Contents

Component Extensions for .NET and UWP

5/13/2022 • 5 minutes to read • Edit Online

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

KEYWORD	CONTEXT SENSITIVE	PURPOSE	REFERENCE
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
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
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
`^`	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-conformant 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.
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.

TOPIC	DESCRIPTION
managed, unmanaged pragmas	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

Tracking Reference Operator (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

A *tracking reference* (`%`) behaves like an ordinary C++ reference (`&`) except that when an object is assigned to a tracking reference, the object's reference count is incremented.

All Platforms

A tracking reference has the following characteristics.

- Assignment of an object to a tracking reference causes the object's reference count to be incremented.
- A native reference (`&`) is the result when you dereference a `*` . A tracking reference (`%`) is the result when you dereference a `^` . As long as you have a `%` to an object, the object will stay alive in memory.
- The dot (`.`) member-access operator is used to access a member of the object.
- Tracking references are valid for value types and handles (for example `String^`).
- A tracking reference cannot be assigned a null or `nullptr` value. A tracking reference may be reassigned to another valid object as many times as required.
- A tracking reference cannot be used as a unary take-address operator.

Windows Runtime

A tracking reference behaves like a standard C++ reference, except that a % is reference-counted. The following snippet shows how to convert between % and ^ types:

```
Foo^ spFoo = ref new Foo();
Foo% srFoo = *spFoo;
Foo^ spFoo2 = %srFoo;
```

The following example shows how to pass a ^ to a function that takes a %.

```
ref class Foo sealed {};

    // internal or private
    void UseFooHelper(Foo% f)
    {
        auto x = %f;
    }

    // public method on ABI boundary
    void UseFoo(Foo^ f)
    {
        if (f != nullptr) { UseFooHelper(*f); }
    }
```

Common Language Runtime

In C++/CLI, you can use a tracking reference to a handle when you bind to an object of a CLR type on the garbage-collected heap.

In the CLR, the value of a tracking reference variable is updated automatically whenever the garbage collector moves the referenced object.

A tracking reference can be declared only on the stack. A tracking reference cannot be a member of a class.

It is not possible to have a native C++ reference to an object on the garbage-collected heap.

For more information about tracking references in C++/CLI, see:

- How to: Use Tracking References in C++/CLI

Examples

The following sample for C++/CLI shows how to use a tracking reference with native and managed types.

```
// tracking_reference_1.cpp
// compile with: /clr
ref class MyClass {
public:
   int i;
};

value struct MyStruct {
   int k;
};

int main() {
   MyClass ^ x = ref new MyClass;
   MyClass ^% y = x;   // tracking reference handle to reference object

   int %ti = x->i;   // tracking reference to member of reference type

   int j = 0;
   int %tj = j;   // tracking reference to object on the stack

   int * pi = new int[2];
   int % ti2 = pi[0];   // tracking reference to object on native heap

   int *% tpi = pi;   // tracking reference to native pointer

   MyStruct ^ x2 = ref new MyStruct;
   MyStruct ^% y2 = x2;   // tracking reference to value object

   MyStruct z;
   int %tk = z.k;   // tracking reference to member of value type

   delete[] pi;
}
```

The following sample for C++/CLI shows how to bind a tracking reference to an array.

```
// tracking_reference_2.cpp
// compile with: /clr
using namespace System;

int main() {
   array<int> ^ a = ref new array<Int32>(5);
   a[0] = 21;
   Console::WriteLine(a[0]);
   array<int> ^% arr = a;
   arr[0] = 222;
   Console::WriteLine(a[0]);
}
```

```
21
222
```

Handle to Object Operator (^) (C++/CLI and C++/CX)

5/13/2022 • 5 minutes to read • Edit Online

The *handle declarator* (`^` , pronounced "hat"), modifies the type specifier to mean that the declared object should be automatically deleted when the system determines that the object is no longer accessible.

Accessing the Declared Object

A variable that is declared with the handle declarator behaves like a pointer to the object. However, the variable points to the entire object, cannot point to a member of the object, and it does not support pointer arithmetic. Use the indirection operator (`*`) to access the object, and the arrow member-access operator (`->`) to access a member of the object.

Windows Runtime

The compiler uses the COM *reference counting* mechanism to determine if the object is no longer being used and can be deleted. This is possible because an object that is derived from a Windows Runtime interface is actually a COM object. The reference count is incremented when the object is created or copied, and decremented when the object is set to null or goes out of scope. If the reference count goes to zero, the object is automatically and immediately deleted.

