Component Extensions for .NET and UWP

10/12/20185 minutes to read 

In this article

The C++ standard allows compiler vendors to provide non-standard extensions to the language. Microsoft provides extensions to help you connect native C++ code to code that runs on the .NET Framework or the Universal Windows Platform (UWP). The .NET extensions are called C++/CLI and produce code that executes in the .NET managed execution environment that is called the Common Language Runtime (CLR). The UWP extensions are called C++/CX and they produce native machine code.

> **Note**
>
> For new applications, we recommend using C++/WinRT rather than C++/CX. C++/WinRT is a new, standard C++17 language projection for Windows Runtime APIs. We will continue to support C++/CX and WRL, but highly recommend that new applications use C++/WinRT. For more information, see **C++/WinRT**.

Two runtimes, one set of extensions

C++/CLI extends the ISO/ANSI C++ standard, and is defined under the Ecma C++/CLI Standard. For more information, see .NET Programming with C++/CLI (Visual C++).

The C++/CX extensions are a subset of C++/CLI. Although the extension syntax is identical in most cases, the code that is generated depends on whether you specify the `/ZW` compiler option to target UWP, or the `/clr` option to target .NET. These switches are set automatically when you use Visual Studio to create a project.

Data Type Keywords

The language extensions include *aggregate keywords*, which consist of two tokens separated by white space. The tokens might have one meaning when they are used separately, and another meaning when they are used together. For example, the word "ref" is an ordinary identifier, and the word "class" is a keyword that declares a native class. But when these words are combined to form **ref class**, the resulting aggregate keyword declares an entity that is known as a *runtime class*.

The extensions also include *context-sensitive* keywords. A keyword is treated as context-sensitive depending on the kind of statement that contains it, and its placement in that statement. For example, the token "property" can be an identifier, or it can declare a special kind of public class member.

The following table lists keywords in the C++ language extension.

Keyword	Context sensitive	Purpose	Reference
ref class **ref struct**	No	Declares a class.	Classes and Structs
value class **value struct**	No	Declares a value class.	Classes and Structs
interface class **interface struct**	No	Declares an interface.	interface class
enum class **enum struct**	No	Declares an enumeration.	enum class
property	Yes	Declares a property.	property
delegate	Yes	Declares a delegate.	delegate (C++/CLI and C++/CX)
event	Yes	Declares an event.	event

Override Specifiers

You can use the following keywords to qualify override behavior for derivation. Although the **new** keyword is not an extension of C++, it is listed here because it can be used in an additional context. Some specifiers are also valid for native programming. For more information, see How to: Declare Override Specifiers in Native Compilations (C++/CLI).

Keyword	Context Sensitive	Purpose	Reference

Keyword	Context Sensitive	Purpose	Reference
abstract	Yes	Indicates that functions or classes are abstract.	abstract
new	No	Indicates that a function is not an override of a base class version.	new (new slot in vtable)
override	Yes	Indicates that a method must be an override of a base-class version.	override
sealed	Yes	Prevents classes from being used as base classes.	sealed

Keywords for Generics

The following keywords have been added to support generic types. For more information, see Generics.

Keyword	Context sensitive	Purpose
generic	No	Declares a generic type.
where	Yes	Specifies the constraints that are applied to a generic type parameter.

Miscellaneous Keywords

The following keywords have been added to the C++ extensions.

Keyword	Context sensitive	Purpose	Reference
finally	Yes	Indicates default exception handlings behavior.	Exception Handling
for each, in	No	Enumerates elements of a collection.	for each, in
gcnew	No	Allocates types on the garbage-collected heap. Use instead of **new** and **delete**.	ref new, gcnew

Keyword	Context sensitive	Purpose	Reference
ref new	Yes	Allocates a Windows Runtime type. Use instead of **`new`** and **`delete`**.	ref new, gcnew
initonly	Yes	Indicates that a member can only be initialized at declaration or in a static constructor.	initonly (C++/CLI)
literal	Yes	Creates a literal variable.	literal
`nullptr`	No	Indicates that a handle or pointer does not point at an object.	nullptr

Template Constructs

The following language constructs are implemented as templates, instead of as keywords. If you specify the `/ZW` compiler option, they are defined in the `lang` namespace. If you specify the `/clr` compiler option, they are defined in the `cli` namespace.

Keyword	Purpose	Reference
array	Declares an array.	Arrays
interior_ptr	(CLR only) Points to data in a reference type.	interior_ptr (C++/CLI)
pin_ptr	(CLR only) Points to CLR reference types to temporarily suppress the garbage-collection system.	pin_ptr (C++/CLI)
safe_cast	Determines and executes the optimal casting method for a runtime type.	safe_cast
`typeid`	(CLR only) Retrieves a System.Type object that describes the given type or object.	typeid

Declarators

The following type declarators instruct the runtime to automatically manage the lifetime and deletion of allocated objects.

Operator	Purpose	Reference

Operator	Purpose	Reference
^	Declares a handle to an object; that is, a pointer to a Windows Runtime or CLR object that is automatically deleted when it is no longer usable.	Handle to Object Operator (^)
%	Declares a tracking reference; that is, a reference to a Windows Runtime or CLR object that is automatically deleted when it is no longer usable.	Tracking Reference Operator

Additional Constructs and Related Topics

This section lists additional programming constructs, and topics that pertain to the CLR.

Topic	Description
__identifier (C++/CLI)	(Windows Runtime and CLR) Enables the use of keywords as identifiers.
Variable Argument Lists (...) (C++/CLI)	(Windows Runtime and CLR) Enables a function to take a variable number of arguments.
.NET Framework Equivalents to C++ Native Types (C++/CLI)	Lists the CLR types that are used in place of C++ integral types.
appdomain **__declspec** modifier	**__declspec** modifier that mandates that static and global variables exist per appdomain.
C-Style Casts with /clr (C++/CLI)	Describes how C-style casts are interpreted.
__clrcall calling convention	Indicates the CLR-compliant calling convention.
`__cplusplus_cli`	Predefined Macros
Custom Attributes	Describes how to define your own CLR attributes.
Exception Handling	Provides an overview of exception handling.
Explicit Overrides	Demonstrates how member functions can override arbitrary members.

Topic	Description
Friend Assemblies (C++)	Discusses how a client assembly can access all types in an assembly component.
Boxing	Demonstrates the conditions in which values types are boxed.
Compiler Support for Type Traits	Discusses how to detect characteristics of types at compile time.
managed, unmanagedpragmas	Demonstrates how managed and unmanaged functions can co-exist in the same module.
process `__declspec` modifier	`__declspec` modifier that mandates that static and global variables exist per process.
Reflection (C++/CLI)	Demonstrates the CLR version of run-time type information.
String	Discusses compiler conversion of string literals to String.
Type Forwarding (C++/CLI)	Enables the movement of a type in a shipping assembly to another assembly so that client code does not have to be recompiled.
User-Defined Attributes	Demonstrates user-defined attributes.
#using Directive	Imports external assemblies.
XML Documentation	Explains XML-based code documentation by using /doc (Process Documentation Comments) (C/C++)

See also

.NET Programming with C++/CLI (Visual C++)
Native and .NET Interoperability