinRTObject	Unregisters one or more Windows Runtime objects so that other applications cannot connect to them.

Protected Methods

NAME	DESCRIPTION
Module::Create	Creates an instance of a module.

Protected Data Members

NAME	DESCRIPTION
Module::objectCount_	Keeps track of how many classes have been created with the Make function.
Module::releaseNotifier_	Holds a pointer to a `ReleaseNotifier` object.

Macros

NAME	DESCRIPTION
ActivatableClass	Populates an internal cache that contains a factory that can create an instance of the specified class. This macro specifies default factory and group ID parameters.
ActivatableClassWithFactory	Populates an internal cache that contains a factory that can create an instance of the specified class. This macro enables you to specify a particular factory parameter.
ActivatableClassWithFactoryEx	Populates an internal cache that contains a factory that can create an instance of the specified class. This macro enables you to specify particular factory and group ID parameters.

Inheritance Hierarchy

`ModuleBase`

`Module`

`Module`

Requirements

Header: module.h

Namespace: Microsoft::WRL

Module::~Module

Deinitializes the current instance of the `Module` class.

```
virtual ~Module();
```

Module::Create

Creates an instance of a module.

```
WRL_NOTHROW static Module& Create();
template<typename T>
WRL_NOTHROW static Module& Create(
   T callback
);
template<typename T>
WRL_NOTHROW static Module& Create(
   _In_ T* object,
   _In_ void (T::* method)()
);
```

Parameters

T

Module type.

callback

Called when the last instance object of the module is released.

object
The *object* and *method* parameters are used in combination. Points to the last instance object when the last instance object in the module is released.

method
The *object* and *method* parameters are used in combination. Points to the method of the last instance object when the last instance object in the module is released.

Return Value

Reference to the module.

Module::DecrementObjectCount

Decrements the number of objects tracked by the module.

```
virtual long DecrementObjectCount();
```

Return Value

The count before the decrement operation.

Module::GetActivationFactory

Gets an activation factory for the module.

```
WRL_NOTHROW HRESULT GetActivationFactory(
   _In_ HSTRING pActivatibleClassId,
   _Deref_out_ IActivationFactory **ppIFactory,
   wchar_t* serverName = nullptr
);
```

Parameters

pActivatibleClassId
IID of a runtime class.

ppIFactory
The IActivationFactory for the specified runtime class.

serverName
The name of a subset of class factories in the current module. Specify the server name used in the ActivatableClassWithFactoryEx macro, or specify `nullptr` to get the default server name.

Return Value

S_OK if successful; otherwise, the HRESULT returned by GetActivationFactory.

Module::GetClassObject

Retreives a cache of class factories.

```
HRESULT GetClassObject(
   REFCLSID clsid,
   REFIID riid,
   _Deref_out_ void **ppv,
   wchar_t* serverName = nullptr
);
```

Parameters

clsid
Class ID.

riid
Interface ID that you request.

ppv
Pointer to returned object.

serverName
The server name that is specified in either the `ActivatableClassWithFactory`, `ActivatableClassWithFactoryEx`, or `ActivatableClass` macro; or `nullptr` to get the default server name.

Return Value

Remarks

Use this method only for COM, not the Windows Runtime. This method exposes only `IClassFactory` methods.

Module::GetModule

Creates an instance of a module.

```
static Module& GetModule();
WRL_NOTHROW static Module& GetModule();
```

Return Value

A reference to a module.

Module::GetObjectCount

Retrieves the number of objects managed by this module.

```
virtual long GetObjectCount() const;
```

Return Value

The current number of objects managed by this module.

Module::IncrementObjectCount

Increments the number of objects tracked by the module.

```
virtual long IncrementObjectCount();
```

Return Value

The count before the increment operation.

Module::Module

Initializes a new instance of the `Module` class.

```
Module();
```

Remarks

This constructor is protected and cannot be called with the `new` keyword. Instead, call either Module::GetModule or Module::Create.

Module::objectCount_

Keeps track of how many classes have been created with the Make function.

```
volatile long objectCount_;
```

Module::RegisterCOMObject

Registers one or more COM objects so other applications can connect to them.

```
WRL_NOTHROW virtual HRESULT RegisterCOMObject(
   const wchar_t* serverName,
   IID* clsids,
   IClassFactory** factories,
   DWORD* cookies,
   unsigned int count);
```

Parameters

serverName
Fully-qualified name of a server.

clsids
An array of CLSIDs to register.

factories
An array of IUnknown interfaces of the class objects whose availability is being published.

cookies
When the operation completes, an array of pointers to values that identify the class objects that were registered. These values are later used revoke the registration.

count
The number of CLSIDs to register.

Return Value

S_OK if successfu; otherwise, an HRESULT such as CO_E_OBJISREG that indicates the reason the operation failed.

Remarks

The COM objects are registered with the CLSCTX_LOCAL_SERVER enumerator of the CLSCTX enumeration.

The type of connection to the registered objects is specified by a combination of the current *comflag* template parameter and the REGCLS_SUSPENDED enumerator of the REGCLS enumeration.

Module::RegisterObjects

Registers COM or Windows Runtime objects so other applications can connect to them.

```
HRESULT RegisterObjects(
   ModuleBase* module,
   const wchar_t* serverName);
```

Parameters

module
An array of COM or Windows Runtime objects.

serverName
Name of the server that created the objects.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the reason the operation failed.

Module::RegisterWinRTObject

Registers one or more Windows Runtime objects so other applications can connect to them.

```
HRESULT RegisterWinRTObject(const wchar_t* serverName,
   wchar_t** activatableClassIds,
   WINRT_REGISTRATION_COOKIE* cookie,
   unsigned int count)
```

Parameters

serverName
A name that specifies a subset of objects affected by this operation.

activatableClassIds
An array of activatable CLSIDs to register.

cookie
A value that identifies the class objects that were registered. This value is used later to revoke the registration.

count
The number of objects to register.

Return Value

S_OK if successful; otherwise, an error HRESULT such as CO_E_OBJISREG that indicates the reason the operation failed.

Module::releaseNotifier_

Holds a pointer to a `ReleaseNotifier` object.

```
ReleaseNotifier *releaseNotifier_;
```

Module::Terminate

Causes all factories instantiated by the module to shut down.

```
void Terminate();
```

Remarks

Releases the factories in the cache.

Module::UnregisterCOMObject

Unregisters one or more COM objects, which prevents other applications from connecting to them.

```
virtual HRESULT UnregisterCOMObject(
   const wchar_t* serverName,
   DWORD* cookies,
   unsigned int count
```

Parameters

serverName
(Unused)

cookies
An array of pointers to values that identify the class objects to be unregistered. The array was created by the RegisterCOMObject method.

count
The number of classes to unregister.

Return Value

S_OK if this operation is successful; otherwise, an error HRESULT that indicates the reason the operation failed.

Module::UnregisterObjects

Unregisters the objects in the specified module so that other applications cannot connect to them.

```
HRESULT UnregisterObjects(
   ModuleBase* module,
   const wchar_t* serverName);
```

Parameters

module
Pointer to a module.

serverName
A qualifying name that specifies a subset of objects affected by this operation.

Return Value

S_OK if this operation is successful; otherwise, an error HRESULT that indicates the reason this operation failed.

Module::UnregisterWinRTObject

Unregisters one or more Windows Runtime objects so that other applications cannot connect to them.

```
virtual HRESULT UnregisterWinRTObject(
   unsigned int,
   _Inout_ WINRT_REGISTRATION_COOKIE* cookie
);
```

Parameters

cookie
A pointer to a value that identifies the class object whose registration is to be revoked.

Module::GenericReleaseNotifier Class

9/21/2022 • 2 minutes to read • Edit Online

Invokes an event handler when the last object in the current module is released. The event handler is specified by on a lambda, functor, or pointer-to-function.

Syntax

```
template<typename T>
class GenericReleaseNotifier : public ReleaseNotifier;
```

Parameters

T

The type of the data member that contains the location of the event handler.

Members

Public Constructors

NAME	DESCRIPTION
Module::GenericReleaseNotifier::GenericReleaseNotifier	Initializes a new instance of the `Module::GenericReleaseNotifier` class.

Public Methods

NAME	DESCRIPTION
Module::GenericReleaseNotifier::Invoke	Calls the event handler associated with the current `Module::GenericReleaseNotifier` object.

Protected Data Members

NAME	DESCRIPTION
Module::GenericReleaseNotifier::callback_	Holds the lambda, functor, or pointer-to-function event handler associated with the current `Module::GenericReleaseNotifier` object.

Inheritance Hierarchy

`ReleaseNotifier`

`GenericReleaseNotifier`

Requirements

Header: module.h

Namespace: Microsoft::WRL

Module::GenericReleaseNotifier::callback_

Holds the lambda, functor, or pointer-to-function event handler associated with the current `Module::GenericReleaseNotifier` object.

```
T callback_;
```

Module::GenericReleaseNotifier::GenericReleaseNotifier

Initializes a new instance of the `Module::GenericReleaseNotifier` class.

```
GenericReleaseNotifier(
   T callback,
   bool release
) throw() : ReleaseNotifier(release), callback_(callback);
```

Parameters

callback
A lambda, functor, or pointer-to-function event handler that can be invoked with the parentheses function operator (`()`).

release
Specify `true` to enable calling the underlying Module::ReleaseNotifier::Release() method; otherwise, specify `false`.

Module::GenericReleaseNotifier::Invoke

Calls the event handler associated with the current `Module::GenericReleaseNotifier` object.

```
void Invoke();
```

Module::MethodReleaseNotifier Class

9/21/2022 • 2 minutes to read • Edit Online

Invokes an event handler when the last object in the current module is released. The event handler is specified by an object and its pointer-to-a-method member.

Syntax

```
template<typename T>
class MethodReleaseNotifier : public ReleaseNotifier;
```

Parameters

T

The type of the object whose member function is the event handler.

Members

Public Constructors

NAME	DESCRIPTION
Module::MethodReleaseNotifier::MethodReleaseNotifier	Initializes a new instance of the `Module::MethodReleaseNotifier` class.

Public Methods

NAME	DESCRIPTION
Module::MethodReleaseNotifier::Invoke	Calls the event handler associated with the current `Module::MethodReleaseNotifier` object.

Protected Data Members

NAME	DESCRIPTION
Module::MethodReleaseNotifier::method_	Holds a pointer to the event handler for the current `Module::MethodReleaseNotifier` object.
Module::MethodReleaseNotifier::object_	Holds a pointer to the object whose member function is the event handler for the current `Module::MethodReleaseNotifier` object.

Inheritance Hierarchy

`ReleaseNotifier`

`MethodReleaseNotifier`

Requirements

Header: module.h

Namespace: Microsoft::WRL

Module::MethodReleaseNotifier::Invoke

Calls the event handler associated with the current `Module::MethodReleaseNotifier` object.

```
void Invoke();
```

Module::MethodReleaseNotifier::method_

Holds a pointer to the event handler for the current `Module::MethodReleaseNotifier` object.

```
void (T::* method_)();
```

Module::MethodReleaseNotifier::MethodReleaseNotifier

Initializes a new instance of the `Module::MethodReleaseNotifier` class.

```
MethodReleaseNotifier(
   _In_ T* object,
   _In_ void (T::* method)(),
   bool release) throw() :
            ReleaseNotifier(release), object_(object),
            method_(method);
```

Parameters

object
An object whose member function is an event handler.

method
The member function of parameter *object* that is the event handler.

release
Specify `true` to enable calling the underlying Module::ReleaseNotifier::Release() method; otherwise, specify `false`.

Module::MethodReleaseNotifier::object_

Holds a pointer to the object whose member function is the event handler for the current `Module::MethodReleaseNotifier` object.

```
T* object_;
```

Module::ReleaseNotifier Class

9/21/2022 • 2 minutes to read • Edit Online

Invokes an event handler when the last object in a module is released.

Syntax

```
class ReleaseNotifier;
```

Members

Public Constructors

NAME	DESCRIPTION
Module::ReleaseNotifier::~ReleaseNotifier	Deinitializes the current instance of the `Module::ReleaseNotifier` class.
Module::ReleaseNotifier::ReleaseNotifier	Initializes a new instance of the `Module::ReleaseNotifier` class.

Public Methods

NAME	DESCRIPTION
Module::ReleaseNotifier::Invoke	When implemented, calls an event handler when the last object in a module is released.
Module::ReleaseNotifier::Release	Deletes the current `Module::ReleaseNotifier` object if the object was constructed with a parameter of `true`.

Inheritance Hierarchy

`ReleaseNotifier`

Requirements

Header: module.h

Namespace: Microsoft::WRL

Module::ReleaseNotifier::~ReleaseNotifier

Deinitializes the current instance of the `Module::ReleaseNotifier` class.

```
WRL_NOTHROW virtual ~ReleaseNotifier();
```

Module::ReleaseNotifier::Invoke

When implemented, calls an event handler when the last object in a module is released.

```
virtual void Invoke() = 0;
```

Module::ReleaseNotifier::Release

Deletes the current `Module::ReleaseNotifier` object if the object was constructed with a parameter of `true`.

```
void Release() throw();
```

Module::ReleaseNotifier::ReleaseNotifier

Initializes a new instance of the `Module::ReleaseNotifier` class.

```
ReleaseNotifier(bool release) throw();
```

Parameters

release
`true` to delete this instance when the `Release` method is called; `false` to not delete this instance.

ModuleType Enumeration

9/21/2022 • 2 minutes to read • Edit Online

Specifies whether a module should support an in-process server or an out-of-process server.

Syntax

```
enum ModuleType;
```

Members

Values

NAME	DESCRIPTION
`InProc`	An in-process server.
`OutOfProc`	An out-of-process server.
`DisableCaching`	Disable caching mechanism on Module.
`InProcDisableCaching`	Combination of `InProc` and `DisableCaching`.
`OutOfProcDisableCaching`	Combination of `OutOfProc` and `DisableCaching`.

Requirements

Header: module.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

operator!= Operator (Microsoft::WRL)

9/21/2022 • 2 minutes to read • Edit Online

Inequality operator for ComPtr and ComPtrRef objects.

Syntax

```
WRL_NOTHROW bool operator!=(
   const ComPtr<T>& a,
   const ComPtr<U>& b
);
WRL_NOTHROW bool operator!=(
   const ComPtr<T>& a,
   decltype(__nullptr)
);
WRL_NOTHROW bool operator!=(
   decltype(__nullptr),
   const ComPtr<T>& a
);
WRL_NOTHROW bool operator!=(
   const Details::ComPtrRef<ComPtr<T>>& a,
   const Details::ComPtrRef<ComPtr<U>>& b
);
WRL_NOTHROW bool operator!=(
   const Details::ComPtrRef<ComPtr<T>>& a,
   decltype(__nullptr)
);
WRL_NOTHROW bool operator!=(
   decltype(__nullptr),
   const Details::ComPtrRef<ComPtr<T>>& a
);
WRL_NOTHROW bool operator!=(
   const Details::ComPtrRef<ComPtr<T>>& a,
   void* b
);
WRL_NOTHROW bool operator!=(
   void* b,
   const Details::ComPtrRef<ComPtr<T>>& a
);
```

Parameters

a

The left object.

b

The right object.

Return Value

`true` if the objects are not equal; otherwise, `false`.

Requirements

Header: client.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

operator== Operator (Microsoft::WRL)

9/21/2022 • 2 minutes to read • Edit Online

Equality operator for ComPtr and ComPtrRef objects.

Syntax

```
WRL_NOTHROW bool operator==(
   const ComPtr<T>& a,
   const ComPtr<U>& b
);
WRL_NOTHROW bool operator==(
   const ComPtr<T>& a,
   decltype(__nullptr)
);
WRL_NOTHROW bool operator==(
   decltype(__nullptr),
   const ComPtr<T>& a
);
WRL_NOTHROW bool operator==(
   const Details::ComPtrRef<ComPtr<T>>& a,
   const Details::ComPtrRef<ComPtr<U>>& b
);
WRL_NOTHROW bool operator==(
   const Details::ComPtrRef<ComPtr<T>>& a,
   decltype(__nullptr)
);
WRL_NOTHROW bool operator==(
   decltype(__nullptr),
   const Details::ComPtrRef<ComPtr<T>>& a
);
WRL_NOTHROW bool operator==(
   const Details::ComPtrRef<ComPtr<T>>& a,
   void* b
);
WRL_NOTHROW bool operator==(
   void* b,
   const Details::ComPtrRef<ComPtr<T>>& a
);
```

Parameters

a

The left object.

b

The right object.

Return Value

`true` if the objects are equal; otherwise, `false`.

Requirements

Header: client.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

`operator<` Operator (Microsoft::WRL)

9/21/2022 • 2 minutes to read • Edit Online

Determines if the address of one object is less than another.

Syntax

```
template<class T, class U>
bool operator<(const ComPtr<T>& a, const ComPtr<U>& b) throw();
template<class T, class U>
bool operator<(const Details::ComPtrRef<ComPtr<T>>& a, const Details::ComPtrRef<ComPtr<U>>& b) throw();
```

Parameters

a
The left object.

b
The right object.

Return Value

`true` if the address of *a* is less than the address of *b*; otherwise, `false`.

Requirements

Header: client.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

RuntimeClass Class

9/21/2022 • 3 minutes to read • Edit Online

Represents a WinRT or COM class that inherits the specified interfaces and provides the specified Windows Runtime, classic COM, and weak reference support.

This class provides the boilerplate implementation of WinRT and COM classes, providing the implementation of `QueryInterface`, `AddRef`, `Release` etc., manages the reference count of the module and has support for providing the class factory for activatable objects.

Syntax

```
template <typename ...TInterfaces> class RuntimeClass
template <unsigned int classFlags, typename ...TInterfaces> class RuntimeClass;
```

Parameters

classFlags
Optional parameter. A combination of one or more RuntimeClassType enumeration values. The `__WRL_CONFIGURATION_LEGACY__` macro can be defined to change the default value of classFlags for all runtime classes in the project. If defined, RuntimeClass instances are non-agile by default. When not defined, RuntimeClass instances are agile by default. To avoid ambiguity always specify the `Microsoft::WRL::FtmBase` in `TInterfaces` or `RuntimeClassType::InhibitFtmBase`. Note, if InhibitFtmBase and FtmBase are both used the object will be agile.