The advantage of the handle declarator is that in COM you must explicitly manage the reference count for an object, which is a tedious and error prone process. That is, to increment and decrement the reference count you must call the object's AddRef() and Release() methods. However, if you declare an object with the handle declarator, the compiler generates code that automatically adjusts the reference count.

For information on how to instantiate an object, see ref new.

Requirements

Compiler option: `/ZW`

Common Language Runtime

The system uses the CLR *garbage collector* mechanism to determine if the object is no longer being used and can be deleted. The common language runtime maintains a heap on which it allocates objects, and uses managed references (variables) in your program indicate the location of objects on the heap. When an object is no longer used, the memory that it occupied on the heap is freed. Periodically, the garbage collector compacts the heap to better use the freed memory. Compacting the heap can move objects on the heap, which invalidates the locations referred to by managed references. However, the garbage collector is aware of the location of all managed references, and automatically updates them to indicate the current location of the objects on the heap.

Because native C++ pointers (`*`) and references (`&`) are not managed references, the garbage collector cannot automatically update the addresses they point to. To solve this problem, use the handle declarator to specify a variable that the garbage collector is aware of and can update automatically.

For more information, see How to: Declare Handles in Native Types.

Examples

This sample shows how to create an instance of a reference type on the managed heap. This sample also shows that you can initialize one handle with another, resulting in two references to same object on managed, garbage-collected heap. Notice that assigning nullptr to one handle does not mark the object for garbage collection.

```
// mcppv2_handle.cpp
// compile with: /clr
ref class MyClass {
public:
   MyClass() : i(){}
   int i;
   void Test() {
      i++;
      System::Console::WriteLine(i);
   }
};

int main() {
   MyClass ^ p_MyClass = gcnew MyClass;
   p_MyClass->Test();

   MyClass ^ p_MyClass2;
   p_MyClass2 = p_MyClass;

   p_MyClass = nullptr;
   p_MyClass2->Test();
}
```

```
1
2
```

The following sample shows how to declare a handle to an object on the managed heap, where the type of object is a boxed value type. The sample also shows how to get the value type from the boxed object.

```
// mcppv2_handle_2.cpp
// compile with: /clr
using namespace System;

void Test(Object^ o) {
   Int32^ i = dynamic_cast<Int32^>(o);

   if(i)
      Console::WriteLine(i);
   else
      Console::WriteLine("Not a boxed int");
}

int main() {
   String^ str = "test";
   Test(str);

   int n = 100;
   Test(n);
}
```

```
Not a boxed int
100
```

This sample shows that the common C++ idiom of using a `void*` pointer to point to an arbitrary object is replaced by `Object^`, which can hold a handle to any reference class. It also shows that all types, such as arrays and delegates, can be converted to an object handle.

```
// mcppv2_handle_3.cpp
// compile with: /clr
using namespace System;
using namespace System::Collections;
public delegate void MyDel();
ref class MyClass {
public:
   void Test() {}
};

void Test(Object ^ x) {
   Console::WriteLine("Type is {0}", x->GetType());
}

int main() {
   // handle to Object can hold any ref type
   Object ^ h_MyClass = gcnew MyClass;

   ArrayList ^ arr = gcnew ArrayList();
   arr->Add(gcnew MyClass);

   h_MyClass = dynamic_cast<MyClass ^>(arr[0]);
   Test(arr);

   Int32 ^ bi = 1;
   Test(bi);

   MyClass ^ h_MyClass2 = gcnew MyClass;

   MyDel^ DelInst = gcnew MyDel(h_MyClass2, &MyClass::Test);
   Test(DelInst);
}
```

```
Type is System.Collections.ArrayList

Type is System.Int32

Type is MyDel
```

This sample shows that a handle can be dereferenced and that a member can be accessed via a dereferenced handle.

```
// mcppv2_handle_4.cpp
// compile with: /clr
using namespace System;
value struct DataCollection {
private:
   int Size;
   array<String^>^ x;

public:
   DataCollection(int i) : Size(i) {
      x = gcnew array<String^>(Size);
      for (int i = 0 ; i < Size ; i++)
         x[i] = i.ToString();
   }

   void f(int Item) {
      if (Item >= Size)
      {
         System::Console::WriteLine("Cannot access array element {0}, size is {1}", Item, Size);
         return;
      }
      else
         System::Console::WriteLine("Array value: {0}", x[Item]);
   }
};

void f(DataCollection y, int Item) {
   y.f(Item);
}

int main() {
   DataCollection ^ a = gcnew DataCollection(10);
   f(*a, 7);   // dereference a handle, return handle's object
   (*a).f(11);   // access member via dereferenced handle
}
```

```
Array value: 7

Cannot access array element 11, size is 10
```

This sample shows that a native reference (`&`) can't bind to an `int` member of a managed type, as the `int` might be stored in the garbage collected heap, and native references don't track object movement in the managed heap. The fix is to use a local variable, or to change `&` to `%`, making it a tracking reference.

```
// mcppv2_handle_5.cpp
// compile with: /clr
ref struct A {
   void Test(unsigned int &){}
   void Test2(unsigned int %){}
   unsigned int i;
};

int main() {
   A a;
   a.i = 9;
   a.Test(a.i);   // C2664
   a.Test2(a.i);   // OK

   unsigned int j = 0;
   a.Test(j);   // OK
}
```

Requirements

Compiler option: `/clr`

See also

Component Extensions for .NET and UWP
Tracking Reference Operator

abstract (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

The **abstract** keyword declares either:

- A type can be used as a base type, but the type itself cannot be instantiated.
- A type member function can be defined only in a derived type.

All Platforms

Syntax

class-declaration class-identifier **abstract {}**

`virtual` *return-type member-function-identifier* **() abstract ;**

Remarks

The first example syntax declares a class to be abstract. The *class-declaration* component can be either a native C++ declaration (** `class` **** or `struct`), or a C++ extension declaration (**ref class** or **ref struct**) if the `/ZW` or `/clr` compiler option is specified.

The second example syntax declares a virtual member function to be abstract. Declaring a function abstract is the same as declaring it a pure virtual function. Declaring a member function abstract also causes the enclosing class to be declared abstract.

The **abstract** keyword is supported in native and platform-specific code; that is, it can be compiled with or without the `/ZW` or `/clr` compiler option.

You can detect at compile time if a type is abstract with the `__is_abstract(type)` type trait. For more information, see Compiler Support for Type Traits.

The **abstract** keyword is a context-sensitive override specifier. For more information about context-sensitive keywords, see Context-Sensitive Keywords. For more information about override specifiers, see How to: Declare Override Specifiers in Native Compilations.

Windows Runtime

For more information, see Ref classes and structs.

Requirements

Compiler option: `/ZW`

Common Language Runtime

Requirements

Compiler option: `/clr`

Examples

The following code example generates an error because class `X` is marked **abstract**.

```
// abstract_keyword.cpp
// compile with: /clr
ref class X abstract {
public:
   virtual void f() {}
};

int main() {
   X ^ MyX = gcnew X;   // C3622
}
```

The following code example generates an error because it instantiates a native class that is marked **abstract**. This error will occur with or without the `/clr` compiler option.

```
// abstract_keyword_2.cpp
class X abstract {
public:
   virtual void f() {}
};

int main() {
   X * MyX = new X; // C3622: 'X': a class declared as 'abstract'
                    // cannot be instantiated. See declaration of 'X'}
```

The following code example generates an error because function `f` includes a definition but is marked **abstract**. The final statement in the example shows that declaring an abstract virtual function is equivalent to declaring a pure virtual function.

```
// abstract_keyword_3.cpp
// compile with: /clr
ref class X {
public:
   virtual void f() abstract {}   // C3634
   virtual void g() = 0 {}   // C3634
};
```

See also

Component Extensions for .NET and UWP

Arrays (C++/CLI and C++/CX)

5/13/2022 • 4 minutes to read • Edit Online

The `Platform::Array<T>` type in C++/CX, or the **array** keyword in C++/CLI, declares an array of a specified type and initial value.

All Platforms

The array must be declared by using the handle-to-object (^) modifier after the closing angle bracket (>) in the declaration. The number of elements of the array is not part of the type. One array variable can refer to arrays of different sizes.

Unlike standard C++, subscripting is not a synonym for pointer arithmetic and is not commutative.

For more information about arrays, see:

- How to: Use Arrays in C++/CLI
- Variable Argument Lists (...) (C++/CLI)

Windows Runtime

Arrays are members of the `Platform` namespace. Arrays can be only one-dimensional.

Syntax

The first example of the syntax uses the **ref new** aggregate keyword to allocate an array. The second example declares a local array.

```
[qualifiers] [Platform::]Array<[qualifiers] array-type [,rank]>^ identifier =
    ref new[Platform::]Array<initialization-type> [{initialization-list [,...]}]

[qualifiers] [Platform::]Array<[qualifiers] array-type [,rank]>^ identifier =
    {initialization-list [,...]}
```

qualifiers
(Optional) One or more of these storage class specifiers: mutable, volatile, const, extern, static.

array-type
The type of the array variable. Valid types are Windows Runtime classes and fundamental types, ref classes and structs, value classes and structs, and native pointers (`type*`).

rank
(Optional) The number of dimensions of the array. Must be 1.

identifier
The name of the array variable.

initialization-type
The type of the values that initialize the array. Typically, *array-type* and *initialization-type* are the same type. However, the types can be different if there is a conversion from *initialization-type* to *array-type*—for example, if *initialization-type* is derived from *array-type*.

initialization-list

(Optional) A comma-delimited list of values in curly brackets that initialize the elements of the array. For example, if *rank-size-list* were `(3)`, which declares a one-dimensional array of 3 elements, *initialization list* could be `{1,2,3}`.