TInterfaces
The list of interfaces the object implements beyond `IUnknown`, `IInspectable` or other interfaces controlled by RuntimeClassType. It also may list other classes to be derived from, notably `Microsoft::WRL::FtmBase` to make the object agile and cause it to implement `IMarshal`.

Members

`RuntimeClassInitialize`
A function which initializes the object if the `MakeAndInitialize` template function is used to construct the object. It returns S_OK if the object was successfully initialized, or a COM error code if initialization failed. The COM error code is propagated as the return value of `MakeAndInitialize`. Note that the `RuntimeClassInitialize` method is not called if the `Make` template function is used to construct the object.

Public Constructors

NAME	DESCRIPTION
RuntimeClass::RuntimeClass	Initializes the current instance of the `RuntimeClass` class.
RuntimeClass::~RuntimeClass	Deinitializes the current instance of the `RuntimeClass` class.

Public Methods

NAME	DESCRIPTION
RuntimeClass::AddRef	Increments the reference count for the current `RuntimeClass` object.
RuntimeClass::DecrementReference	Decrements the reference count for the current `RuntimeClass` object.
RuntimeClass::GetIids	Gets an array that can contain the interface IDs implemented by the current `RuntimeClass` object.
RuntimeClass::GetRuntimeClassName	Gets the runtime class name of the current `RuntimeClass` object.
RuntimeClass::GetTrustLevel	Gets the trust level of the current `RuntimeClass` object.
RuntimeClass::GetWeakReference	Gets a pointer to the weak reference object for the current `RuntimeClass` object.
RuntimeClass::InternalAddRef	Increments the reference count to the current `RuntimeClass` object.
RuntimeClass::QueryInterface	Retrieves a pointer to the specified interface ID.
RuntimeClass::Release	Performs a COM Release operation on the current `RuntimeClass` object.

Inheritance Hierarchy

This is an implementation detail.

Requirements

Header: implements.h

Namespace: Microsoft::WRL

RuntimeClass::~RuntimeClass

Deinitializes the current instance of the `RuntimeClass` class.

```
virtual ~RuntimeClass();
```

RuntimeClass::AddRef

Increments the reference count for the current `RuntimeClass` object.

```
STDMETHOD_(
   ULONG,
   AddRef
)();
```

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

RuntimeClass::DecrementReference

Decrements the reference count for the current `RuntimeClass` object.

```
ULONG DecrementReference();
```

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

RuntimeClass::GetIids

Gets an array that can contain the interface IDs implemented by the current `RuntimeClass` object.

```
STDMETHOD(
   GetIids
)
   (_Out_ ULONG *iidCount,
   _Deref_out_ _Deref_post_cap_(*iidCount) IID **iids);
```

Parameters

iidCount
When this operation completes, the total number of elements in array *iids*.

iids
When this operation completes, a pointer to an array of interface IDs.

Return Value

S_OK if successful; otherwise, E_OUTOFMEMORY.

RuntimeClass::GetRuntimeClassName

Gets the runtime class name of the current `RuntimeClass` object.

```
STDMETHOD( GetRuntimeClassName )(
    _Out_ HSTRING* runtimeName
);
```

Parameters

runtimeName
When this operation completes, the runtime class name.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

Remarks

An assert error is emitted if `__WRL_STRICT__` or `__WRL_FORCE_INSPECTABLE_CLASS_MACRO__` isn't defined.

RuntimeClass::GetTrustLevel

Gets the trust level of the current `RuntimeClass` object.

```
STDMETHOD(GetTrustLevel)(
    _Out_ TrustLevel* trustLvl
);
```

Parameters

trustLvl
When this operation completes, the trust level of the current `RuntimeClass` object.

Return Value

Always S_OK.

Remarks

An assert error is emitted if `__WRL_STRICT__` or `__WRL_FORCE_INSPECTABLE_CLASS_MACRO__` isn't defined.

RuntimeClass::GetWeakReference

Gets a pointer to the weak reference object for the current `RuntimeClass` object.

```
STDMETHOD(
   GetWeakReference
)(_Deref_out_ IWeakReference **weakReference);
```

Parameters

weakReference
When this operation completes, a pointer to a weak reference object.

Return Value

Always S_OK.

RuntimeClass::InternalAddRef

Increments the reference count to the current `RuntimeClass` object.

```
ULONG InternalAddRef();
```

Return Value

The resulting reference count.

RuntimeClass::QueryInterface

Retrieves a pointer to the specified interface ID.

```
STDMETHOD(
   QueryInterface
)
   (REFIID riid,
   _Deref_out_ void **ppvObject);
```

Parameters

riid
An interface ID.

ppvObject
When this opereation completes, a pointer to the interface specified by the *riid* parameter.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

RuntimeClass::Release

Performs a COM Release operation on the current `RuntimeClass` object.

```
STDMETHOD_(
   ULONG,
   Release
)();
```

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

Remarks

If the reference count becomes zero, the `RuntimeClass` object is deleted.

RuntimeClass::RuntimeClass

Initializes the current instance of the `RuntimeClass` class.

```
RuntimeClass();
```

RuntimeClassFlags Structure

9/21/2022 • 2 minutes to read • Edit Online

Contains the type for an instance of a RuntimeClass.

Syntax

```
template <unsigned int flags>
struct RuntimeClassFlags;
```

Parameters

flags
A RuntimeClassType Enumeration value.

Members

Public Constants

NAME	DESCRIPTION
RuntimeClassFlags::value Constant	Contains a RuntimeClassType Enumeration value.

Inheritance Hierarchy

`RuntimeClassFlags`

Requirements

Header: implements.h

Namespace: Microsoft::WRL

RuntimeClassFlags::value Constant

A field that contains a RuntimeClassType Enumeration value.

```
static const unsigned int value = flags;
```

RuntimeClassType Enumeration

9/21/2022 • 2 minutes to read • Edit Online

Specifies the type of RuntimeClass instance that is supported.

Syntax

```
enum RuntimeClassType;
```

Members

Values

NAME	DESCRIPTION
`ClassicCom`	A classic COM runtime class.
`Delegate`	Equivalent to `ClassicCom`.
`InhibitFtmBase`	Disables `FtmBase` support while `__WRL_CONFIGURATION_LEGACY__` is not defined.
`InhibitWeakReference`	Disables weak reference support.
`WinRt`	A Windows Runtime class.
`WinRtClassicComMix`	A combination of `WinRt` and `ClassicCom`.

Requirements

Header: implements.h

Namespace: Microsoft::WRL

See also

Microsoft::WRL Namespace

SimpleActivationFactory Class

9/21/2022 • 2 minutes to read • Edit Online

Provides a fundamental mechanism to create a Windows Runtime or classic COM base class.

Syntax

```
template<typename Base>
class SimpleActivationFactory : public ActivationFactory<>;
```

Parameters

Base
A base class.

Remarks

The base class must provide a default constructor.

The following code example demonstrates how to use SimpleActivationFactory with the ActivatableClassWithFactoryEx macro.

```
ActivatableClassWithFactoryEx(MyClass, SimpleActivationFactory, MyServerName);
```

Members

Public Methods

NAME	DESCRIPTION
SimpleActivationFactory::ActivateInstance Method	Creates an instance of the specified interface.
SimpleActivationFactory::GetRuntimeClassName Method	Gets the runtime class name of an instance of the class specified by the *Base* class template parameter.
SimpleActivationFactory::GetTrustLevel Method	Gets the trust level of an instance of the class specified by the *Base* class template parameter.

Inheritance Hierarchy

`I0`

`ChainInterfaces`

`I0`

`RuntimeClassBase`

`ImplementsHelper`

`DontUseNewUseMake`

`RuntimeClassFlags`

`RuntimeClassBaseT`

`RuntimeClass`

`ActivationFactory`

`SimpleActivationFactory`

Requirements

Header: module.h

Namespace: Microsoft::WRL

SimpleActivationFactory::ActivateInstance Method

Creates an instance of the specified interface.

```
STDMETHOD( ActivateInstance )(
    _Deref_out_ IInspectable **ppvObject
);
```

Parameters

ppvObject

When this operation completes, pointer to an instance of the object specified by the `Base` class template parameter.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

Remarks

If `__WRL_STRICT__` is defined, an assert error is emitted if the base class specified in the class template parameter isn't derived from RuntimeClass, or isn't configured with the WinRt or WinRtClassicComMix RuntimeClassType enumeration value.

SimpleActivationFactory::GetRuntimeClassName Method

Gets the runtime class name of an instance of the class specified by the `Base` class template parameter.

```
STDMETHOD( GetRuntimeClassName )(
    _Out_ HSTRING* runtimeName
);
```

Parameters

runtimeName

When this operation completes, the runtime class name.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

Remarks

If `__WRL_STRICT__` is defined, an assert error is emitted if the class specified by the `Base` class template parameter isn't derived from RuntimeClass, or isn't configured with the WinRt or WinRtClassicComMix RuntimeClassType enumeration value.

SimpleActivationFactory::GetTrustLevel Method

Gets the trust level of an instance of the class specified by the `Base` class template parameter.

```
STDMETHOD(
   GetTrustLevel
)(_Out_ TrustLevel* trustLvl);
```

Parameters

trustLvl

When this operation completes, the trust level of the current class object.

Return Value

Always S_OK.

SimpleClassFactory Class

9/21/2022 • 2 minutes to read • Edit Online

Provides a fundamental mechanism to create a base class.

Syntax

```
template<typename Base>
class SimpleClassFactory : public ClassFactory<>;
```

Parameters

Base
A base class.

Remarks

The base class must provide a default constructor.

The following code example demonstrates how to use `SimpleClassFactory` with the ActivatableClassWithFactoryEx macro.

```
ActivatableClassWithFactoryEx(MyClass, SimpleClassFactory, MyServerName);
```

Members

Public Methods

NAME	DESCRIPTION
SimpleClassFactory::CreateInstance Method	Creates an instance of the specified interface.

Inheritance Hierarchy

`I0`

`ChainInterfaces`

`I0`

`RuntimeClassBase`

`ImplementsHelper`

`DontUseNewUseMake`

`RuntimeClassFlags`

`RuntimeClassBaseT`

`RuntimeClass`

`ClassFactory`

`SimpleClassFactory`

Requirements

Header: module.h

Namespace: Microsoft::WRL

SimpleClassFactory::CreateInstance Method

Creates an instance of the specified interface.

```
STDMETHOD( CreateInstance )(
   _Inout_opt_ IUnknown* pUnkOuter,
   REFIID riid,
   _Deref_out_ void** ppvObject
);
```

Parameters

pUnkOuter
Must be `nullptr`; otherwise, the return value is CLASS_E_NOAGGREGATION.

SimpleClassFactory doesn't support aggregation. If aggregation were supported and the object being created was part of an aggregate, *pUnkOuter* would be a pointer to the controlling `IUnknown` interface of the aggregate.

riid
Interface ID of the object to create.

ppvObject
When this operation completes, pointer to an instance of the object specified by the *riid* parameter.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

Remarks

If `__WRL_STRICT__` is defined, an assert error is emitted if the base class specified in the class template parameter isn't derived from RuntimeClass, or isn't configured with the ClassicCom or WinRtClassicComMix RuntimeClassType enumeration value.

`WeakRef` class

9/21/2022 • 5 minutes to read • Edit Online

Represents a *weak reference* that can be used by only the Windows Runtime, not classic COM. A weak reference represents an object that might or might not be accessible.

Syntax

```
class WeakRef : public ComPtr<IWeakReference>;
```

Members

Public constructors

NAME	DESCRIPTION
`WeakRef::WeakRef` constructor	Initializes a new instance of the `WeakRef` class.
`WeakRef::~WeakRef` destructor	Deinitializes the current instance of the `WeakRef` class.

Public methods

NAME	DESCRIPTION
`WeakRef::As`	Sets the specified `ComPtr` pointer parameter to represent the specified interface.
`WeakRef::AsIID`	Sets the specified `ComPtr` pointer parameter to represent the specified interface ID.
`WeakRef::CopyTo`	Assigns a pointer to an interface, if available, to the specified pointer variable.

Public operators

NAME	DESCRIPTION
`WeakRef::operator&`	Returns a `ComPtrRef` object that represents the current `WeakRef` object.

Remarks

A `WeakRef` object maintains a *strong reference*, which is associated with an object, and can be valid or invalid. Call the `As()` or `AsIID()` method to obtain a strong reference. When the strong reference is valid, it can access the associated object. When the strong reference is invalid (`nullptr`), the associated object is inaccessible.

A `WeakRef` object is typically used to represent an object whose existence is controlled by an external thread or application. For example, construct a `WeakRef` object from a reference to a file object. While the file is open, the strong reference is valid. But if the file is closed, the strong reference becomes invalid.

There's a behavior change in the `As`, `AsIID`, and `CopyTo` methods in the Windows SDK. Previously, after calling any of these methods, you could check the `WeakRef` for `nullptr` to determine if a strong reference was successfully obtained, as in the following code:

```
WeakRef wr;
strongComptrRef.AsWeak(&wr);

// Now suppose that the object strongComPtrRef points to no longer exists
// and the following code tries to get a strong ref from the weak ref:
ComPtr<ISomeInterface> strongRef;
HRESULT hr = wr.As(&strongRef);

// This check won't work with the Windows 10 SDK version of the library.
// Check the input pointer instead.
if(wr == nullptr)
{
    wprintf(L"Couldn’t get strong ref!");
}
```

The above code doesn't work when using the Windows 10 SDK (or later). Instead, check the pointer that was passed in for `nullptr`.

```
if (strongRef == nullptr)
{
    wprintf(L"Couldn't get strong ref!");
}
```

Inheritance hierarchy

`ComPtr`
  └ `WeakRef`

Requirements

Header: *`client.h`*

Namespace: `Microsoft::WRL`

`WeakRef::WeakRef` constructor

Initializes a new instance of the `WeakRef` class.

```
WeakRef();
WeakRef(
   decltype(__nullptr)
);

WeakRef(
   _In_opt_ IWeakReference* ptr
);

WeakRef(
   const ComPtr<IWeakReference>& ptr
);

WeakRef(
   const WeakRef& ptr
);

WeakRef(
   _Inout_ WeakRef&& ptr
);
```

Parameters

ptr

A pointer, reference, or rvalue-reference to an existing object that initializes the current `WeakRef` object.

Remarks

The first constructor initializes an empty `WeakRef` object. The second constructor initializes a `WeakRef` object from a pointer to the `IWeakReference` interface. The third constructor initializes a `WeakRef` object from a reference to a `ComPtr<IWeakReference>` object. The fourth and fifth constructors initialize a `WeakRef` object from another `WeakRef` object.

`WeakRef::~WeakRef` destructor

Deinitializes the current instance of the `WeakRef` class.

```
~WeakRef();
```

`WeakRef::As`

Sets the specified `ComPtr` pointer parameter to represent the specified interface.

```
template<typename U>
HRESULT As(
   _Out_ ComPtr<U>* ptr
);

template<typename U>
HRESULT As(
   _Out_ Details::ComPtrRef<ComPtr<U>> ptr
);
```

Parameters

U

An interface ID.

ptr

When this operation completes, an object that represents parameter *U*.

Return value

- `S_OK` if this operation succeeds; otherwise, an HRESULT that indicates the reason the operation failed, and `ptr` is set to `nullptr`.
- `S_OK` if this operation succeeds, but the current `WeakRef` object has already been released. Parameter `ptr` is set to `nullptr`.
- `S_OK` if this operation succeeds, but the current `WeakRef` object isn't derived from parameter `U`. Parameter `ptr` is set to `nullptr`.

Remarks

An error is emitted if parameter `U` is `IWeakReference`, or isn't derived from `IInspectable`.

The first template is the form that you should use in your code. The second template is an internal, helper specialization; it supports C++ language features such as the `auto` type deduction keyword.

Starting in the Windows 10 SDK, this method doesn't set the `WeakRef` instance to `nullptr` if the weak reference couldn't be obtained, so you should avoid error-checking code that checks the `WeakRef` for `nullptr`. Instead, check *ptr* for `nullptr`.

`WeakRef::AsIID`

Sets the specified `ComPtr` pointer parameter to represent the specified interface ID.

```
HRESULT AsIID(
   REFIID riid,
   _Out_ ComPtr<IInspectable>* ptr
);
```

Parameters

`riid`
An interface ID.

`ptr`
When this operation completes, an object that represents parameter `riid`.

Return value

- `S_OK` if this operation succeeds; otherwise, an HRESULT that indicates the reason the operation failed, and `ptr` is set to `nullptr`.
- `S_OK` if this operation succeeds, but the current `WeakRef` object has already been released. Parameter `ptr` is set to `nullptr`.
- `S_OK` if this operation succeeds, but the current `WeakRef` object isn't derived from parameter `riid`. Parameter `ptr` is set to `nullptr`. (For more information, see Remarks.)

Remarks

An error is emitted if parameter `riid` isn't derived from `IInspectable`. This error supersedes the return value.

The first template is the form that you should use in your code. The second template (not shown here, but declared in the header file) is an internal, helper specialization that supports C++ language features such as the `auto` type deduction keyword.

Starting in the Windows 10 SDK, this method doesn't set the `WeakRef` instance to `nullptr` if the weak reference

couldn't be obtained, so you should avoid error-checking code that checks the `WeakRef` for `nullptr`. Instead, check `ptr` for `nullptr`.

`WeakRef::CopyTo`

Assigns a pointer to an interface, if available, to the specified pointer variable.

```
HRESULT CopyTo(
   REFIID riid,
   _Deref_out_ IInspectable** ptr
);

template<typename U>
HRESULT CopyTo(
   _Deref_out_ U** ptr
);

HRESULT CopyTo(
   _Deref_out_ IWeakReference** ptr
);
```

Parameters

`U`
Pointer an `IInspectable` interface. An error is emitted if `U` isn't derived from `IInspectable`.