Remarks

You can detect at compile time whether a type is a reference-counted array with `__is_ref_array(type)`. For more information, see Compiler Support for Type Traits.

Requirements

Compiler option: `/ZW`

Examples

The following example creates a one-dimensional array that has 100 elements.

```
// cwr_array.cpp
// compile with: /ZW
using namespace Platform;
ref class MyClass {};
int main() {
   // one-dimensional array
   Array<MyClass^>^ My1DArray = ref new Array<MyClass^>(100);
   My1DArray[99] = ref new MyClass();
}
```

Common Language Runtime

Syntax

The first example of the syntax uses the **gcnew** keyword to allocate an array. The second example declares a local array.

```
[qualifiers] [cli::]array<[qualifiers] array-type [,rank]>^ identifier =
    gcnew [cli::]array<initialization-type[,rank]>(rank-size-list[,...]) [{initialization-list [,...]}]

[qualifiers] [cli::]array<[qualifiers] array-type [,rank]>^ identifier =
    {initialization-list [,...]}
```

qualifiers
(Optional) One or more of these storage class specifiers: mutable, volatile, const, extern, static.

array-type
The type of the array variable. Valid types are Windows Runtime classes and fundamental types, ref classes and structs, value classes and structs, native pointers (`type*`), and native POD (plain old data) types.

rank
(Optional) The number of dimensions of the array. The default is 1; the maximum is 32. Each dimension of the array is itself an array.

identifier
The name of the array variable.

initialization-type
The type of the values that initialize the array. Typically, *array-type* and *initialization-type* are the same type. However, the types can be different if there is a conversion from *initialization-type* to *array-type*—for example, if *initialization-type* is derived from *array-type*.

rank-size-list

A comma-delimited list of the size of each dimension in the array. Alternatively, if the *initialization-list* parameter is specified, the compiler can deduce the size of each dimension and *rank-size-list* can be omitted.

initialization-list
(Optional) A comma-delimited list of values in curly brackets that initialize the elements of the array. Or a comma-delimited list of nested *initialization-list* items that initialize the elements in a multi-dimensional array.

For example, if *rank-size-list* were `(3)`, which declares a one-dimensional array of 3 elements, *initialization list* could be `{1,2,3}`. If *rank-size-list* were `(3,2,4)`, which declares a three-dimensional array of 3 elements in the first dimension, 2 elements in the second, and 4 elements in the third, *initialization-list* could be `{{1,2,3},{0,0},{-5,10,-21,99}}`.)

Remarks

array is in the Platform, default, and cli Namespaces namespace.

Like standard C++, the indices of an array are zero-based, and an array is subscripted by using square brackets ([]). Unlike standard C++, the indices of a multi-dimensional array are specified in a list of indices for each dimension instead of a set of square-bracket ([]) operators for each dimension. For example, *identifier*[*index1*, *index2*] instead of *identifier*[*index1*][*index2*].

All managed arrays inherit from `System::Array`. Any method or property of `System::Array` can be applied directly to the array variable.

When you allocate an array whose element type is pointer-to a managed class, the elements are 0-initialized.

When you allocate an array whose element type is a value type `V`, the default constructor for `V` is applied to each array element. For more information, see .NET Framework Equivalents to C++ Native Types (C++/CLI).

At compile time, you can detect whether a type is a common language runtime (CLR) array with `__is_ref_array(type)`. For more information, see Compiler Support for Type Traits.

Requirements

Compiler option: `/clr`

Examples

The following example creates a one-dimensional array that has 100 elements, and a three-dimensional array that has 3 elements in the first dimension, 5 elements in the second, and 6 elements in the third.

```
// clr_array.cpp
// compile with: /clr
ref class MyClass {};
int main() {
   // one-dimensional array
   array<MyClass ^> ^ My1DArray = gcnew array<MyClass ^>(100);
   My1DArray[99] = gcnew MyClass();

   // three-dimensional array
   array<MyClass ^, 3> ^ My3DArray = gcnew array<MyClass ^, 3>(3, 5, 6);
   My3DArray[0,0,0] = gcnew MyClass();
}
```

See also

Component Extensions for .NET and UWP

Boxing (C++/CLI and C++/CX)

5/13/2022 • 3 minutes to read • Edit Online

The conversion of value types to objects is called *boxing*, and the conversion of objects to value types is called *unboxing*.