`riid`
An interface ID. An error is emitted if `riid` isn't derived from `IWeakReference`.

`ptr`
A doubly indirect pointer to `IInspectable` or `IWeakReference`.

Return value

`S_OK` if successful; otherwise, an HRESULT that describes the failure. For more information, see **Remarks**.

Remarks

A return value of `S_OK` means that this operation succeeded, but doesn't indicate whether the weak reference was resolved to a strong reference. If `S_OK` is returned, test that parameter `ptr` is a strong reference; that is, parameter `ptr` isn't equal to `nullptr`.

Starting in the Windows 10 SDK, this method doesn't set the `WeakRef` instance to `nullptr` if the weak reference couldn't be obtained, so you should avoid error checking code that checks the `WeakRef` for `nullptr`. Instead, check `ptr` for `nullptr`.

`WeakRef::operator&`

Returns a `ComPtrRef` object that represents the current `WeakRef` object.

```
Details::ComPtrRef<WeakRef> operator&() throw()
```

Return value

A `ComPtrRef` object that represents the current `WeakRef` object.

Remarks

`WeakRef::operator&` is an internal helper operator that's not meant to be used in your code.

Microsoft::WRL::Details Namespace

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
namespace Microsoft::WRL::Details;
```

Members

Classes

NAME	DESCRIPTION
ComPtrRef Class	Represents a reference to an object of type ComPtr<T>.
ComPtrRefBase Class	Represents the base class for the ComPtrRef class.
DontUseNewUseMake Class	Prevents using operator `new` in `RuntimeClass`. Consequently, you must use the Make function instead.
EventTargetArray Class	Represents an array of event handlers.
MakeAllocator Class	Allocates memory for an activatable class, with or without weak reference support.
ModuleBase Class	Represents the base class of the Module classes.
RemoveIUnknown Class	Makes a type that is equivalent to an `IUnknown`-based type, but has non-virtual `QueryInterface`, `AddRef`, and `Release` methods.
WeakReference Class	Represents a *weak reference* that can be used with the Windows Runtime or classic COM. A weak reference represents an object that might or might not be accessible.

Structures

NAME	DESCRIPTION
ArgTraits Structure	Declares a specified delegate interface and an anonymous member function that has a specified number of parameters.
ArgTraitsHelper Structure	Helps define common characteristics of delegate arguments.
BoolStruct Structure	Defines whether a `ComPtr` is managing the object lifetime of an interface. `BoolStruct` is used internally by the BoolType() operator.

NAME	DESCRIPTION
CreatorMap Structure	Contains information about how to initialize, register, and unregister objects.
DerefHelper Structure	Represent a dereferenced pointer to the `T*` template parameter.
EnableIf Structure	Defines a data member of the type specified by the second template parameter if the first template parameter evaluates to `true`.
FactoryCache Structure	Contains the location of a class factory and a value that identifies a registered Windows Runtime or COM class object.
ImplementsBase Structure	Used to validate template parameter types in Implements Structure.
ImplementsHelper Structure	Helps implement the Implements structure.
InterfaceList Structure	Used to create a recursive list of interfaces.
InterfaceListHelper Structure	Builds an `InterfaceList` type by recursively applying the specified template parameter arguments.
InterfaceTraits Structure	Implements common characteristics of an interface.
InvokeHelper Structure	Provides an implementation of the `Invoke()` method based on the specified number and type of arguments.
IsBaseOfStrict Structure	Tests whether one type is the base of another.
IsSame Structure	Tests whether one specified type is the same as another specified type.
Nil Structure	Used to indicate an unspecified, optional template parameter.
RemoveReference Structure	Strips the reference or rvalue-reference trait from the specified class template parameter.
RuntimeClassBase Structure	Used to detect `RuntimeClass` in the Make function.
RuntimeClassBaseT Structure	Provides helper methods for `QueryInterface` operations and getting interface IDs.
VerifyInheritanceHelper Structure	Tests whether one interface is derived from another interface.
VerifyInterfaceHelper Structure	Verifies that the interface specified by the template parameter meets certain requirements.

Enumerations

NAME	DESCRIPTION
AsyncStatusInternal Enumeration	Specifies a mapping between internal enumerations for the state of asynchronous operations and the `Windows::Foundation::AsyncStatus` enumeration.

Functions

NAME	DESCRIPTION
ActivationFactoryCallback Function	Gets the activation factory for the specified activation ID.
Move Function	Moves the specified argument from one location to another.
RaiseException Function	Raises an exception in the calling thread.
Swap Function (WRL)	Exchanges the values of the two specified arguments.
TerminateMap Function	Shuts down the class factories in the specified module.

Requirements

Header: async.h, client.h, corewrappers.h, event.h, ftm.h, implements.h, internal.h, module.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL Namespace
Microsoft::WRL::Wrappers Namespace

ActivationFactoryCallback Function

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
inline HRESULT STDAPICALLTYPE ActivationFactoryCallback(
   HSTRING activationId,
   IActivationFactory **ppFactory
);
```

Parameters

activationId
Handle to a string that specifies a runtime class name.

ppFactory
When this operation completes, an activation factory that corresponds to parameter *activationId*.

Return Value

S_OK if successful; otherwise, an HRESULT that describes the failure. Likely failure HRESULTs are CLASS_E_CLASSNOTAVAILABLE and E_INVALIDARG.

Remarks

Gets the activation factory for the specified activation ID.

The Windows Runtime calls this callback function to request an object specified by its runtime class name.

Requirements

Header: module.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

ArgTraits Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template<typename TMemberFunction>
struct ArgTraits;

template<typename TDelegateInterface>
struct ArgTraits<HRESULT (STDMETHODCALLTYPE TDelegateInterface::*)(void)>;

template<typename TDelegateInterface, typename TArg1>
struct ArgTraits<HRESULT (STDMETHODCALLTYPE TDelegateInterface::*)(TArg1)>;

template<typename TDelegateInterface, typename TArg1, typename TArg2>
struct ArgTraits<
    HRESULT (STDMETHODCALLTYPE TDelegateInterface::*)(TArg1, TArg2)>;

template<
    typename TDelegateInterface,
    typename TArg1,
    typename TArg2,
    typename TArg3
>
struct ArgTraits<
    HRESULT (STDMETHODCALLTYPE TDelegateInterface::*)(TArg1, TArg2, TArg3)>;

template<
    typename TDelegateInterface,
    typename TArg1,
    typename TArg2,
    typename TArg3,
    typename TArg4
>
struct ArgTraits<
    HRESULT (STDMETHODCALLTYPE TDelegateInterface::*)
             (TArg1, TArg2, TArg3, TArg4)>;

template<
    typename TDelegateInterface,
    typename TArg1,
    typename TArg2,
    typename TArg3,
    typename TArg4,
    typename TArg5
>
struct ArgTraits<
    HRESULT (STDMETHODCALLTYPE TDelegateInterface::*)
             (TArg1, TArg2, TArg3, TArg4, TArg5)>;

template<
    typename TDelegateInterface,
    typename TArg1,
    typename TArg2,
    typename TArg3,
    typename TArg4,
    typename TArg5,
    typename TArg6
>
```

```
struct ArgTraits<
    HRESULT (STDMETHODCALLTYPE TDelegateInterface::*)
             (TArg1, TArg2, TArg3, TArg4, TArg5, TArg6)>;

template<
    typename TDelegateInterface,
    typename TArg1,
    typename TArg2,
    typename TArg3,
    typename TArg4,
    typename TArg5,
    typename TArg6,
    typename TArg7
>
struct ArgTraits<
    HRESULT (STDMETHODCALLTYPE TDelegateInterface::*)
             (TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7)>;

template<
    typename TDelegateInterface,
    typename TArg1,
    typename TArg2,
    typename TArg3,
    typename TArg4,
    typename TArg5,
    typename TArg6,
    typename TArg7,
    typename TArg8
>
struct ArgTraits<
    HRESULT (STDMETHODCALLTYPE TDelegateInterface::*)
             (TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8)>;

template<
    typename TDelegateInterface,
    typename TArg1,
    typename TArg2,
    typename TArg3,
    typename TArg4,
    typename TArg5,
    typename TArg6,
    typename TArg7,
    typename TArg8,
    typename TArg9
>
struct ArgTraits<
    HRESULT (STDMETHODCALLTYPE TDelegateInterface::*)
             (TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9)>;
```

Parameters

TMemberFunction
Typename parameter for an ArgTraits structure that cannot match any `Invoke` method signature.

TDelegateInterface
A delegate interface.

TArg1
The type of the first argument of the `Invoke` method.

TArg2
The type of the second argument of the `Invoke` method.

TArg3
The type of the third argument of the `Invoke` method.

TArg4
The type of the fourth argument of the `Invoke` method.

TArg5
The type of the fifth argument of the `Invoke` method.

TArg6
The type of the sixth argument of the `Invoke` method.

TArg7
The type of the seventh argument of the `Invoke` method.

TArg8
The type of the eighth argument of the `Invoke` method.

TArg9
The type of the ninth argument of the `Invoke` method.

Remarks

The `ArgTraits` structure declares a specified delegate interface and an anonymous member function that has a specified number of parameters.

Members

Public Typedefs

NAME	DESCRIPTION
`Arg1Type`	The typedef for TArg1.
`Arg2Type`	The typedef for TArg2.
`Arg3Type`	The typedef for TArg3.
`Arg4Type`	The typedef for TArg4.
`Arg5Type`	The typedef for TArg5.
`Arg6Type`	The typedef for TArg6.
`Arg7Type`	The typedef for TArg7.
`Arg8Type`	The typedef for TArg8.
`Arg9Type`	The typedef for TArg9.

Public Constants

NAME	DESCRIPTION
ArgTraits::args	Keeps count of the number of parameters on the `Invoke` method of a delegate interface.

Inheritance Hierarchy

`ArgTraits`

Requirements

Header: event.h

Namespace: Microsoft::WRL::Details

ArgTraits::args

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
static const int args = -1;
```

Remarks

Keeps count of the number of parameters on the `Invoke` method of a delegate interface. When `args` equals -1, there can be no match for the `Invoke` method signature.

ArgTraitsHelper Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template<typename TDelegateInterface>
struct ArgTraitsHelper;
```

Parameters

TDelegateInterface
A delegate interface.

Remarks

Helps define common characteristics of delegate arguments.

Members

Public Typedefs

NAME	DESCRIPTION
`methodType`	A synonym for `decltype(&TDelegateInterface::Invoke)`.
`Traits`	A synonym for `ArgTraits<methodType>`.

Public Constants

NAME	DESCRIPTION
ArgTraitsHelper::args	Helps ArgTraits::args keep count of the number of parameters on the `Invoke` method of a delegate interface.

Inheritance Hierarchy

`ArgTraitsHelper`

Requirements

Header: event.h

Namespace: Microsoft::WRL::Details

ArgTraitsHelper::args

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
static const int args = Traits::args;
```

Remarks

Helps `ArgTraitsHelper::args` keep count of the number of parameters on the `Invoke` method of a delegate interface.

AsyncStatusInternal Enumeration

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
enum AsyncStatusInternal;
```

Remarks

Specifies a mapping between internal enumerations for the state of asynchronous operations and the `Windows::Foundation::AsyncStatus` enumeration.

Members

`_Created`
Equivalent to `::Windows::Foundation::AsyncStatus::Created`

`_Started`
Equivalent to `::Windows::Foundation::AsyncStatus::Started`

`_Completed`
Equivalent to `::Windows::Foundation::AsyncStatus::Completed`

`_Cancelled`
Equivalent to `::Windows::Foundation::AsyncStatus::Cancelled`

`_Error`
Equivalent to `::Windows::Foundation::AsyncStatus::Error`

Requirements

Header: async.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

BoolStruct Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
struct BoolStruct;
```

Remarks

The `BoolStruct` structure defines whether a `ComPtr` is managing the object lifetime of an interface. `BoolStruct` is used internally by the BoolType() operator.

Members

Public Data Members

NAME	DESCRIPTION
BoolStruct::Member	Specifies that a ComPtr is, or is not, managing the object lifetime of an interface.

Inheritance Hierarchy

`BoolStruct`

Requirements

Header: internal.h

Namespace: Microsoft::WRL::Details

BoolStruct::Member

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
int Member;
```

Remarks

Specifies that a ComPtr is, or is not, managing the object lifetime of an interface.

ComPtrRef Class

9/21/2022 • 4 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <typename T>
class ComPtrRef : public ComPtrRefBase<T>;
```

Parameters

T

A ComPtr<T> type or a type derived from it, not merely the interface represented by the `ComPtr`.

Remarks

Represents a reference to an object of type `ComPtr<T>`.

Members

Public Constructors

NAME	DESCRIPTION
ComPtrRef::ComPtrRef	Initializes a new instance of the `ComPtrRef` class from the specified pointer to another `ComPtrRef` object.

Public Methods

NAME	DESCRIPTION
ComPtrRef::GetAddressOf	Retrieves the address of a pointer to the interface represented by the current `ComPtrRef` object.
ComPtrRef::ReleaseAndGetAddressOf	Deletes the current `ComPtrRef` object and returns a pointer-to-a-pointer to the interface that was represented by the `ComPtrRef` object.

Public Operators

NAME	DESCRIPTION
ComPtrRef::operator InterfaceType**	Deletes the current `ComPtrRef` object and returns a pointer-to-a-pointer to the interface that was represented by the `ComPtrRef` object.
ComPtrRef::operator T*	Returns the value of the ptr_ data member of the current ComPtrRef object.

NAME	DESCRIPTION
ComPtrRef::operator void**	Deletes the current `ComPtrRef` object, casts the pointer to the interface that was represented by the `ComPtrRef` object as a pointer-to-pointer-to `void`, and then returns the cast pointer.
ComPtrRef::operator*	Retrieves the pointer to the interface represented by the current `ComPtrRef` object.
ComPtrRef::operator==	Indicates whether two `ComPtrRef` objects are equal.
ComPtrRef::operator!=	Indicates whether two `ComPtrRef` objects are not equal.

Inheritance Hierarchy

`ComPtrRefBase`

`ComPtrRef`

Requirements

Header: client.h

Namespace: Microsoft::WRL::Details

ComPtrRef::ComPtrRef

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
ComPtrRef(
   _In_opt_ T* ptr
);
```

Parameters

ptr
The underlying value of another `ComPtrRef` object.

Remarks

Initializes a new instance of the `ComPtrRef` class from the specified pointer to another `ComPtrRef` object.

ComPtrRef::GetAddressOf

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
InterfaceType* const * GetAddressOf() const;
```

Return Value

Address of a pointer to the interface represented by the current `ComPtrRef` object.

Remarks

Retrieves the address of a pointer to the interface represented by the current `ComPtrRef` object.

ComPtrRef::operator==

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
bool operator==(
   const Details::ComPtrRef<ComPtr<T>>& a,
   const Details::ComPtrRef<ComPtr<U>>& b
);

bool operator==(
   const Details::ComPtrRef<ComPtr<T>>& a,
   decltype(__nullptr)
);

bool operator==(
   decltype(__nullptr),
   const Details::ComPtrRef<ComPtr<T>>& a
);

bool operator==(
   const Details::ComPtrRef<ComPtr<T>>& a,
   void* b
);

bool operator==(
   void* b,
   const Details::ComPtrRef<ComPtr<T>>& a
);
```

Parameters

a

A reference to a `ComPtrRef` object.

b

A reference to another `ComPtrRef` object, or a pointer to an anonymous type (`void*`).

Return Value

The first operator yields `true` if object *a* is equal to object *b*; otherwise, `false`.

The second and third operators yield `true` if object *a* is equal to `nullptr`; otherwise, `false`.

The fourth and fifth operators yield `true` if object *a* is equal to object *b*; otherwise, `false`.

Remarks

Indicates whether two `ComPtrRef` objects are equal.

ComPtrRef::operator!=

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
bool operator!=(
   const Details::ComPtrRef<ComPtr<T>>& a,
   const Details::ComPtrRef<ComPtr<U>>& b
);

bool operator!=(
   const Details::ComPtrRef<ComPtr<T>>& a,
   decltype(__nullptr)
);

bool operator!=(
   decltype(__nullptr),
   const Details::ComPtrRef<ComPtr<T>>& a
);

bool operator!=(
   const Details::ComPtrRef<ComPtr<T>>& a,
   void* b
);

bool operator!=(
   void* b,
   const Details::ComPtrRef<ComPtr<T>>& a
);
```

Parameters

a
A reference to a `ComPtrRef` object.

b
A reference to another `ComPtrRef` object, or a pointer to an anonymous object (`void*`).

Return Value

The first operator yields `true` if object *a* is not equal to object *b*; otherwise, `false`.

The second and third operators yield `true` if object *a* is not equal to `nullptr`; otherwise, `false`.

The fourth and fifth operators yield `true` if object *a* is not equal to object *b*; otherwise, `false`.

Remarks

Indicates whether two `ComPtrRef` objects are not equal.

ComPtrRef::operator InterfaceType**

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
operator InterfaceType**();
```

Remarks

Deletes the current `ComPtrRef` object and returns a pointer-to-a-pointer to the interface that was represented by the `ComPtrRef` object.

ComPtrRef::operator*

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
InterfaceType* operator *();
```

Return Value

Pointer to the interface represented by the current `ComPtrRef` object.

Remarks

Retrieves the pointer to the interface represented by the current `ComPtrRef` object.

ComPtrRef::operator T*

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
operator T*();
```

Remarks

Returns the value of the ptr_ data member of the current `ComPtrRef` object.

ComPtrRef::operator void**

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
operator void**() const;
```

Remarks

Deletes the current `ComPtrRef` object, casts the pointer to the interface that was represented by the `ComPtrRef` object as a pointer-to-pointer-to `void`, and then returns the cast pointer.