All Runtimes

(There are no remarks for this language feature that apply to all runtimes.)

Windows Runtime

C++/CX supports a shorthand syntax for boxing value types and unboxing reference types. A value type is boxed when it is assigned to a variable of type `Object`. An `Object` variable is unboxed when it is assigned to a value type variable and the unboxed type is specified in parentheses; that is, when the object variable is cast to a value type.

```
   Platform::Object^
   object_variable  = value_variable;
value_variable = (value_type) object_variable;
```

Requirements

Compiler option: `/ZW`

Examples

The following code example boxes and unboxes a `DateTime` value. First, the example obtains a `DateTime` value that represents the current date and time and assigns it to a `DateTime` variable. Then the `DateTime` is boxed by assigning it to an `Object` variable. Finally, the boxed value is unboxed by assigning it to another `DateTime` variable.

To test the example, create a `BlankApplication` project, replace the `BlankPage::OnNavigatedTo()` method, and then specify breakpoints at the closing bracket and the assignment to variable `str1`. When the example reaches the closing bracket, examine `str1`.

```
void BlankPage::OnNavigatedTo(NavigationEventArgs^ e)
{
    using namespace Windows::Globalization::DateTimeFormatting;

    Windows::Foundation::DateTime dt, dtAnother;
    Platform::Object^ obj1;

    Windows::Globalization::Calendar^ c =
        ref new Windows::Globalization::Calendar;
    c->SetToNow();
    dt = c->GetDateTime();
    auto dtf = ref new DateTimeFormatter(
                          YearFormat::Full,
                          MonthFormat::Numeric,
                          DayFormat::Default,
                          DayOfWeekFormat::None);
    String^ str1 = dtf->Format(dt);
    OutputDebugString(str1->Data());
    OutputDebugString(L"\r\n");

    // Box the value type and assign to a reference type.
    obj1 = dt;
    // Unbox the reference type and assign to a value type.
    dtAnother = (Windows::Foundation::DateTime) obj1;

    // Format the DateTime for display.
    String^ str2 = dtf->Format(dtAnother);
    OutputDebugString(str2->Data());
}
```

For more information, see Boxing (C++/CX).

Common Language Runtime

The compiler boxes value types to Object. This is possible because of a compiler-defined conversion to convert value types to Object.

Boxing and unboxing enable value types to be treated as objects. Value types, including both struct types and built-in types such as int, can be converted to and from the type Object.

For more information, see:

- How to: Explicitly Request Boxing
- How to: Use gcnew to Create Value Types and Use Implicit Boxing
- How to: Unbox
- Standard Conversions and Implicit Boxing

Requirements

Compiler option: `/clr`

Examples

The following sample shows how implicit boxing works.

```
// vcmcppv2_explicit_boxing2.cpp
// compile with: /clr
using namespace System;

ref class A {
public:
   void func(System::Object^ o){Console::WriteLine("in A");}
};

value class V {};

interface struct IFace {
   void func();
};

value class V1 : public IFace {
public:
   virtual void func() {
      Console::WriteLine("Interface function");
   }
};

value struct V2 {
   // conversion operator to System::Object
   static operator System::Object^(V2 v2) {
      Console::WriteLine("operator System::Object^");
      return (V2^)v2;
   }
};

void func1(System::Object^){Console::WriteLine("in void func1(System::Object^)");}
void func1(V2^){Console::WriteLine("in func1(V2^)");}

void func2(System::ValueType^){Console::WriteLine("in func2(System::ValueType^)");}
void func2(System::Object^){Console::WriteLine("in func2(System::Object^)");}

int main() {
   // example 1 simple implicit boxing
   Int32^ bi = 1;
   Console::WriteLine(bi);

   // example 2 calling a member with implicit boxing
   Int32 n = 10;
   Console::WriteLine("xx = {0}", n.ToString());

   // example 3 implicit boxing for function calls
   A^ a = gcnew A;
   a->func(n);

   // example 4 implicit boxing for WriteLine function call
   V v;
   Console::WriteLine("Class {0} passed using implicit boxing", v);
   Console::WriteLine("Class {0} passed with forced boxing", (V^)(v));   // force boxing

   // example 5 casting to a base with implicit boxing
   V1 v1;
   IFace ^ iface = v1;
   iface->func();

   // example 6 user-defined conversion preferred over implicit boxing for function-call parameter matching
   V2 v2;
   func1(v2);   // user defined conversion from V2 to System::Object preferred over implicit boxing
                // Will call void func1(System::Object^);

   func2(v2);   // OK: Calls "static V2::operator System::Object^(V2 v2)"
   func2((V2^)v2);   // Using explicit boxing: calls func2(System::ValueType^)
}
```

```
1

xx = 10

in A

Class V passed using implicit boxing

Class V passed with forced boxing

Interface function

in func1(V2^)

in func2(System::ValueType^)

in func2(System::ValueType^)
```