ComPtrRef::ReleaseAndGetAddressOf

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
InterfaceType** ReleaseAndGetAddressOf();
```

Return Value

Pointer to the interface that was represented by the deleted `ComPtrRef` object.

Remarks

Deletes the current `ComPtrRef` object and returns a pointer-to-a-pointer to the interface that was represented by the `ComPtrRef` object.

ComPtrRefBase Class

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <typename T>
class ComPtrRefBase;
```

Parameters

T

A ComPtr<T> type or a type derived from it, not merely the interface represented by the `ComPtr`.

Remarks

Represents the base class for the ComPtrRef class.

Members

Public Typedefs

NAME	DESCRIPTION
`InterfaceType`	A synonym for the type of template parameter *T*.

Public Operators

NAME	DESCRIPTION
ComPtrRefBase::operator IInspectable**	Casts the current ptr_ data member to a pointer-to-a-pointer-to the `IInspectable` interface.
ComPtrRefBase::operator IUnknown**	Casts the current ptr_ data member to a pointer-to-a-pointer-to the `IUnknown` interface.

Protected Data Members

NAME	DESCRIPTION
ComPtrRefBase::ptr_	Pointer to the type specified by the current template parameter.

Inheritance Hierarchy

`ComPtrRefBase`

Requirements

Header: client.h

Namespace: Microsoft::WRL::Details

ComPtrRefBase::operator IInspectable** Operator

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
operator IInspectable**() const;
```

Remarks

Casts the current ptr_ data member to a pointer-to-a-pointer-to the `IInspectable` interface.

An error is emitted if the current `ComPtrRefBase` doesn't derive from `IInspectable`.

This cast is available only if `__WRL_CLASSIC_COM__` is defined.

ComPtrRefBase::operator IUnknown** Operator

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
operator IUnknown**() const;
```

Remarks

Casts the current ptr_ data member to a pointer-to-a-pointer-to the `IUnknown` interface.

An error is emitted if the current `ComPtrRefBase` doesn't derive from `IUnknown`.

ComPtrRefBase::ptr_

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
T* ptr_;
```

Remarks

Pointer to the type specified by the current template parameter. `ptr_` is the protected data member.

CreatorMap Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the Windows Runtime C++ Template Library infrastructure and is not intended to be used directly from your code.

Syntax

```
struct CreatorMap;
```

Remarks

Contains information about how to initialize, register, and unregister objects.

`CreatorMap` contains the following information:

- How to initialize, register, and unregister objects.
- How to compare activation data depending on a classic COM or Windows Runtime factory.
- Information about the factory cache and server name for an interface.

Members

Public Data Members

NAME	DESCRIPTION
CreatorMap::activationId	Represents an object ID that is identified either by a classic COM class ID or a Windows Runtime name.
CreatorMap::factoryCache	Stores the pointer to the factory cache for the `CreatorMap`.
CreatorMap::factoryCreator	Creates a factory for the specified `CreatorMap`.
CreatorMap::serverName	Stores the server name for the `CreatorMap`.

Inheritance Hierarchy

`CreatorMap`

Requirements

Header: module.h

Namespace: Microsoft::WRL::Details

CreatorMap::activationId

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
union {
   const IID* clsid;
   const wchar_t* (*getRuntimeName)();
} activationId;
```

Parameters

clsid
An interface ID.

getRuntimeName
A function that retrieves the Windows runtime name of an object.

Remarks

Represents an object ID that is identified either by a classic COM class ID or a Windows runtime name.

CreatorMap::factoryCache

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
FactoryCache* factoryCache;
```

Remarks

Stores the pointer to the factory cache for the `CreatorMap`.

CreatorMap::factoryCreator

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
HRESULT (*factoryCreator)(
   unsigned int* currentflags,
   const CreatorMap* entry,
   REFIID iidClassFactory,
IUnknown** factory);
```

Parameters

currentflags
One of the RuntimeClassType enumerators.

entry
A CreatorMap.

iidClassFactory
The interface ID of a class factory.

factory
When the operation completes, the address of a class factory.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

Remarks

Creates a factory for the specified CreatorMap.

CreatorMap::serverName

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
const wchar_t* serverName;
```

Remarks

Stores the server name for the CreatorMap.

DerefHelper Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <typename T>
struct DerefHelper;

template <typename T>
struct DerefHelper<T*>;
```

Parameters

T

A template parameter.

Remarks

Represent a dereferenced pointer to the `T*` template parameter.

DerefHelper is used in an expression such as:
`ComPtr<Details::DerefHelper<ProgressTraits::Arg1Type>::DerefType> operationInterface;`.

Members

Public Typedefs

NAME	DESCRIPTION
`DerefType`	Identifier for the dereferenced template parameter `T*`.

Inheritance Hierarchy

`DerefHelper`

Requirements

Header: async.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

DontUseNewUseMake Class

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
class DontUseNewUseMake;
```

Remarks

Prevents using operator `new` in `RuntimeClass`. Consequently, you must use the Make function instead.

Members

Public Operators

NAME	DESCRIPTION
DontUseNewUseMake::operator new	Overloads operator `new` and prevents it from being used in `RuntimeClass`.

Inheritance Hierarchy

`DontUseNewUseMake`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

DontUseNewUseMake::operator new

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
void* operator new(
   size_t,
   _In_ void* placement
);
```

Parameters

__unnamed0
An unnamed parameter that specifies the number of bytes of memory to allocate.

placement
The type to be allocated.

Return Value

Provides a way to pass additional arguments if you overload operator `new`.

Remarks

Overloads operator `new` and prevents it from being used in `RuntimeClass`.

EnableIf Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <bool b, typename T = void>
struct EnableIf;

template <typename T>
struct EnableIf<true, T>;
```

Parameters

T
A type.

b
A Boolean expression.

Remarks

Defines a data member of the type specified by the second template parameter if the first template parameter evaluates to `true`.

Members

Public Typedefs

NAME	DESCRIPTION
`type`	If template parameter *b* evaluates to `true`, the partial specialization defines data member `type` to be of type `T`.

Inheritance Hierarchy

`EnableIf`

Requirements

Header: internal.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

EventTargetArray Class

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
class EventTargetArray :
    public Microsoft::WRL::RuntimeClass<
        Microsoft::WRL::RuntimeClassFlags<ClassicCom>,
        IUnknown
    >;
```

Remarks

Represents an array of event handlers.

The event handlers that are associated with an EventSource object are stored in a protected `EventTargetArray` data member.

Members

Public Constructors

NAME	DESCRIPTION
EventTargetArray::EventTargetArray	Initializes a new instance of the `EventTargetArray` class.
EventTargetArray::~EventTargetArray	Deinitializes the current `EventTargetArray` class.

Public Methods

NAME	DESCRIPTION
EventTargetArray::AddTail	Appends the specified event handler to the end of the internal array of event handlers.
EventTargetArray::Begin	Gets the address of the first element in the internal array of event handlers.
EventTargetArray::End	Gets the address of the last element in the internal array of event handlers.
EventTargetArray::Length	Gets the current number of elements in the internal array of event handlers.

Inheritance Hierarchy

`EventTargetArray`

Requirements

Header: event.h

Namespace: Microsoft::WRL::Details

EventTargetArray::~EventTargetArray

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
~EventTargetArray();
```

Remarks

Deinitializes the current `EventTargetArray` class.

EventTargetArray::AddTail

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
void AddTail(
   _In_ IUnknown* element
);
```

Parameters

element
Pointer to the event handler to append.

Remarks

Appends the specified event handler to the end of the internal array of event handlers.

`AddTail()` is intended to be used internally by only the `EventSource` class.

EventTargetArray::Begin

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
ComPtr<IUnknown>* Begin();
```

Return Value

The address of the first element in the internal array of event handlers.

Remarks

Gets the address of the first element in the internal array of event handlers.

EventTargetArray::End

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
ComPtr<IUnknown>* End();
```

Return Value

The address of the last element in the internal array of event handlers.

Remarks

Gets the address of the last element in the internal array of event handlers.

EventTargetArray::EventTargetArray

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
EventTargetArray(
   _Out_ HRESULT* hr,
   size_t items
);
```

Parameters

hr
After this constructor operations, parameter *hr* indicates whether allocation of the array succeeded or failed. The following list shows the possible values for *hr*.

- S_OK
  The operation succeeded.
- E_OUTOFMEMORY
  Memory couldn't be allocated for the array.
- S_FALSE
  Parameter *items* is less than or equal to zero.

items
The number of array elements to allocate.

Remarks

Initializes a new instance of the `EventTargetArray` class.

`EventTargetArray` is used to keep an array of event handlers in an `EventSource` object.

EventTargetArray::Length

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
size_t Length();
```

Return Value

The current number of elements in the internal array of event handlers.

Remarks

Gets the current number of elements in the internal array of event handlers.

FactoryCache Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the Windows Runtime C++ Template Library infrastructure and is not intended to be used directly from your code.

Syntax

```
struct FactoryCache;
```

Remarks

Contains the location of a class factory and a value that identifies a registered wrt or COM class object.

Members

Public Data Members

NAME	DESCRIPTION
FactoryCache::cookie	Contains a value that identifies a registered Windows Runtime or COM class object, and is later used to unregister the object.
FactoryCache::factory	Points to a Windows Runtime or COM class factory.

Inheritance Hierarchy

`FactoryCache`

Requirements

Header: module.h

Namespace: Microsoft::WRL::Details

FactoryCache::cookie

Supports the Windows Runtime C++ Template Library infrastructure and is not intended to be used directly from your code.

```
union {
   WINRT_REGISTRATION_COOKIE winrt;
   DWORD com;
} cookie;
```

Remarks

Contains a value that identifies a registered Windows Runtime or COM class object, and is later used to unregister the object.

FactoryCache::factory

Supports the Windows Runtime C++ Template Library infrastructure and is not intended to be used directly from your code.

```
IUnknown* factory;
```

Remarks

Points to a Windows Runtime or COM class factory.

ImplementsBase Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
struct ImplementsBase;
```

Remarks

Used to validate template parameter types in Implements Structure.

The **ImplementsBase** structure is empty by design.

Inheritance Hierarchy

`ImplementsBase`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

ImplementsHelper Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <typename RuntimeClassFlagsT, typename ILst, bool IsDelegateToClass>
friend struct Details::ImplementsHelper;
```

Parameters

RuntimeClassFlagsT
A field of flags that specifies one or more RuntimeClassType enumerators.

ILst
A list of interface IDs.

IsDelegateToClass
Specify `true` if the current instance of `Implements` is a base class of the first interface ID in *ILst*; otherwise, `false`.

Remarks

Helps implement the Implements structure.

This template traverses a list of interfaces and adds them as base classes, and as information necessary to enable `QueryInterface`.

Members

Protected Methods

NAME	DESCRIPTION
ImplementsHelper::CanCastTo	Gets a pointer to the specified interface ID.
ImplementsHelper::CastToUnknown	Gets a pointer to the underlying `IUnknown` interface for the current `Implements` structure.
ImplementsHelper::FillArrayWithIid	Inserts the interface ID specified by the current zeroth template parameter into the specified array element.
ImplementsHelper::IidCount	Holds the number of implemented interface IDs in the current `Implements` object.

Inheritance Hierarchy

`ImplementsHelper`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

ImplementsHelper::CanCastTo

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
HRESULT CanCastTo(
   REFIID riid,
   _Deref_out_ void **ppv
);

HRESULT CanCastTo(
   _In_ const IID &iid,
   _Deref_out_ void **ppv
);
```

Parameters

riid
Reference to an interface ID.

ppv
If this operation is successful, a pointer to the interface specified by *riid* or *iid*.

iid
Reference to an interface ID.

Return Value

S_OK if successful; otherwise, an HRESULT that indicates the error.

Remarks

Gets a pointer to the specified interface ID.

ImplementsHelper::CastToUnknown

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
IUnknown* CastToUnknown();
```

Return Value

Pointer to the underlying `IUnknown` interface.

Remarks

Gets a pointer to the underlying `IUnknown` interface for the current `Implements` structure.

ImplementsHelper::FillArrayWithIid

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
void FillArrayWithIid(
   _Inout_ unsigned long *index,
   _Inout_ IID* iids) throw();
```

Parameters

index
A zero-based index that indicates the starting array element for this operation. When this operation completes, *index* is incremented by 1.

iids
An array of type IIDs.

Remarks

Inserts the interface ID specified by the current zeroth template parameter into the specified array element.

ImplementsHelper::IidCount

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
static const unsigned long IidCount;
```

Remarks

Holds the number of implemented interface IDs in the current `Implements` object.

InterfaceList Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <typename T, typename U>
struct InterfaceList;
```

Parameters

T

An interface name; the first interface in the recursive list.

U

An interface name; the remaining interfaces in the recursive list.

Remarks

Used to create a recursive list of interfaces.

Members

Public Typedefs

NAME	DESCRIPTION
`FirstT`	Synonym for template parameter *T*.
`RestT`	Synonym for template parameter *U*.

Inheritance Hierarchy

`InterfaceList`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

InterfaceListHelper Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <
    typename T0,
    typename T1 = Nil,
    typename T2 = Nil,
    typename T3 = Nil,
    typename T4 = Nil,
    typename T5 = Nil,
    typename T6 = Nil,
    typename T7 = Nil,
    typename T8 = Nil,
    typename T9 = Nil
>
struct InterfaceListHelper;

template <typename T0>
struct InterfaceListHelper<T0, Nil, Nil, Nil, Nil, Nil, Nil, Nil, Nil>;
```

Parameters

T0

Template parameter 0, which is required.

T1

Template parameter 1, which by default is unspecified.

T2

Template parameter 2, which by default is unspecified.The third template parameter.

T3

Template parameter 3, which by default is unspecified.

T4

Template parameter 4, which by default is unspecified.

T5

Template parameter 5, which by default is unspecified.

T6

Template parameter 6, which by default is unspecified.

T7

Template parameter 7, which by default is unspecified.

T8

Template parameter 8, which by default is unspecified.

T9

Template parameter 9, which by default is unspecified.

Remarks

Builds an `InterfaceList` type by recursively applying the specified template parameter arguments.

The **InterfaceListHelper** template uses template parameter *T0* to define the first data member in an `InterfaceList` structure, and then recursively applies the **InterfaceListHelper** template to any remaining template parameters. **InterfaceListHelper** stops when there are no remaining template parameters.

Members

Public Typedefs

NAME	DESCRIPTION
`TypeT`	A synonym for the InterfaceList type.

Inheritance Hierarchy

`InterfaceListHelper`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

InterfaceTraits Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template<typename I0>
struct __declspec(novtable) InterfaceTraits;

template<typename CloakedType>
struct __declspec(novtable) InterfaceTraits<
    CloakedIid<CloakedType>
>;

template<>
struct __declspec(novtable) InterfaceTraits<Nil>;
```

Parameters

I0
The name of an interface.

CloakedType
For `RuntimeClass`, `Implements` and `ChainInterfaces`, an interface that won't be in the list of supported interface IDs.

Remarks

Implements common characteristics of an interface.

The second template is a specialization for cloaked interfaces. The third template is a specialization for Nil parameters.

Members

Public Typedefs

NAME	DESCRIPTION
`Base`	A synonym for the *I0* template parameter.

Public Methods

NAME	DESCRIPTION
InterfaceTraits::CanCastTo	Indicates whether the specified pointer can be cast to a pointer to `Base`.
InterfaceTraits::CastToBase	Casts the specified pointer to a pointer to `Base`.
InterfaceTraits::CastToUnknown	Casts the specified pointer to a pointer to `IUnknown`.

NAME	DESCRIPTION
InterfaceTraits::FillArrayWithIid	Assigns the interface ID of `Base` to the array element specified by the index argument.
InterfaceTraits::Verify	Verifies that `Base` is properly derived.

Public Constants

NAME	DESCRIPTION
InterfaceTraits::IidCount	Holds the number of interface IDs associated with the current `InterfaceTraits` object.

Inheritance Hierarchy

`InterfaceTraits`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

InterfaceTraits::CanCastTo

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
template<typename T>
static __forceinline bool CanCastTo(
   _In_ T* ptr,
   REFIID riid,
   _Deref_out_ void **ppv
);
```

Parameters

ptr
The name of a pointer to a type.

riid
The interface ID of `Base`.

ppv
If this operation is successful, *ppv* points to the interface specified by `Base`. Otherwise, *ppv* is set to `nullptr`.

Return Value

`true` if this operation is successful and *ptr* is cast to a pointer to `Base`; otherwise, `false`.

Remarks

Indicates whether the specified pointer can be cast to a pointer to `Base`.

For more information about `Base`, see the Public Typedefs section.

InterfaceTraits::CastToBase

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
template<typename T>
static __forceinline Base* CastToBase(
   _In_ T* ptr
);
```

Parameters

T
The type of parameter *ptr*.

ptr
Pointer to a type *T*.

Return Value

A pointer to `Base`.

Remarks

Casts the specified pointer to a pointer to `Base`.

For more information about `Base`, see the Public Typedefs section.

InterfaceTraits::CastToUnknown

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
template<typename T>
static __forceinline IUnknown* CastToUnknown(
   _In_ T* ptr
);
```

Parameters

T
The type of parameter *ptr*.

ptr
Pointer to type *T*.

Return Value

Pointer to the IUnknown from which `Base` is derived.

Remarks

Casts the specified pointer to a pointer to `IUnknown`.

For more information about `Base`, see the Public Typedefs section.

InterfaceTraits::FillArrayWithIid

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
__forceinline static void FillArrayWithIid(
   _Inout_ unsigned long &index,
   _In_ IID* iids
);
```

Parameters

index
Pointer to a field that contains a zero-based index value.

iids
An array of interface IDs.

Remarks

Assigns the interface ID of `Base` to the array element specified by the index argument.

Contrary to the name of this API, only one array element is modified; not the entire array.

For more information about `Base`, see the Public Typedefs section.

InterfaceTraits::IidCount

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
static const unsigned long IidCount = 1;
```

Remarks

Holds the number of interface IDs associated with the current `InterfaceTraits` object.