See also

Component Extensions for .NET and UWP

ref class and ref struct (C++/CLI and C++/CX)

5/13/2022 • 3 minutes to read • Edit Online

The **ref class** or **ref struct** extensions declare a class or struct whose *object lifetime* is administered automatically. When the object is no longer accessible or goes out of scope, the memory is released.

All Runtimes

Syntax

```
class_access ref class name modifier : inherit_access base_type {};
class_access ref struct name modifier : inherit_access base_type {};
class_access value class name modifier : inherit_access base_type {};
class_access value struct name modifier : inherit_access base_type {};
```

Parameters

class_access
(Optional) The accessibility of the class or struct outside the assembly. Possible values are `public` and `private` (`private` is the default). Nested classes or structs cannot have a *class_access* specifier.

name
The name of the class or struct.

modifier
(Optional) abstract and sealed are valid modifiers.

inherit_access
(Optional) The accessibility of *base_type*. The only permitted accessibility is `public` (`public` is the default).

base_type
(Optional) A base type. However, a value type cannot act as a base type.

For more information, see the language-specific descriptions of this parameter in the Windows Runtime and Common Language Runtime sections.

Remarks

The default member accessibility of an object declared with **ref class** or **value class** is `private`. And the default member accessibility of an object declared with **ref struct** or **value struct** is `public`.

When a reference type inherits from another reference type, virtual functions in the base class must explicitly be overridden (with override) or hidden (with new (new slot in vtable)). The derived class functions must also be explicitly marked as `virtual`.

To detect at compile time whether a type is a **ref class** or **ref struct**, or a **value class** or **value struct**, use `__is_ref_class (type)`, `__is_value_class (type)`, or `__is_simple_value_class (type)`. For more information, see Compiler Support for Type Traits.

For more information on classes and structs, see

- Instantiating Classes and Structs
- C++ Stack Semantics for Reference Types
- Classes, Structures, and Unions

- Destructors and finalizers in How to: Define and consume classes and structs (C++/CLI)
- User-Defined Operators (C++/CLI)
- User-Defined Conversions (C++/CLI)
- How to: Wrap Native Class for Use by C#
- Generic Classes (C++/CLI)

Windows Runtime

Remarks

See Ref classes and structs and Value classes and structs.

Parameters

base_type
(Optional) A base type. A **ref class** or **ref struct** can inherit from zero or more interfaces and zero or one **ref** types. A **value class** or **value struct** can only inherit from zero or more interfaces.

When you declare an object by using the **ref class** or **ref struct** keywords, the object is accessed by a handle to an object; that is, a reference-counter pointer to the object. When the declared variable goes out of scope, the compiler automatically deletes the underlying object. When the object is used as a parameter in a call or is stored in a variable, a handle to the object is actually passed or stored.

When you declare an object by using the **value class** or **value struct** keywords, the object lifetime of the declared object is not supervised. The object is like any other standard C++ class or struct.

Requirements

Compiler option: `/ZW`

Common Language Runtime

Remarks

The following table lists differences from the syntax shown in the **All Runtimes** section that are specific to C++/CLI.

Parameters

base_type
(Optional) A base type. A **ref class** or **ref struct** can inherit from zero or more managed interfaces and zero or one ref types. A **value class** or **value struct** can only inherit from zero or more managed interfaces.

The **ref class** and **ref struct** keywords tell the compiler that the class or structure is to be allocated on the heap. When the object is used as a parameter in a call or is stored in a variable, a reference to the object is actually passed or stored.

The **value class** and **value struct** keywords tells the compiler that the value of the allocated class or structure is passed to functions or stored in members.

Requirements

Compiler option: `/clr`

See also

Component Extensions for .NET and UWP

Platform, default, and cli Namespaces (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

A namespace qualifies the names of language elements so the names do not conflict with otherwise identical names elsewhere in the source code. For example, a name collision might prevent the compiler from recognizing Context-Sensitive Keywords. Namespaces are used by the compiler but are not preserved in the compiled assembly.