InterfaceTraits::Verify

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
__forceinline static void Verify();
```

Remarks

Verifies that `Base` is properly derived.

For more information about `Base`, see the Public Typedefs section.

InvokeHelper Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template<typename TDelegateInterface, typename TCallback, unsigned int argCount>
struct InvokeHelper;

template<typename TDelegateInterface, typename TCallback>
struct InvokeHelper<TDelegateInterface, TCallback, 0> :
    public Microsoft::WRL::RuntimeClass<
        RuntimeClassFlags<Delegate>,
        TDelegateInterface
    >;

template<typename TDelegateInterface, typename TCallback>
struct InvokeHelper<TDelegateInterface, TCallback, 1> :
    public Microsoft::WRL::RuntimeClass<
        RuntimeClassFlags<Delegate>,
        TDelegateInterface
    >;

template<typename TDelegateInterface, typename TCallback>
struct InvokeHelper<TDelegateInterface, TCallback, 2> :
    public Microsoft::WRL::RuntimeClass<
        RuntimeClassFlags<Delegate>,
        TDelegateInterface
    >;

template<typename TDelegateInterface, typename TCallback>
struct InvokeHelper<TDelegateInterface, TCallback, 3> :
    public Microsoft::WRL::RuntimeClass<
        RuntimeClassFlags<Delegate>,
        TDelegateInterface
    >;

template<typename TDelegateInterface, typename TCallback>
struct InvokeHelper<TDelegateInterface, TCallback, 4> :
    Microsoft::WRL::RuntimeClass<
        RuntimeClassFlags<Delegate>,
        TDelegateInterface
    >;

template<typename TDelegateInterface, typename TCallback>
struct InvokeHelper<TDelegateInterface, TCallback, 5> :
    Microsoft::WRL::RuntimeClass<
        RuntimeClassFlags<Delegate>,
        TDelegateInterface
    >;

template<typename TDelegateInterface, typename TCallback>
struct InvokeHelper<TDelegateInterface, TCallback, 6> :
    Microsoft::WRL::RuntimeClass<
        RuntimeClassFlags<Delegate>,
        TDelegateInterface
    >;

template<typename TDelegateInterface, typename TCallback>
struct InvokeHelper<TDelegateInterface, TCallback, 7> :
```

```
    Microsoft::WRL::RuntimeClass<
        RuntimeClassFlags<Delegate>,
        TDelegateInterface
    >;

template<typename TDelegateInterface, typename TCallback>
struct InvokeHelper<TDelegateInterface, TCallback, 8> :
    Microsoft::WRL::RuntimeClass<
        RuntimeClassFlags<Delegate>,
        TDelegateInterface
    >;

template<typename TDelegateInterface, typename TCallback>
struct InvokeHelper<TDelegateInterface, TCallback, 9> :
    Microsoft::WRL::RuntimeClass<
        RuntimeClassFlags<Delegate>,
        TDelegateInterface
    >;
```

Parameters

TDelegateInterface
The delegate interface type.

TCallback
The type of the event handler function.

argCount
The number of arguments in an `InvokeHelper` specialization.

Remarks

Provides an implementation of the `Invoke()` method based on the specified number and type of arguments.

Members

Public Typedefs

NAME	DESCRIPTION
`Traits`	A synonym for the class that defines the type of each event handler argument.

Public Constructors

NAME	DESCRIPTION
InvokeHelper::InvokeHelper	Initializes a new instance of the `InvokeHelper` class.

Public Methods

NAME	DESCRIPTION
InvokeHelper::Invoke	Calls the event handler whose signature contains the specified number of arguments.

Public Data Members

NAME	DESCRIPTION
InvokeHelper::callback_	Represents the event handler to call when an event occurs.

Inheritance Hierarchy

`InvokeHelper`

Requirements

Header: event.h

Namespace: Microsoft::WRL::Details

InvokeHelper::callback_

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
TCallback callback_;
```

Remarks

Represents the event handler to call when an event occurs.

The `TCallback` template parameter specifies the type of the event handler.

InvokeHelper::Invoke

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
STDMETHOD(
   Invoke
)();
STDMETHOD(
   Invoke
)(typename Traits;
STDMETHOD(
   Invoke
)( typename Traits;
STDMETHOD(
   Invoke
)( typename Traits;
STDMETHOD(
   Invoke
)( typename Traits;
STDMETHOD(
   Invoke
)( typename Traits;
STDMETHOD(
   Invoke
)( typename Traits;
STDMETHOD(
   Invoke
)( typename Traits;
STDMETHOD(
   Invoke
)( typename Traits;
STDMETHOD(
   Invoke
)( typename Traits;
```

Parameters

arg1
Argument 1.

arg2
Argument 2.

arg3
Argument 3.

arg4
Argument 4.

arg5
Argument 5.

arg6
Argument 6.

arg7
Argument 7.

arg8
Argument 8.

arg9
Argument 9.

Return Value

S_OK if successful; otherwise, an HRESULT that describes the error.

Remarks

Calls the event handler whose signature contains the specified number of arguments.

InvokeHelper::InvokeHelper

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
explicit InvokeHelper(
   TCallback callback
);
```

Parameters

callback
An event handler.

Remarks

Initializes a new instance of the `InvokeHelper` class.

The `TCallback` template parameter specifies the type of the event handler.

IsBaseOfStrict Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <typename Base, typename Derived>
struct IsBaseOfStrict;

template <typename Base>
struct IsBaseOfStrict<Base, Base>;
```

Parameters

Base
The base type.

Derived
The derived type.

Remarks

Tests whether one type is the base of another.

The first template tests whether a type is derived from a base type, which might yield `true` or `false`. The second template tests whether a type is derived from itself, which always yields `false`.

Members

Public Constants

NAME	DESCRIPTION
IsBaseOfStrict::value	Indicates whether one type is the base of another.

Inheritance Hierarchy

`IsBaseOfStrict`

Requirements

Header: internal.h

Namespace: Microsoft::WRL::Details

IsBaseOfStrict::value

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
static const bool value = __is_base_of(Base, Derived);
```

Remarks

Indicates whether one type is the base of another.

`value` is `true` if type `Base` is a base class of the type `Derived`, otherwise it is `false`.

IsSame Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <typename T1, typename T2>
struct IsSame;

template <typename T1>
struct IsSame<T1, T1>;
```

Parameters

T1
A type.

T2
Another type.

Remarks

Tests whether one specified type is the same as another specified type.

Members

Public Constants

NAME	DESCRIPTION
IsSame::value	Indicates whether one type is the same as another.

Inheritance Hierarchy

`IsSame`

Requirements

Header: internal.h

Namespace: Microsoft::WRL::Details

IsSame::value

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
template <typename T1, typename T2>
struct IsSame
{
    static const bool value = false;
};

template <typename T1>
struct IsSame<T1, T1>
{
    static const bool value = true;
};
```

Remarks

Indicates whether one type is the same as another.

`value` is `true` if the template parameters are the same, and `false` if the template parameters are different.

MakeAllocator Class

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template<
    typename T,
    bool hasWeakReferenceSupport =
          !__is_base_of(RuntimeClassFlags<InhibitWeakReference>,
                        T)
>
class MakeAllocator;

template<typename T>
class MakeAllocator<T, false>;

template<typename T>
class MakeAllocator<T, true>;
```

Parameters

T

A type name.

hasWeakReferenceSupport

`true` to allocate memory for an object that supports weak references; `false` to allocate memory for an object that doesn't support weak references.

Remarks

Allocates memory for an activatable class, with or without weak reference support.

Override the `MakeAllocator` class to implement a user-defined memory allocation model.

`MakeAllocator` is typically used to prevent memory leaks if an object throws during construction.

Members

Public Constructors

NAME	DESCRIPTION
MakeAllocator::MakeAllocator	Initializes a new instance of the `MakeAllocator` class.
MakeAllocator::~MakeAllocator	Deinitializes the current instance of the `MakeAllocator` class.

Public Methods

NAME	DESCRIPTION
MakeAllocator::Allocate	Allocates memory and associates it with the current `MakeAllocator` object.
MakeAllocator::Detach	Disassociates memory allocated by the Allocate method from the current `MakeAllocator` object.

Inheritance Hierarchy

`MakeAllocator`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

MakeAllocator::Allocate

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
__forceinline void* Allocate();
```

Return Value

If successful, a pointer to the allocated memory; otherwise, `nullptr`.

Remarks

Allocates memory and associates it with the current `MakeAllocator` object.

The size of the allocated memory is the size of the type specified by the current `MakeAllocator` template parameter.

A developer needs to override only the `Allocate()` method to implement a different memory allocation model.

MakeAllocator::Detach

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
__forceinline void Detach();
```

Remarks

Disassociates memory allocated by the Allocate method from the current `MakeAllocator` object.

If you call `Detach()`, you are responsible for deleting the memory provided by the `Allocate` method.

MakeAllocator::MakeAllocator

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
MakeAllocator();
```

Remarks

Initializes a new instance of the `MakeAllocator` class.

MakeAllocator::~MakeAllocator

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
~MakeAllocator();
```

Remarks

Deinitializes the current instance of the `MakeAllocator` class.

This destructor also deletes the underlying allocated memory if necessary.

MakeAndInitialize Function

9/21/2022 • 2 minutes to read • Edit Online

Initializes the specified Windows Runtime class. Use this function to instantiate a component that is defined in the same module.

Syntax

```
template <
    typename T,
    typename I,
    typename TArg1,
    typename TArg2,
    typename TArg3,
    typename TArg4,
    typename TArg5,
    typename TArg6,
    typename TArg7,
    typename TArg8,
    typename TArg9>
HRESULT MakeAndInitialize(
    _Outptr_result_nullonfailure_ I** ppvObject,
    TArg1 &&arg1,
    TArg2 &&arg2,
    TArg3 &&arg3,
    TArg4 &&arg4,
    TArg5 &&arg5,
    TArg6 &&arg6,
    TArg7 &&arg7,
    TArg8 &&arg8,
    TArg9 &&arg9) throw()
```

Parameters

T

A user-specified class that inherits from `WRL::RuntimeClass`.

TArg1

Type of argument 1 that is passed to the specified runtime class.

TArg2

Type of argument 2 that is passed to the specified runtime class.

TArg3

Type of argument 3 that is passed to the specified runtime class.

TArg4

Type of argument 4 that is passed to the specified runtime class.

TArg5

Type of argument 5 that is passed to the specified runtime class.

TArg6

Type of argument 6 that is passed to the specified runtime class.

TArg7

Type of argument 7 that is passed to the specified runtime class.

TArg8
Type of argument 8 that is passed to the specified runtime class.

TArg9
Type of argument 9 that is passed to the specified runtime class.

arg1
Argument 1 that is passed to the specified runtime class.

arg2
Argument 2 that is passed to the specified runtime class.

arg3
Argument 3 that is passed to the specified runtime class.

arg4
Argument 4 that is passed to the specified runtime class.

arg5
Argument 5 that is passed to the specified runtime class.

arg6
Argument 6 that is passed to the specified runtime class.

arg7
Argument 7 that is passed to the specified runtime class.

arg8
Argument 8 that is passed to the specified runtime class.

arg9
Argument 9 that is passed to the specified runtime class.

Return Value

An HRESULT value.

Remarks

See How to: Instantiate WRL Components Directly to learn the differences between this function and Microsoft::WRL::Make, and for an example.

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

ModuleBase Class

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
class ModuleBase;
```

Remarks

Represents the base class of the Module classes.

Members

Public Constructors

NAME	DESCRIPTION
ModuleBase::ModuleBase	Initializes an instance of the `Module` class.
ModuleBase::~ModuleBase	Deinitializes the current instance of the `Module` class.

Public Methods

NAME	DESCRIPTION
ModuleBase::DecrementObjectCount	When implemented, decrements the number of objects tracked by the module.
ModuleBase::IncrementObjectCount	When implemented, increments the number of objects tracked by the module.

Inheritance Hierarchy

`ModuleBase`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

ModuleBase::~ModuleBase

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
virtual ~ModuleBase();
```

Remarks

Deinitializes the current instance of the `ModuleBase` class.

ModuleBase::DecrementObjectCount

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
virtual long DecrementObjectCount() = 0;
```

Return Value

The count before the decrement operation.

Remarks

When implemented, decrements the number of objects tracked by the module.

ModuleBase::IncrementObjectCount

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
virtual long IncrementObjectCount() = 0;
```

Return Value

The count before the increment operation.

Remarks

When implemented, increments the number of objects tracked by the module.

ModuleBase::ModuleBase

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
ModuleBase();
```

Remarks

Initializes an instance of the `Module` class.

Move Function

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template<class T>
inline typename RemoveReference<T>::Type&& Move(
   _Inout_ T&& arg
);
```

Parameters

T
The type of the argument.

arg
An argument to move.

Return Value

Parameter *arg* after reference or rvalue-reference traits, if any, have been removed.

Remarks

Moves the specified argument from one location to another.

For more information, see the **Move Semantics** section of Rvalue Reference Declarator: &&.

Requirements

Header: internal.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

Nil Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
struct Nil;
```

Remarks

Used to indicate an unspecified, optional template parameter.

Nil is an empty structure.

Inheritance Hierarchy

`Nil`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

RaiseException Function

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
inline void __declspec(noreturn)   RaiseException(
      HRESULT hr,
      DWORD dwExceptionFlags = EXCEPTION_NONCONTINUABLE);
```

Parameters

hr
The exception code of the exception being raised; that is, the HRESULT of a failed operation.

dwExceptionFlags
A flag that indicates a continuable exception (the flag value is zero), or noncontinuable exception (flag value is nonzero). By default, the exception is noncontinuable.

Remarks

Raises an exception in the calling thread.

For more information, see the Windows `RaiseException` function.

Requirements

Header: internal.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

RemoveIUnknown Class

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <typename T>
struct RemoveIUnknown;

template <typename T>
class RemoveIUnknown : public T;
```

Parameters

T
A class.

Remarks

Makes a type that is equivalent to an `IUnknown`-based type, but has nonvirtual `QueryInterface`, `AddRef`, and `Release` member functions.

By default, COM methods provide virtual `QueryInterface`, `AddRef`, and `Release` methods. However, `ComPtr` doesn't require the overhead of virtual methods. `RemoveIUnknown` eliminates that overhead by providing private, nonvirtual `QueryInterface`, `AddRef`, and `Release` methods.

Members

Public Typedefs

NAME	DESCRIPTION
`ReturnType`	A synonym for a type that is equivalent to template parameter *T* but has nonvirtual `IUnknown` members.

Inheritance Hierarchy

`T`

`RemoveIUnknown`

Requirements

Header: client.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

RemoveReference Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template<class T>
struct RemoveReference;

template<class T>
struct RemoveReference<T&>;

template<class T>
struct RemoveReference<T&&>;
```

Parameters

T

A class.

Remarks

Strips the reference or rvalue-reference trait from the specified class template parameter.

Members

Public Typedefs

NAME	DESCRIPTION
`Type`	Synonym for the class template parameter.

Inheritance Hierarchy

`RemoveReference`

Requirements

Header: internal.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

RuntimeClassBase Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
struct RuntimeClassBase;
```

Remarks

Used to detect `RuntimeClass` in the Make function.

RuntimeClassBase is an empty structure.

Inheritance Hierarchy

`RuntimeClassBase`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

RuntimeClassBaseT Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <unsigned int RuntimeClassTypeT>
friend struct Details::RuntimeClassBaseT;
```

Parameters

RuntimeClassTypeT
A field of flags that specifies one or more RuntimeClassType enumerators.

Remarks

Provides helper methods for `QueryInterface` operations and getting interface IDs.

Members

Protected Methods

NAME	DESCRIPTION
RuntimeClassBaseT::AsIID	Retrieves a pointer to the specified interface ID.
RuntimeClassBaseT::GetImplementedIIDS	Retrieves an array of interface IDs that are implemented by a specified type.

Inheritance Hierarchy

`RuntimeClassBaseT`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

RuntimeClassBaseT::AsIID

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
template<typename T>
__forceinline static HRESULT AsIID(
   _In_ T* implements,
   REFIID riid,
   _Deref_out_ void **ppvObject
);
```

Parameters

T

A type that implements the interface ID specified by parameter *riid*.

implements

A variable of the type specified by template parameter *T*.

riid

The interface ID to retrieve.

ppvObject

If this operation is successful, a pointer-to-a-pointer to the interface specified by parameter *riid*.

Return Value

S_OK if successful; otherwise, an HRESULT that describes the error.

Remarks

Retrieves a pointer to the specified interface ID.

RuntimeClassBaseT::GetImplementedIIDS

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
template<typename T>
__forceinline static HRESULT GetImplementedIIDS(
   _In_ T* implements,
   _Out_ ULONG *iidCount,
   _Deref_out_ _Deref_post_cap_(*iidCount) IID **iids
);
```

Parameters

T

The type of the *implements* parameter.

implements

Pointer to the type specified by parameter *T*.

iidCount

The maximum number of interface IDs to retrieve.

iids

If this operation completes successfully, an array of the interface IDs implemented by type *T*.

Return Value

S_OK if successful; otherwise, an HRESULT that describes the error.

Remarks

Retrieves an array of interface IDs that are implemented by a specified type.

Swap Function (WRL)

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
WRL_NOTHROW inline void Swap(
   _Inout_ T& left,
   _Inout_ T& right
);
```

Parameters

left
The first argument.

right
The second argument.

Return Value

Remarks

Exchanges the values of the two specified arguments.

Requirements

Header: internal.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

TerminateMap Function

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
inline bool TerminateMap(
   _In_ ModuleBase *module,
   _In_opt_z_ const wchar_t *serverName,
    bool forceTerminate) throw()
```

Parameters

module
A module.

serverName
The name of a subset of class factories in the module specified by parameter *module*.

forceTerminate
`true` to terminate the class factories regardless of they are active; `false` to not terminate the class factories if any factory is active.

Return Value

`true` if all class factories were terminated; otherwise, `false`.

Remarks

Shuts down the class factories in the specified module.

Requirements

Header: module.h

Namespace: Microsoft::WRL::Details

See also

Microsoft::WRL::Details Namespace

VerifyInheritanceHelper Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <typename I, typename Base>
struct VerifyInheritanceHelper;

template <typename I>
struct VerifyInheritanceHelper<I, Nil>;
```

Parameters

I
A type.

Base
Another type.

Remarks

Tests whether one interface is derived from another interface.

Members

Public Methods

NAME	DESCRIPTION
VerifyInheritanceHelper::Verify	Tests the two interfaces specified by the current template parameters and determines whether one interface is derived from the other.

Inheritance Hierarchy

`VerifyInheritanceHelper`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

VerifyInheritanceHelper::Verify

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
static void Verify();
```

Remarks

Tests the two interfaces specified by the current template parameters and determines whether one interface is derived from the other.

An error is emitted if one interface is not derived from the other.

VerifyInterfaceHelper Structure

9/21/2022 • 2 minutes to read • Edit Online

Supports the Windows Runtime C++ Template Library infrastructure and is not intended to be used directly from your code.

Syntax

```
template <bool isWinRTInterface, typename I>
struct VerifyInterfaceHelper;

template <typename I>
struct VerifyInterfaceHelper<false, I>;
```

Parameters

I

An interface to verify.

isWinRTInterface

Remarks

Verifies that the interface specified by the template parameter meets certain requirements.

Members

Public Methods

NAME	DESCRIPTION
VerifyInterfaceHelper::Verify Method	Verifies that the interface specified by the current template parameter meets certain requirements.

Inheritance Hierarchy

`VerifyInterfaceHelper`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

VerifyInterfaceHelper::Verify

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
static void Verify();
```

Remarks

Verifies that the interface specified by the current template parameter meets certain requirements.

WeakReference Class

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
class WeakReference;
```

Remarks

Represents a *weak reference* that can be used with the Windows Runtime or classic COM. A weak reference represents an object that might or might not be accessible.

A `WeakReference` object maintains a *strong reference*, which is a pointer to an object, and a *strong reference count*, which is the number of copies of the strong reference that have been distributed by the `Resolve()` method. While the strong reference count is nonzero, the strong reference is valid and the object is accessible. When the strong reference count becomes zero, the strong reference is invalid and the object is inaccessible.

A `WeakReference` object is typically used to represent an object whose existence is controlled by an external thread or application. For example, construct a `WeakReference` object from a reference to a file object. While the file is open, the strong reference is valid. But if the file is closed, the strong reference becomes invalid.

The `WeakReference` methods are thread safe.

Members

Public Constructors

NAME	DESCRIPTION
WeakReference::WeakReference	Initializes a new instance of the `WeakReference` class.
WeakReference::~WeakReference	Deinitializes (destroys) the current instance of the `WeakReference` class.

Public Methods

NAME	DESCRIPTION
WeakReference::DecrementStrongReference	Decrements the strong reference count of the current `WeakReference` object.
WeakReference::IncrementStrongReference	Increments the strong reference count of the current `WeakReference` object.
WeakReference::Resolve	Sets the specified pointer to the current strong reference value if the strong reference count is nonzero.

NAME	DESCRIPTION
WeakReference::SetUnknown	Sets the strong reference of the current `WeakReference` object to the specified interface pointer.

Inheritance Hierarchy

`WeakReference`

Requirements

Header: implements.h

Namespace: Microsoft::WRL::Details

WeakReference::~WeakReference

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
virtual ~WeakReference();
```

Return Value

Remarks

Deinitializes the current instance of the `WeakReference` class.

WeakReference::DecrementStrongReference

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
ULONG DecrementStrongReference();
```

Remarks

Decrements the strong reference count of the current `WeakReference` object.

When the strong reference count becomes zero, the strong reference is set to `nullptr`.

Return Value

The decremented strong reference count.

WeakReference::IncrementStrongReference

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
ULONG IncrementStrongReference();
```

Return Value

The incremented strong reference count.

Remarks

Increments the strong reference count of the current `WeakReference` object.

WeakReference::Resolve

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
STDMETHOD(Resolve)
   (REFIID riid,
   _Deref_out_opt_ IInspectable **ppvObject
);
```

Parameters

riid
An interface ID.

ppvObject
When this operation completes, a copy of the current strong reference if the strong reference count is nonzero.

Return Value

- S_OK if this operation is successful and the strong reference count is zero. The *ppvObject* parameter is set to `nullptr`.
- S_OK if this operation is successful and the strong reference count is nonzero. The *ppvObject* parameter is set to the strong reference.
- Otherwise, an HRESULT that indicates the reason this operation failed.

Remarks

Sets the specified pointer to the current strong reference value if the strong reference count is nonzero.

WeakReference::SetUnknown

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
void SetUnknown(
   _In_ IUnknown* unk
);
```

Parameters

unk
A pointer to the `IUnknown` interface of an object.

Remarks

Sets the strong reference of the current `WeakReference` object to the specified interface pointer.

WeakReference::WeakReference

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
WeakReference();
```

Remarks

Initializes a new instance of the `WeakReference` class.

The strong reference pointer for the `WeakReference` object is initialized to `nullptr`, and the strong reference count is initialized to 1.

Microsoft::WRL::Wrappers Namespace

9/21/2022 • 2 minutes to read • Edit Online

Defines Resource Acquisition Is Initialization (RAII) wrapper types that simplify the lifetime management of objects, strings, and handles.

Syntax

```
namespace Microsoft::WRL::Wrappers;
```

Members

Typedefs

NAME	DESCRIPTION
`FileHandle`	`HandleT<HandleTraits::FileHandleTraits>`

Classes

NAME	DESCRIPTION
CriticalSection Class	Represents a critical section object.
Event Class (WRL)	Represents an event.
HandleT Class	Represents a handle to an object.
HString Class	Provides support for manipulating HSTRING handles.
HStringReference Class	Represents an HSTRING that is created from an existing string.
Mutex Class	Represents a synchronization object that exclusively controls a shared resource.
RoInitializeWrapper Class	Initializes the Windows Runtime.
Semaphore Class	Represents a synchronization object that controls a shared resource that can support a limited number of users.
SRWLock Class	Represents a slim reader/writer lock.

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers

See also

Microsoft::WRL Namespace

CriticalSection Class

9/21/2022 • 2 minutes to read • Edit Online

Represents a critical section object.

Syntax

```
class CriticalSection;
```

Members

Constructor

NAME	DESCRIPTION
CriticalSection::CriticalSection	Initializes a synchronization object that is similar to a mutex object, but can be used by only the threads of a single process.
CriticalSection::~CriticalSection	Deinitializes and destroys the current `CriticalSection` object.

Public Methods

NAME	DESCRIPTION
CriticalSection::IsValid	Indicates whether the current critical section is valid.
CriticalSection::Lock	Waits for ownership of the specified critical section object. The function returns when the calling thread is granted ownership.
CriticalSection::TryLock	Attempts to enter a critical section without blocking. If the call is successful, the calling thread takes ownership of the critical section.

Protected Data Members

NAME	DESCRIPTION
CriticalSection::cs_	Declares a critical section data member.

Inheritance Hierarchy

`CriticalSection`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers

CriticalSection::~CriticalSection

Deinitializes and destroys the current `CriticalSection` object.

```
WRL_NOTHROW ~CriticalSection();
```

CriticalSection::CriticalSection

Initializes a synchronization object that is similar to a mutex object, but can be used by only the threads of a single process.

```
explicit CriticalSection(
   ULONG spincount = 0
);
```

Parameters

spincount
The spin count for the critical section object. The default value is 0.

Remarks

For more information about critical sections and spincounts, see the `InitializeCriticalSectionAndSpinCount` function in the `Synchronization` section of the Windows API documenation.

CriticalSection::cs_

Declares a critical section data member.

```
CRITICAL_SECTION cs_;
```

Remarks

This data member is protected.

CriticalSection::IsValid

Indicates whether the current critical section is valid.

```
bool IsValid() const;
```

Return Value

By default, always returns `true`.

CriticalSection::Lock

Waits for ownership of the specified critical section object. The function returns when the calling thread is granted ownership.

```
SyncLock Lock();

   static SyncLock Lock(
   _In_ CRITICAL_SECTION* cs
);
```

Parameters

cs
A user-specified critical section object.

Return Value

A lock object that can be used to unlock the current critical section.

Remarks

The first `Lock` function affects the current critical section object. The second `Lock` function affects a user-specified critical section.

CriticalSection::TryLock

Attempts to enter a critical section without blocking. If the call is successful, the calling thread takes ownership of the critical section.

```
SyncLock TryLock();

static SyncLock TryLock(
   _In_ CRITICAL_SECTION* cs
);
```

Parameters

cs
A user-specified critical section object.

Return Value

A nonzero value if the critical section is successfully entered or the current thread already owns the critical section. Zero if another thread already owns the critical section.

Remarks

The first `TryLock` function affects the current critical section object. The second `TryLock` function affects a user-specified critical section.

Event Class (WRL)

9/21/2022 • 2 minutes to read • Edit Online

Represents an event.

Syntax

```
class Event : public HandleT<HandleTraits::EventTraits>;
```

Members

Public Constructors

NAME	DESCRIPTION
Event::Event	Initializes a new instance of the `Event` class.

Public Operators

NAME	DESCRIPTION
Event::operator=	Assigns the specified `Event` reference to the current `Event` instance.

Inheritance Hierarchy

`HandleT`

`Event`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers

Event::Event

Initializes a new instance of the `Event` class.

```
explicit Event(
   HANDLE h = HandleT::Traits::GetInvalidValue()
);
WRL_NOTHROW Event(
   _Inout_ Event&& h
);
```

Parameters

h

Handle to an event. By default, *h* is initialized to `nullptr`.

Event::operator=

Assigns the specified `Event` reference to the current `Event` instance.

```
WRL_NOTHROW Event& operator=(
   _Inout_ Event&& h
);
```

Parameters

h

An rvalue-reference to an `Event` instance.

Return Value

A pointer to the current `Event` instance.

HandleT Class

9/21/2022 • 2 minutes to read • Edit Online

Represents a handle to an object.

Syntax

```
template <typename HandleTraits>
class HandleT;
```

Parameters

HandleTraits
An instance of the HandleTraits structure that defines common characteristics of a handle.

Members

Public Typedefs

NAME	DESCRIPTION
`Traits`	A synonym for `HandleTraits`.

Public Constructors

NAME	DESCRIPTION
HandleT::HandleT	Initializes a new instance of the `HandleT` class.
HandleT::~HandleT	Deinitializes an instance of the `HandleT` class.

Public Methods

NAME	DESCRIPTION
HandleT::Attach	Associates the specified handle with the current `HandleT` object.
HandleT::Close	Closes the current `HandleT` object.
HandleT::Detach	Disassociates the current `HandleT` object from its underlying handle.
HandleT::Get	Gets the value of the underlying handle.
HandleT::IsValid	Indicates whether the current `HandleT` object represents a handle.

Protected Methods

NAME	DESCRIPTION
HandleT::InternalClose	Closes the current `HandleT` object.

Public Operators

NAME	DESCRIPTION
HandleT::operator=	Moves the value of the specified `HandleT` object to the current `HandleT` object.

Protected Data Members

NAME	DESCRIPTION
HandleT::handle_	Contains the handle that is represented by the `HandleT` object.

Inheritance Hierarchy

`HandleT`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers

HandleT::~HandleT

Deinitializes an instance of the `HandleT` class.

```
~HandleT();
```

HandleT::Attach

Associates the specified handle with the current `HandleT` object.

```
void Attach(
   typename HandleTraits::Type h
);
```

Parameters

h

A handle.

HandleT::Close

Closes the current `HandleT` object.

```
void Close();
```

Remarks

The handle that underlies the current `HandleT` is closed, and the `HandleT` is set to the invalid state.

If the handle doesn't close properly, an exception is raised in the calling thread.

HandleT::Detach

Disassociates the current `HandleT` object from its underlying handle.

```
typename HandleTraits::Type Detach();
```

Return Value

The underlying handle.

Remarks

When this operation completes, the current `HandleT` is set to the invalid state.

HandleT::Get

Gets the value of the underlying handle.

```
typename HandleTraits::Type Get() const;
```

Return Value

A handle.

HandleT::handle_

Contains the handle that is represented by the `HandleT` object.

```
typename HandleTraits::Type handle_;
```

HandleT::HandleT

Initializes a new instance of the `HandleT` class.

```
explicit HandleT(
   typename HandleTraits::Type h =
      HandleTraits::GetInvalidValue()
);

HandleT(
   _Inout_ HandleT&& h
);
```

Parameters

h
A handle.

Remarks

The first constructor initializes a `HandleT` object that is not a valid handle to an object. The second constructor creates a new `HandleT` object from parameter *h*.

HandleT::InternalClose

Closes the current `HandleT` object.

```
virtual bool InternalClose();
```

Return Value

`true` if the current `HandleT` closed successfully; otherwise, `false`.

Remarks

`InternalClose()` is `protected`.

HandleT::IsValid

Indicates whether the current `HandleT` object represents a handle.

```
bool IsValid() const;
```

Return Value

`true` if the `HandleT` represents a handle; otherwise, `false`.

HandleT::operator=

Moves the value of the specified `HandleT` object to the current `HandleT` object.

```
HandleT& operator=(
   _Inout_ HandleT&& h
);
```

Parameters

h
An rvalue-reference to a handle.

Return Value

A reference to the current `HandleT` object.

Remarks

This operation invalidates the `HandleT` object specified by parameter *h*.

HString Class

9/21/2022 • 5 minutes to read • Edit Online

A helper class for managing the lifetime of an HSTRING using the RAII pattern.

Syntax

```
class HString;
```

Remarks

The Windows Runtime provides access to strings through HSTRING handles. The `HString` class provides convenience functions and operators to simplify using HSTRING handles. This class can handle the lifetime of the HSTRING it owns through an RAII pattern.

Members

Public Constructors

NAME	DESCRIPTION
HString::HString	Initializes a new instance of the `HString` class.
HString::~HString	Destroys the current instance of the `HString` class.

Public Methods

NAME	DESCRIPTION
HString::Attach	Associates the specified `HString` object with the current `HString` object.
HString::CopyTo	Copies the current `HString` object to an HSTRING object.
HString::Detach	Disassociates the specified `HString` object from its underlying value.
HString::Get	Retrieves the value of the underlying HSTRING handle.
HString::GetAddressOf	Retrieves a pointer to the underlying HSTRING handle.
HString::GetRawBuffer	Retrieves a pointer to the underlying string data.
HString::IsValid	Indicates whether the current `HString` object is valid.
HString::MakeReference	Creates an `HStringReference` object from a specified string parameter.

NAME	DESCRIPTION
HString::Release	Deletes the underlying string value and intializes the current `HString` object to an empty value.
HString::Set	Sets the value of the current `HString` object to the specified wide-character string or `HString` parameter.

Public Operators

NAME	DESCRIPTION
HString::operator=	Moves the value of another `HString` object to the current `HString` object.
HString::operator==	Indicates whether the two parameters are equal.
HString::operator!=	Indicates whether the two parameters are not equal.
`HString::operator<`	Indicates whether the first parameter is less than the second parameter.

Inheritance Hierarchy

`HString`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers

HString::~HString

Destroys the current instance of the `HString` class.

```
~HString() throw()
```

HString::Attach

Associates the specified `HString` object with the current `HString` object.

```
void Attach(
       HSTRING hstr
       ) throw()
```

Parameters

hstr
An existing `HString` object.

HString::CopyTo

Copies the current `HString` object to an HSTRING object.

```
HRESULT CopyTo(
   _Out_ HSTRING *str
   ) const throw();
```

Parameters

str
The HSTRING that receives the copy.

Remarks

This method calls the WindowsDuplicateString function.

HString::Detach

Disassociates the specified `HString` object from its underlying value.

```
HSTRING Detach() throw()
```

Return Value

The underlying `HString` value before the detach operation started.

HString::Get

Retrieves the value of the underlying HSTRING handle.

```
HSTRING Get() const throw()
```

Return Value

The value of the underlying HSTRING handle

HString::GetAddressOf

Retrieves a pointer to the underlying HSTRING handle.

```
HSTRING* GetAddressOf() throw()
```

Return Value

A pointer to the underlying HSTRING handle.

Remarks

After this operation, the string value of the underlying HSTRING handle is destroyed.

HString::GetRawBuffer

Retrieves a pointer to the underlying string data.

```
const wchar_t* GetRawBuffer(unsigned int* length) const;
```

Parameters

length Pointer to an `int` variable that receives the length of the data.

Return Value

A `const` pointer to the underlying string data.

HString::HString

Initializes a new instance of the `HString` class.

```
HString() throw();
HString(HString&& other) throw();
```

Parameters

hstr
An HSTRING handle.

other
An existing `HString` object.

Remarks

The first constructor initializes a new `HString` object that is empty.

The second constructor initializes a new `HString` object to the value of the existing *other* parameter, and then destroys the *other* parameter.

HString::IsValid

Indicates whether the current `HString` object is empty or not.

```
bool IsValid() const throw()
```

Parameters

`true` if the current `HString` object is not empty; otherwise, `false`.

HString::MakeReference

Creates an `HStringReference` object from a specified string parameter.

```
template<unsigned int sizeDest>
    static HStringReference MakeReference(
              wchar_t const (&str)[ sizeDest]);

    template<unsigned int sizeDest>
    static HStringReference MakeReference(
              wchar_t const (&str)[sizeDest],
              unsigned int len);
```

Parameters

sizeDest
A template parameter that specifies the size of the destination `HStringReference` buffer.

str
A reference to a wide-character string.

len
The maximum length of the *str* parameter buffer to use in this operation. If the *len* parameter isn't specified, the entire *str* parameter is used.

Return Value

An `HStringReference` object whose value is the same as the specified *str* parameter.

HString::operator= Operator

Moves the value of another `HString` object to the current `HString` object.

```
HString& operator=(HString&& other) throw()
```

Parameters

other
An existing `HString` object.

Remarks

The value of the existing *other* object is copied to the current `HString` object, and then the *other* object is destroyed.