All Runtimes

Visual Studio provides a default namespace for your project when you create the project. You can manually rename the namespace, although in C++/CX the name of the .winmd file must match the name of the root namespace.

Windows Runtime

For more information, see Namespaces and type visibility (C++/CX).

Requirements

Compiler option: `/ZW`

Common Language Runtime

Syntax

```
using namespace cli;
```

Remarks

The C++/CLI supports the **cli** namespace. When compiling with `/clr`, the `using` statement in the Syntax section is implied.

The following language features are in the **cli** namespace:

- Arrays
- interior_ptr (C++/CLI)
- pin_ptr (C++/CLI)
- safe_cast

Requirements

Compiler option: `/clr`

Examples

The following code example demonstrates that it is possible to use a symbol in the **cli** namespace as a user-defined symbol in your code. However, once you have done so, you will have to explicitly or implicitly qualify your references to the **cli** language element of the same name.

```
// cli_namespace.cpp
// compile with: /clr
using namespace cli;
int main() {
   array<int> ^ MyArray = gcnew array<int>(100);
   int array = 0;

   array<int> ^ MyArray2 = gcnew array<int>(100);   // C2062

   // OK
   cli::array<int> ^ MyArray2 = gcnew cli::array<int>(100);
   ::array<int> ^ MyArray3 = gcnew ::array<int>(100);
}
```

See also

Component Extensions for .NET and UWP

Compiler Support for Type Traits (C++/CLI and C++/CX)

5/13/2022 • 6 minutes to read • Edit Online

The Microsoft C++ compiler supports *type traits* for C++/CLI and C++/CX extensions, which indicate various characteristics of a type at compile time.

All Runtimes

Remarks

Type traits are especially useful to programmers who write libraries.

The following list contains the type traits that are supported by the compiler. All type traits return `false` if the condition specified by the name of the type trait is not met.

(In the following list, code examples are written only in C++/CLI. But the corresponding type trait is also supported in C++/CX unless stated otherwise. The term, "platform type" refers to either Windows Runtime types or common language runtime types.)

- `__has_assign(` *type* `)`

  Returns `true` if the platform or native type has a copy assignment operator.

  ```
  ref struct R {
  void operator=(R% r) {}
  };

  int main() {
  System::Console::WriteLine(__has_assign(R));
  }
  ```

- `__has_copy(` *type* `)`

  Returns `true` if the platform or native type has a copy constructor.

  ```
  ref struct R {
  R(R% r) {}
  };

  int main() {
  System::Console::WriteLine(__has_copy(R));
  }
  ```

- `__has_finalizer(` *type* `)`

  (Not supported in C++/CX.) Returns `true` if the CLR type has a finalizer. See Destructors and finalizers in How to: Define and consume classes and structs (C++/CLI) for more information.

```
using namespace System;
ref struct R {
~R() {}
protected:
!R() {}
};

int main() {
Console::WriteLine(__has_finalizer(R));
}
```

- `__has_nothrow_assign(` *type* `)`

  Returns `true` if a copy assignment operator has an empty exception specification.

```
#include <stdio.h>
struct S {
void operator=(S& r) throw() {}
};

int main() {
__has_nothrow_assign(S) == true ?
printf("true\n") : printf("false\n");
}
```

- `__has_nothrow_constructor(` *type* `)`

  Returns `true` if the default constructor has an empty exception specification.

```
#include <stdio.h>
struct S {
S() throw() {}
};

int main() {
__has_nothrow_constructor(S) == true ?
printf("true\n") : printf("false\n");
}
```

- `__has_nothrow_copy(` *type* `)`

  Returns `true` if the copy constructor has an empty exception specification.

```
#include <stdio.h>
struct S {
S(S& r) throw() {}
};

int main() {
__has_nothrow_copy(S) == true ?
printf("true\n") : printf("false\n");
}
```

- `__has_trivial_assign(` *type* `)`

  Returns `true` if the type has a trivial, compiler-generated assignment operator.

```
#include <stdio.h>
struct S {};

int main() {
__has_trivial_assign(S) == true ?
printf("true\n") : printf("false\n");
}
```

- `__has_trivial_constructor(` *type* `)`

  Returns `true` if the type has a trivial, compiler-generated constructor.

  ```
  #include <stdio.h>
  struct S {};

  int main() {
  __has_trivial_constructor(S) == true ?
  printf("true\n") : printf("false\n");
  }
  ```

- `__has_trivial_copy(` *type* `)`

  Returns `true` if the type has a trivial, compiler-generated copy constructor.

  ```
  #include <stdio.h>
  struct S {};

  int main() {
  __has_trivial_copy(S) == true ?
  printf("true\n") : printf("false\n");
  }
  ```