HString::operator== Operator

Indicates whether the two parameters are equal.

```
inline bool operator==(
               const HString& lhs,
               const HString& rhs) throw()

inline bool operator==(
                const HString& lhs,
                const HStringReference& rhs) throw()

inline bool operator==(
                const HStringReference& lhs,
                const HString& rhs) throw()

inline bool operator==(
                 const HSTRING& lhs,
                 const HString& rhs) throw()

inline bool operator==(
                 const HString& lhs,
                 const HSTRING& rhs) throw()
```

Parameters

lhs
The first parameter to compare. *lhs* can be an `HString` or `HStringReference` object, or an HSTRING handle.

rhs
The second parameter to compare.*rhs* can be an `HString` or `HStringReference` object, or an HSTRING handle.

Return Value

`true` if the *lhs* and *rhs* parameters are equal; otherwise, `false`.

HString::operator!= Operator

Indicates whether the two parameters are not equal.

```
inline bool operator!=( const HString& lhs,
                        const HString& rhs) throw()

inline bool operator!=( const HStringReference& lhs,
                        const HString& rhs) throw()

inline bool operator!=( const HString& lhs,
                        const HStringReference& rhs) throw()

inline bool operator!=( const HSTRING& lhs,
                        const HString& rhs) throw()

inline bool operator!=( const HString& lhs,
                        const HSTRING& rhs) throw()
```

Parameters

lhs
The first parameter to compare. *lhs* can be an `HString` or `HStringReference` object, or an HSTRING handle.

rhs
The second parameter to compare.*rhs* can be an `HString` or `HStringReference` object, or an HSTRING handle.

Return Value

`true` if the *lhs* and *rhs* parameters are not equal; otherwise, `false`.

`HString::operator<` Operator

Indicates whether the first parameter is less than the second parameter.

```
inline bool operator<(
    const HString& lhs,
    const HString& rhs) throw()
```

Parameters

lhs
The first parameter to compare. *lhs* can be a reference to an `HString`.

rhs
The second parameter to compare. *rhs* can be a reference to an `HString`.

Return Value

`true` if the *lhs* parameter is less than the *rhs* parameter; otherwise, `false`.

HString::Release

Deletes the underlying string value and intializes the current `HString` object to an empty value.

```
void Release() throw()
```

HString::Set

Sets the value of the current `HString` object to the specified wide-character string or `HString` parameter.

```
HRESULT Set(
          const wchar_t* str) throw();
HRESULT Set(
          const wchar_t* str,
          unsigned int len
           ) throw();
HRESULT Set(
          const HSTRING& hstr
           ) throw();
```

Parameters

str

A wide-character string.

len

The maximum length of the *str* parameter that is assigned to the current `HString` object.

hstr

An existing `HString` object.

HStringReference Class

9/21/2022 • 3 minutes to read • Edit Online

Represents an HSTRING that is created from an existing string.

Syntax

```
class HStringReference;
```

Remarks

The lifetime of the backing buffer in the new HSTRING is not managed by the Windows Runtime. The caller allocates a source string on the stack frame to avoid a heap allocation and to eliminate the risk of a memory leak. Also, the caller must ensure that source string remains unchanged during the lifetime of the attached HSTRING. For more information, see WindowsCreateStringReference function.

Members

Public Constructors

NAME	DESCRIPTION
HStringReference::HStringReference	Initializes a new instance of the `HStringReference` class.

Public Methods

MEMBER	DESCRIPTION
HStringReference::CopyTo	Copies the current `HStringReference` object to an HSTRING object.
HStringReference::Get	Retrieves the value of the underlying HSTRING handle.
HStringReference::GetRawBuffer	Retrieves a pointer to the underlying string data.

Public Operators

NAME	DESCRIPTION
HStringReference::operator=	Moves the value of another `HStringReference` object to the current `HStringReference` object.
HStringReference::operator==	Indicates whether the two parameters are equal.
HStringReference::operator!=	Indicates whether the two parameters are not equal.
`HStringReference::operator<`	Indicates whether the first parameter is less than the second parameter.

Inheritance Hierarchy

`HStringReference`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers

HStringReference::CopyTo

Copies the current `HStringReference` object to an HSTRING object.

```
HRESULT CopyTo(
   _Out_ HSTRING *str
   ) const throw();
```

Parameters

str
The HSTRING that receives the copy.

Remarks

This method calls the WindowsDuplicateString function.

HStringReference::Get

Retrieves the value of the underlying HSTRING handle.

```
HSTRING Get() const throw()
```

Return Value

The value of the underlying HSTRING handle.

HStringReference::GetRawBuffer

Retrieves a pointer to the underlying string data.

```
const wchar_t* GetRawBuffer(unsigned int* length) const;
```

Parameters

length Pointer to an `int` variable that receives the length of the data.

Return Value

A `const` pointer to the underlying string data.

HStringReference::HStringReference

Initializes a new instance of the `HStringReference` class.

```
template<unsigned int sizeDest>
HStringReference(wchar_t const (&str)[ sizeDest]) throw();

template<unsigned int sizeDest>
HStringReference(wchar_t const (&str)[ sizeDest],
                 unsigned int len) throw();

HStringReference(HStringReference&& other) throw();
```

Parameters

sizeDest
A template parameter that specifies the size of the destination `HStringReference` buffer.

str
A reference to a wide-character string.

len
The maximum length of the *str* parameter buffer to use in this operation. If the *len* parameter isn't specified, the entire *str* parameter is used. If *len* is greater than *sizeDest*, *len* is set to *sizeDest*-1.

other
Another `HStringReference` object.

Remarks

The first constructor initializes a new `HStringReference` object that the same size as parameter *str*.

The second constructor initializes a new `HStringReference` object that the size specifeid by parameter *len*.

The third constructor initializes a new `HStringReference` object to the value of the *other* parameter, and then destroys the *other* parameter.

HStringReference::operator=

Moves the value of another `HStringReference` object to the current `HStringReference` object.

```
HStringReference& operator=(HStringReference&& other) throw()
```

Parameters

other
An existing `HStringReference` object.

Remarks

The value of the existing *other* object is copied to the current `HStringReference` object, and then the *other* object is destroyed.

HStringReference::operator==

Indicates whether the two parameters are equal.

```
inline bool operator==(
               const HStringReference& lhs,
               const HStringReference& rhs) throw()

inline bool operator==(
               const HSTRING& lhs,
               const HStringReference& rhs) throw()

inline bool operator==(
               const HStringReference& lhs,
               const HSTRING& rhs) throw()
```

Parameters

lhs
The first parameter to compare. *lhs* can be an `HStringReference` object or an HSTRING handle.

rhs
The second parameter to compare. *rhs* can be an `HStringReference` object or an HSTRING handle.

Return Value

`true` if the *lhs* and *rhs* parameters are equal; otherwise, `false`.

HStringReference::operator!=

Indicates whether the two parameters are not equal.

```
inline bool operator!=(
               const HStringReference& lhs,
               const HStringReference& rhs) throw()

inline bool operator!=(
               const HSTRING& lhs,
               const HStringReference& rhs) throw()

inline bool operator!=(
               const HStringReference& lhs,
               const HSTRING& rhs) throw()
```

Parameters

lhs
The first parameter to compare. *lhs* can be an `HStringReference` object or an HSTRING handle.

rhs
The second parameter to compare. *rhs* can be an `HStringReference` object or an HSTRING handle.

Return Value

`true` if the *lhs* and *rhs* parameters are not equal; otherwise, `false`.

`HStringReference::operator<`

Indicates whether the first parameter is less than the second parameter.

```
inline bool operator<(
    const HStringReference& lhs,
    const HStringReference& rhs) throw()
```

Parameters

lhs

The first parameter to compare. *lhs* can be a reference to an `HStringReference`.

rhs

The second parameter to compare. *rhs* can be a reference to an `HStringReference`.

Return Value

`true` if the *lhs* parameter is less than the *rhs* parameter; otherwise, `false`.

Mutex Class

9/21/2022 • 2 minutes to read • Edit Online

Represents a synchronization object that exclusively controls a shared resource.

Syntax

```
class Mutex : public HandleT<HandleTraits::MutexTraits>;
```

Members

Public Typedefs

NAME	DESCRIPTION
`SyncLock`	A synonym for a class that supports synchronous locks.

Public Constructor

NAME	DESCRIPTION
Mutex::Mutex	Initializes a new instance of the `Mutex` class.

Public Members

NAME	DESCRIPTION
Mutex::Lock	Waits until the current object, or the `Mutex` object associated with the specified handle, releases the mutex or the specified time-out interval has elapsed.

Public Operator

NAME	DESCRIPTION
Mutex::operator=	Assigns (moves) the specified `Mutex` object to the current `Mutex` object.

Inheritance Hierarchy

`Mutex`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers

Mutex::Lock

Waits until the current object, or the `Mutex` object associated with the specified handle, releases the mutex or the specified time-out interval has elapsed.

```
SyncLock Lock(
   DWORD milliseconds = INFINITE
);

static SyncLock Lock(
   HANDLE h,
   DWORD milliseconds = INFINITE
);
```

Parameters

milliseconds
The time-out interval, in milliseconds. The default value is INFINITE, which waits indefinitely.

h
The handle of a `Mutex` object.

Return Value

Mutex::Mutex

Initializes a new instance of the `Mutex` class.

```
explicit Mutex(
   HANDLE h
);

Mutex(
   _Inout_ Mutex&& h
);
```

Parameters

h
A handle, or an rvalue-reference to a handle, to a `Mutex` object.

Remarks

The first constructor initializes a `Mutex` object from the specified handle. The second constructor initializes a `Mutex` object from the specified handle, and then moves ownership of the mutex to the current `Mutex` object.

Mutex::operator=

Assigns (moves) the specified `Mutex` object to the current `Mutex` object.

```
Mutex& operator=(
   _Inout_ Mutex&& h
);
```

Parameters

h
An rvalue-reference to a `Mutex` object.

Return Value

A reference to the current `Mutex` object.

Remarks

For more information, see the **Move Semantics** section of Rvalue Reference Declarator: &&.

RoInitializeWrapper Class

9/21/2022 • 2 minutes to read • Edit Online

Initializes the Windows Runtime.

Syntax

```
class RoInitializeWrapper;
```

Remarks

`RoInitializeWrapper` is a convenience that initializes the Windows Runtime and returns an HRESULT that indicates whether the operation was successful. Because the class destructor calls `::Windows::Foundation::Uninitialize`, instances of `RoInitializeWrapper` must be declared at global or top-level scope.

Members

Public Constructors

NAME	DESCRIPTION
RoInitializeWrapper::RoInitializeWrapper	Initializes a new instance of the `RoInitializeWrapper` class.
RoInitializeWrapper::~RoInitializeWrapper	Destroys the current instance of the `RoInitializeWrapper` class.

Public Operators

NAME	DESCRIPTION
RoInitializeWrapper::HRESULT()	Retrieves the HRESULT produced by the `RoInitializeWrapper` constructor.

Inheritance Hierarchy

`RoInitializeWrapper`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers

RoInitializeWrapper::HRESULT()

Retrieves the HRESULT value produced by the last `RoInitializeWrapper` constructor.

```
operator HRESULT()
```

RoInitializeWrapper::RoInitializeWrapper

Initializes a new instance of the `RoInitializeWrapper` class.

```
RoInitializeWrapper(RO_INIT_TYPE flags)
```

Parameters

flags
One of the RO_INIT_TYPE enumerations, which specifies the support provided by the Windows Runtime.

Remarks

The `RoInitializeWrapper` class invokes `Windows::Foundation::Initialize(flags)`.

RoInitializeWrapper::~RoInitializeWrapper

Uninitializes the Windows Runtime.

```
~RoInitializeWrapper()
```

Remarks

The `RoInitializeWrapper` class invokes `Windows::Foundation::Uninitialize()`.

Semaphore Class

9/21/2022 • 2 minutes to read • Edit Online

Represents a synchronization object that controls a shared resource that can support a limited number of users.

Syntax

```
class Semaphore : public HandleT<HandleTraits::SemaphoreTraits>;
```

Members

Public Typedefs

NAME	DESCRIPTION
`SyncLock`	A synonym for a class that supports synchronous locks.

Public Constructors

NAME	DESCRIPTION
Semaphore::Semaphore	Initializes a new instance of the `Semaphore` class.

Public Methods

NAME	DESCRIPTION
Semaphore::Lock	Waits until the current object, or the object associated with the specified handle, is in the signaled state or the specified time-out interval has elapsed.

Public Operators

NAME	DESCRIPTION
Semaphore::operator=	Moves the specified handle from a `Semaphore` object to the current `Semaphore` object.

Inheritance Hierarchy

`Semaphore`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers

Semaphore::Lock

Waits until the current object, or the `Semaphore` object associated with the specified handle, is in the signaled state or the specified time-out interval has elapsed.

```
SyncLock Lock(
   DWORD milliseconds = INFINITE
);

static SyncLock Lock(
   HANDLE h,
   DWORD milliseconds = INFINITE
);
```

Parameters

milliseconds
The time-out interval, in milliseconds. The default value is INFINITE, which waits indefinitely.

h
A handle to a `Semaphore` object.

Return Value

A `Details::SyncLockWithStatusT<HandleTraits::SemaphoreTraits>`

Semaphore::operator=

Moves the specified handle from a `Semaphore` object to the current `Semaphore` object.

```
Semaphore& operator=(
   _Inout_ Semaphore&& h
);
```

Parameters

h
Rvalue-reference to a `Semaphore` object.

Return Value

A reference to the current `Semaphore` object.

Semaphore::Semaphore

Initializes a new instance of the `Semaphore` class.

```
explicit Semaphore(
   HANDLE h
);

WRL_NOTHROW Semaphore(
   _Inout_ Semaphore&& h
);
```

Parameters

h
A handle or an rvalue-reference to a `Semaphore` object.

SRWLock Class

9/21/2022 • 2 minutes to read • Edit Online

Represents a slim reader/writer lock.

Syntax

```
class SRWLock;
```

Remarks

A slim reader/writer lock is used to synchronize access across threads to an object or resource. For more information, see Synchronization Functions.

Members

Public Typedefs

NAME	DESCRIPTION
`SyncLockExclusive`	Synonym for an `SRWLock` object that is acquired in exclusive mode.
`SyncLockShared`	Synonym for an `SRWLock` object that is acquired in shared mode.

Public Constructors

NAME	DESCRIPTION
SRWLock::SRWLock	Initializes a new instance of the `SRWLock` class.
SRWLock::~SRWLock	Deinitializes an instance of the `SRWLock` class.

Public Methods

NAME	DESCRIPTION
SRWLock::LockExclusive	Acquires an `SRWLock` object in exclusive mode.
SRWLock::LockShared	Acquires an `SRWLock` object in shared mode.
SRWLock::TryLockExclusive	Attempts to acquire a `SRWLock` object in exclusive mode for the current or specified `SRWLock` object.
SRWLock::TryLockShared	Attempts to acquire a `SRWLock` object in shared mode for the current or specified `SRWLock` object.

Protected Data Member

NAME	DESCRIPTION
SRWLock::SRWLock_	Contains the underlying lock variable for the current `SRWLock` object.

Inheritance Hierarchy

`SRWLock`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers

SRWLock::~SRWLock

Deinitializes an instance of the `SRWLock` class.

```
~SRWLock();
```

SRWLock::LockExclusive

Acquires an `SRWLock` object in exclusive mode.

```
SyncLockExclusive LockExclusive();

static SyncLockExclusive LockExclusive(
   _In_ SRWLOCK* lock
);
```

Parameters

lock
Pointer to an `SRWLock` object.

Return Value

An `SRWLock` object in exclusive mode.

SRWLock::LockShared

Acquires an `SRWLock` object in shared mode.

```
SyncLockShared LockShared();

static SyncLockShared LockShared(
   _In_ SRWLOCK* lock
);
```

Parameters

lock
Pointer to an `SRWLock` object.

Return Value

An `SRWLock` object in shared mode.

SRWLock::SRWLock

Initializes a new instance of the `SRWLock` class.

```
SRWLock();
```

SRWLock::SRWLock_

Contains the underlying lock variable for the current `SRWLock` object.

```
SRWLOCK SRWLock_;
```

SRWLock::TryLockExclusive

Attempts to acquire a `SRWLock` object in exclusive mode for the current or specified `SRWLock` object. If the call is successful, the calling thread takes ownership of the lock.

```
SyncLockExclusive TryLockExclusive();

static SyncLockExclusive TryLockExclusive(
   _In_ SRWLOCK* lock
);
```

Parameters

lock
Pointer to an `SRWLock` object.

Return Value

If successful, an `SRWLock` object in exclusive mode and the calling thread takes ownership of the lock. Otherwise, an `SRWLock` object whose state is invalid.

SRWLock::TryLockShared

Attempts to acquire a `SRWLock` object in shared mode for the current or specified `SRWLock` object.

```
WRL_NOTHROW SyncLockShared TryLockShared();
WRL_NOTHROW static SyncLockShared TryLockShared(
   _In_ SRWLOCK* lock
);
```

Parameters

lock
Pointer to an `SRWLock` object.

Return Value

If successful, an `SRWLock` object in shared mode and the calling thread takes ownership of the lock. Otherwise, an `SRWLock` object whose state is invalid.

Microsoft::WRL::Wrappers::Details Namespace

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
namespace Microsoft::WRL::Wrappers::Details;
```

Members

Classes

NAME	DESCRIPTION
SyncLockT Class	Represents a type that can take exclusive or shared ownership of a resource.
SyncLockWithStatusT Class	Represents a type that can take exclusive or shared ownership of a resource.

Methods

NAME	DESCRIPTION
CompareStringOrdinal Method	Compares two specified `HSTRING` objects and returns an integer that indicates their relative position in a sort order.

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::Details

See also

Microsoft::WRL::Wrappers Namespace

CompareStringOrdinal Method

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
inline INT32 CompareStringOrdinal(
   HSTRING lhs,
   HSTRING rhs)
```

Parameters

lhs
The first HSTRING to compare.

rhs
The second HSTRING to compare.

Return Value

VALUE	CONDITION
-1	*lhs* is less than *rhs*.
0	*lhs* equals *rhs*.
1	*lhs* is greater than *rhs*.

Remarks

Compares two specified HSTRING objects and returns an integer that indicates their relative position in a sort order.

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::Details

See also

Microsoft::WRL::Wrappers::Details Namespace

SyncLockT Class

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <typename SyncTraits>
class SyncLockT;
```

Parameters

SyncTraits
The type that can take ownership of a resource.

Remarks

Represents a type that can take exclusive or shared ownership of a resource.

The `SyncLockT` class is used, for example, to help implement the SRWLock class.

Members

Public Constructors

NAME	DESCRIPTION
SyncLockT::SyncLockT	Initializes a new instance of the `SyncLockT` class.
SyncLockT::~SyncLockT	Deinitializes an instance of the `SyncLockT` class.

Protected Constructors

NAME	DESCRIPTION
SyncLockT::SyncLockT	Initializes a new instance of the `SyncLockT` class.

Public Methods

NAME	DESCRIPTION
SyncLockT::IsLocked	Indicates whether the current `SyncLockT` object owns a resource; that is, the `SyncLockT` object is *locked*.
SyncLockT::Unlock	Releases control of the resource held by the current `SyncLockT` object, if any.

Protected Data Members

NAME	DESCRIPTION
SyncLockT::sync_	Holds the underlying resource represented by the `SyncLockT` class.

Inheritance Hierarchy

`SyncLockT`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::Details

SyncLockT::~SyncLockT

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
~SyncLockT();
```

Remarks

Deinitializes an instance of the `SyncLockT` class.

This destructor also unlocks the current `SyncLockT` instance.

SyncLockT::IsLocked

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
bool IsLocked() const;
```

Return Value

`true` if the `SyncLockT` object is locked; otherwise, `false`.

Remarks

Indicates whether the current `SyncLockT` object owns a resource; that is, the `SyncLockT` object is *locked*.

SyncLockT::sync_

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
typename SyncTraits::Type sync_;
```

Remarks

Holds the underlying resource represented by the `SyncLockT` class.

SyncLockT::SyncLockT

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
SyncLockT(
   _Inout_ SyncLockT&& other
);

explicit SyncLockT(
   typename SyncTraits::Type sync = SyncTraits::GetInvalidValue()
);
```

Parameters

other
An rvalue-reference to another `SyncLockT` object.

sync
A reference to another `SyncLockWithStatusT` object.

Remarks

Initializes a new instance of the `SyncLockT` class.

The first constructor initializes the current `SyncLockT` object from another `SyncLockT` object specified by parameter *other*, and then invalidates the other `SyncLockT` object. The second constructor is `protected`, and initializes the current `SyncLockT` object to an invalid state.

SyncLockT::Unlock

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
void Unlock();
```

Remarks

Releases control of the resource held by the current `SyncLockT` object, if any.

SyncLockWithStatusT Class

9/21/2022 • 2 minutes to read • Edit Online

Supports the WRL infrastructure and is not intended to be used directly from your code.

Syntax

```
template <typename SyncTraits>
class SyncLockWithStatusT : public SyncLockT<SyncTraits>;
```

Parameters

SyncTraits
A type that can take exclusive or shared ownership of a resource.

Remarks

Represents a type that can take exclusive or shared ownership of a resource.

The `SyncLockWithStatusT` class is used to implement the Mutex and Semaphore classes.

Members

Public Constructors

NAME	DESCRIPTION
SyncLockWithStatusT::SyncLockWithStatusT	Initializes a new instance of the `SyncLockWithStatusT` class.

Protected Constructors

NAME	DESCRIPTION
SyncLockWithStatusT::SyncLockWithStatusT	Initializes a new instance of the `SyncLockWithStatusT` class.

Public Methods

NAME	DESCRIPTION
SyncLockWithStatusT::GetStatus	Retrieves the wait status of the current `SyncLockWithStatusT` object.
SyncLockWithStatusT::IsLocked	Indicates whether the current `SyncLockWithStatusT` object owns a resource; that is, the `SyncLockWithStatusT` object is *locked*.

Protected Data Members

NAME	DESCRIPTION
SyncLockWithStatusT::status_	Holds the result of the underlying wait operation after a lock operation on an object based on the current `SyncLockWithStatusT` object.

Inheritance Hierarchy

`SyncLockT`

`SyncLockWithStatusT`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::Details

SyncLockWithStatusT::GetStatus

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
DWORD GetStatus() const;
```

Return Value

The result of a wait operation on the object that is based on the `SyncLockWithStatusT` class, such as a Mutex or Semaphore. Zero (0) indicates the wait operation returned the signaled state; otherwise, another state occurred, such as time-out value elapsed.

Remarks

Retrieves the wait status of the current `SyncLockWithStatusT` object.

The GetStatus() function retrieves the value of the underlying status_ data member. When an object based on the `SyncLockWithStatusT` class performs a lock operation, the object first waits for the object to become available. The result of that wait operation is stored in the `status_` data member. The possible values of the `status_` data member are the return values of the wait operation. For more information, see the return values of the `WaitForSingleObjectEx` function.

SyncLockWithStatusT::IsLocked

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
bool IsLocked() const;
```

Remarks

Indicates whether the current `SyncLockWithStatusT` object owns a resource; that is, the `SyncLockWithStatusT` object is *locked*.

Return Value

`true` if the `SyncLockWithStatusT` object is locked; otherwise, `false`.

SyncLockWithStatusT::status_

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
DWORD status_;
```

Remarks

Holds the result of the underlying wait operation after a lock operation on an object based on the current `SyncLockWithStatusT` object.

SyncLockWithStatusT::SyncLockWithStatusT

Supports the WRL infrastructure and is not intended to be used directly from your code.

```
SyncLockWithStatusT(
   _Inout_ SyncLockWithStatusT&& other
);

explicit SyncLockWithStatusT(
   typename SyncTraits::Type sync,
   DWORD status
);
```

Parameters

other
An rvalue-reference to another `SyncLockWithStatusT` object.

sync
A reference to another `SyncLockWithStatusT` object.

status
The value of the status_ data member of the *other* parameter or the *sync* parameter.

Remarks

Initializes a new instance of the `SyncLockWithStatusT` class.

The first constructor initializes the current `SyncLockWithStatusT` object from another `SyncLockWithStatusT` specified by parameter *other*, and then invalidates the other `SyncLockWithStatusT` object. The second constructor is `protected`, and initializes the current `SyncLockWithStatusT` object to an invalid state.

Microsoft::WRL::Wrappers::HandleTraits Namespace

9/21/2022 • 2 minutes to read • Edit Online

Describes characteristics of common handle-based resource types.

Syntax

```
namespace Microsoft::WRL::Wrappers::HandleTraits;
```

Members

Structures

NAME	DESCRIPTION
CriticalSectionTraits Structure	Specializes a `CriticalSection` object to support either an invalid critical section or a function to release a critical section.
EventTraits Structure	Defines characteristics of an `Event` class handle.
FileHandleTraits Structure	Defines characteristics of a file handle.
HANDLENullTraits Structure	Defines common characteristics of an uninitialized handle.
HANDLETraits Structure	Defines common characteristics of a handle.
MutexTraits Structure	Defines common characteristics of the Mutex class.
SemaphoreTraits Structure	Defines common characteristics of a Semaphore object.
SRWLockExclusiveTraits Structure	Describes common characteristics of the `SRWLock` class in exclusive lock mode.
SRWLockSharedTraits Structure	Describes common characteristics of the `SRWLock` class in shared lock mode.

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers

See also

Microsoft::WRL::Wrappers Namespace

CriticalSectionTraits Structure

9/21/2022 • 2 minutes to read • Edit Online

Specializes a `CriticalSection` object to support either an invalid critical section or a function to release a critical section.

Syntax

```
struct CriticalSectionTraits;
```

Members

Public Typedefs

NAME	DESCRIPTION
`Type`	A `typedef` that defines a pointer to a critical section. `Type` is defined as `typedef CRITICAL_SECTION* Type;`.

Public Methods

NAME	DESCRIPTION
CriticalSectionTraits::GetInvalidValue	Specializes a `CriticalSection` template so that the template is always invalid.
CriticalSectionTraits::Unlock	Specializes a `CriticalSection` template so that it supports releasing ownership of the specified critical section object.

Inheritance Hierarchy

`CriticalSectionTraits`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::HandleTraits

CriticalSectionTraits::GetInvalidValue

Specializes a `CriticalSection` template so that the template is always invalid.

```
inline static Type GetInvalidValue();
```

Return Value

Always returns a pointer to an invalid critical section.

Remarks

The `Type` modifier is defined as `typedef CRITICAL_SECTION* Type;`.

CriticalSectionTraits::Unlock

Specializes a `CriticalSection` template so that it supports releasing ownership of the specified critical section object.

```
inline static void Unlock(
   _In_ Type cs
);
```

Parameters

cs
A pointer to a critical section object.

Remarks

The `Type` modifier is defined as `typedef CRITICAL_SECTION* Type;`.

For more information, see **LeaveCriticalSection function** in the **Synchronization Functions** section of the Windows API documentation.

EventTraits Structure

9/21/2022 • 2 minutes to read • Edit Online

Defines characteristics of an `Event` class handle.

Syntax

```
struct EventTraits : HANDLENullTraits;
```

Members

Inheritance Hierarchy

`HANDLENullTraits`

`EventTraits`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::HandleTraits

See also

Microsoft::WRL::Wrappers::HandleTraits Namespace

FileHandleTraits Structure

9/21/2022 • 2 minutes to read • Edit Online

Defines characteristics of a file handle.

Syntax

```
struct FileHandleTraits : HANDLETraits;
```

Members

Inheritance Hierarchy

`HANDLETraits`

`FileHandleTraits`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::HandleTraits

See also

Microsoft::WRL::Wrappers::HandleTraits Namespace

HANDLENullTraits Structure

9/21/2022 • 2 minutes to read • Edit Online

Defines common characteristics of an uninitialized handle.

Syntax

```
struct HANDLENullTraits;
```

Members

Public Typedefs

NAME	DESCRIPTION
`Type`	A synonym for HANDLE.

Public Methods

NAME	DESCRIPTION
HANDLENullTraits::Close	Closes the specified handle.
HANDLENullTraits::GetInvalidValue	Represents an invalid handle.

Inheritance Hierarchy

`HANDLENullTraits`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::HandleTraits

HANDLENullTraits::Close

Closes the specified handle.

```
inline static bool Close(
   _In_ Type h
);
```

Parameters

h

The handle to close.

Return Value

`true` if handle *h* closed successfully; otherwise, `false`.

HANDLENullTraits::GetInvalidValue

Represents an invalid handle.

```
inline static Type GetInvalidValue();
```

Return Value

Always returns `nullptr`.

HANDLETraits Structure

9/21/2022 • 2 minutes to read • Edit Online

Defines common characteristics of a handle.

Syntax

```
struct HANDLETraits;
```

Members

Public Typedefs

NAME	DESCRIPTION
`Type`	A synonym for HANDLE.

Public Methods

NAME	DESCRIPTION
HANDLETraits::Close	Closes the specified handle.
HANDLETraits::GetInvalidValue	Represents an invalid handle.

Inheritance Hierarchy

`HANDLETraits`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::HandleTraits

HANDLETraits::Close

Closes the specified handle.

```
inline static bool Close(
   _In_ Type h
);
```

Parameters

h

The handle to close.

Return Value

`true` if handle *h* closed successfully; otherwise, `false`.

HANDLETraits::GetInvalidValue

Represents an invalid handle.

```
inline static HANDLE GetInvalidValue();
```

Return Value

Always returns INVALID_HANDLE_VALUE. (INVALID_HANDLE_VALUE is defined by Windows.)

MutexTraits Structure

9/21/2022 • 2 minutes to read • Edit Online

Defines common characteristics of the Mutex class.

Syntax

```
struct MutexTraits : HANDLENullTraits;
```

Members

Public Methods

NAME	DESCRIPTION
MutexTraits::Unlock	Releases exclusive control of a shared resource.

Inheritance Hierarchy

`HANDLENullTraits`

`MutexTraits`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::HandleTraits

MutexTraits::Unlock Method

Releases exclusive control of a shared resource.

```
inline static void Unlock(
   _In_ Type h
);
```

Parameters

h
Handle to a mutex object.

SemaphoreTraits Structure

9/21/2022 • 2 minutes to read • Edit Online

Defines common characteristics of a `Semaphore` object.

Syntax

```
struct SemaphoreTraits : HANDLENullTraits;
```

Members

Public Methods

NAME	DESCRIPTION
SemaphoreTraits::Unlock	Releases control of a shared resource.

Inheritance Hierarchy

`HANDLENullTraits`

`SemaphoreTraits`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::HandleTraits

SemaphoreTraits::Unlock

Releases control of a shared resource.

```
inline static void Unlock(
   _In_ Type h
);
```

Parameters

h
Handle to a `Semaphore` object.

Remarks

If the unlock operation is unsuccessful, `Unlock()` emits an error that indicates the cause of the failure.

SRWLockExclusiveTraits Structure

9/21/2022 • 2 minutes to read • Edit Online

Describes common characteristics of the `SRWLock` class in exclusive lock mode.

Syntax

```
struct SRWLockExclusiveTraits;
```

Members

Public Typedefs

NAME	DESCRIPTION
`Type`	Synonym for a pointer to the SRWLOCK class.

Public Methods

NAME	DESCRIPTION
SRWLockExclusiveTraits::GetInvalidValue	Retrieves an `SRWLockExclusiveTraits` object that is always invalid.
SRWLockExclusiveTraits::Unlock	Releases exclusive control of the specified `SRWLock` object.

Inheritance Hierarchy

`SRWLockExclusiveTraits`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::HandleTraits

SRWLockExclusiveTraits::GetInvalidValue

Retrieves an `SRWLockExclusiveTraits` object that is always invalid.

```
inline static Type GetInvalidValue();
```

Return Value

An empty `SRWLockExclusiveTraits` object.

SRWLockExclusiveTraits::Unlock

Releases exclusive control of the specified `SRWLock` object.

```
inline static void Unlock(
   _In_ Type srwlock
);
```

Parameters

srwlock
Handle to an `SRWLock` object.

SRWLockSharedTraits Structure

9/21/2022 • 2 minutes to read • Edit Online

Describes common characteristics of the `SRWLock` class in shared lock mode.

Syntax

```
struct SRWLockSharedTraits;
```

Members

Public Typedefs

NAME	DESCRIPTION
`Type`	Synonym for a pointer to the SRWLOCK class.

Public Methods

NAME	DESCRIPTION
SRWLockSharedTraits::GetInvalidValue	Retrieves an `SRWLockSharedTraits` object that is always invalid.
SRWLockSharedTraits::Unlock	Releases exclusive control of the specified `SRWLock` object.

Inheritance Hierarchy

`SRWLockSharedTraits`

Requirements

Header: corewrappers.h

Namespace: Microsoft::WRL::Wrappers::HandleTraits

SRWLockSharedTraits::GetInvalidValue

Retrieves an `SRWLockSharedTraits` object that is always invalid.

```
inline static Type GetInvalidValue();
```

Return Value

A handle to a `SRWLockSharedTraits` object.

SRWLockSharedTraits::Unlock

Releases exclusive control of the specified `SRWLock` object.

```
inline static void Unlock(
   _In_ Type srwlock
);
```

Parameters

srwlock
A handle to an `SRWLock` object.

Windows::Foundation Namespace

9/21/2022 • 2 minutes to read • Edit Online

Enables fundamental Windows Runtime functionality, such as object and factory creation.

Syntax

```
namespace Windows::Foundation;
```

Members

Functions

NAME	DESCRIPTION
ActivateInstance Function	Registers and retrieves an instance of a specified type defined in a specified class ID.
GetActivationFactory Function	Retrieves an activation factory for the type specified by the template parameter.

Requirements

Header: client.h

Namespace: Windows

See also

Windows UWP Namespaces

ActivateInstance Function

9/21/2022 • 2 minutes to read • Edit Online

Registers and retrieves an instance of a specified type defined in a specified class ID.

Syntax

```
template<typename T>
inline HRESULT ActivateInstance(
   _In_ HSTRING activatableClassId,
   _Out_ Microsoft::WRL::Details::ComPtrRef<T> instance
);
```

Parameters

T

A type to activate.

activatableClassId

The name of the class ID that defines parameter *T*.

instance

When this operation completes, a reference to an instance of *T*.

Return Value

S_OK if successful; otherwise, an error HRESULT that indicates the cause of the error.

Requirements

Header: client.h

Namespace: Windows::Foundation

See also

Windows::Foundation Namespace

GetActivationFactory Function

9/21/2022 • 2 minutes to read • Edit Online

Retrieves an activation factory for the type specified by the template parameter.

Syntax

```
template<typename T>
inline HRESULT GetActivationFactory(
   _In_ HSTRING activatableClassId,
   _Out_ Microsoft::WRL::Details::ComPtrRef<T> factory
);
```

Parameters

T

A template parameter that specifies the type of the activation factory.

activatableClassId

The name of the class that the activation factory can produce.

factory

When this operation completes, a reference to the activation factory for type *T*.

Return Value

S_OK if successful; otherwise, an error HRESULT that indicates why this operation failed.

Requirements

Header: client.h

Namespace: Windows::Foundation

See also

Windows::Foundation Namespace

IID_PPV_ARGS_Helper Function

9/21/2022 • 2 minutes to read • Edit Online

Verifies that the type of the specified argument derives from the `IUnknown` interface.

> **IMPORTANT**
> This template specialization supports the WRL infrastructure and is not intended to be used directly from your code. Use IID_PPV_ARGS instead.

Syntax

```
template<typename T>
void** IID_PPV_ARGS_Helper(
   _Inout_ Microsoft::WRL::Details::ComPtrRef<T> pp
);
```

Parameters

T
The type of argument *pp*.

pp
A doubly-indirect pointer.

Return Value

Argument *pp* cast to a pointer-to-a-pointer to `void`.

Remarks

A compile-time error is generated if the template parameter *T* doesn't derive from `IUnknown`.

Requirements

Header: client.h