- `__has_trivial_destructor(` *type* `)`

  Returns `true` if the type has a trivial, compiler-generated destructor.

  ```
  // has_trivial_destructor.cpp
  #include <stdio.h>
  struct S {};

  int main() {
  __has_trivial_destructor(S) == true ?
  printf("true\n") : printf("false\n");
  }
  ```

- `__has_user_destructor(` *type* `)`

  Returns `true` if the platform or native type has a user-declared destructor.

```
// has_user_destructor.cpp

using namespace System;
ref class R {
~R() {}
};

int main() {
Console::WriteLine(__has_user_destructor(R));
}
```

- `__has_virtual_destructor(` *type* `)`

  Returns `true` if the type has a virtual destructor.

  `__has_virtual_destructor` also works on platform types, and any user-defined destructor in a platform type is a virtual destructor.

```
// has_virtual_destructor.cpp
#include <stdio.h>
struct S {
virtual ~S() {}
};

int main() {
__has_virtual_destructor(S) == true ?
printf("true\n") : printf("false\n");
}
```

- `__is_abstract(` *type* `)`

  Returns `true` if the type is an abstract type. For more information on native abstract types, see Abstract Classes.

  `__is_abstract` also works for platform types. An interface with at least one member is an abstract type, as is a reference type with at least one abstract member. For more information on abstract platform types, see abstract.

```
// is_abstract.cpp
#include <stdio.h>
struct S {
virtual void Test() = 0;
};

int main() {
__is_abstract(S) == true ?
printf("true\n") : printf("false\n");
}
```

- `__is_base_of(` `base` `,` `derived` `)`

  Returns `true` if the first type is a base class of the second type, or if both types are the same.

  `__is_base_of` also works on platform types. For example, it will return `true` if the first type is an interface class and the second type implements the interface.

```
// is_base_of.cpp
#include <stdio.h>
struct S {};
struct T : public S {};

int main() {
__is_base_of(S, T) == true ?
printf("true\n") : printf("false\n");

__is_base_of(S, S) == true ?
printf("true\n") : printf("false\n");
}
```

- `__is_class(` *type* `)`

  Returns `true` if the type is a native class or struct.

```
#include <stdio.h>
struct S {};

int main() {
__is_class(S) == true ?
printf("true\n") : printf("false\n");
}
```

- `__is_convertible_to(` `from` `,` `to` `)`

  Returns `true` if the first type can be converted to the second type.

```
#include <stdio.h>
struct S {};
struct T : public S {};

int main() {
S * s = new S;
T * t = new T;
s = t;
__is_convertible_to(T, S) == true ?
printf("true\n") : printf("false\n");
}
```

- `__is_delegate(` *type* `)`

  Returns `true` if `type` is a delegate. For more information, see delegate (C++/CLI and C++/CX).

```
delegate void MyDel();
int main() {
System::Console::WriteLine(__is_delegate(MyDel));
}
```

- `__is_empty(` *type* `)`

  Returns `true` if the type has no instance data members.

```
#include <stdio.h>
struct S {
int Test() {}
static int i;
};
int main() {
__is_empty(S) == true ?
printf("true\n") : printf("false\n");
}
```

- `__is_enum(` *type* `)`

  Returns `true` if the type is a native enum.

  ```
  // is_enum.cpp
  #include <stdio.h>
  enum E { a, b };

  struct S {
  enum E2 { c, d };
  };

  int main() {
  __is_enum(E) == true ?
  printf("true\n") : printf("false\n");

  __is_enum(S::E2) == true ?
  printf("true\n") : printf("false\n");
  }
  ```

- `__is_interface_class(` *type* `)`

  Returns `true` if passed a platform interface. For more information, see interface class.

  ```
  // is_interface_class.cpp

  using namespace System;
  interface class I {};
  int main() {
  Console::WriteLine(__is_interface_class(I));
  }
  ```

- `__is_pod(` *type* `)`

  Returns `true` if the type is a class or union with no constructor or private or protected non-static members, no base classes, and no virtual functions. See the C++ standard, sections 8.5.1/1, 9/4, and 3.9/10 for more information on PODs.

  `__is_pod` will return false on fundamental types.

  ```
  #include <stdio.h>
  struct S {};

  int main() {
  __is_pod(S) == true ?
  printf("true\n") : printf("false\n");
  }
  ```

- `__is_polymorphic(` *type* `)`

Returns `true` if a native type has virtual functions.

```
#include <stdio.h>
struct S {
virtual void Test(){}
};

int main() {
__is_polymorphic(S) == true ?
printf("true\n") : printf("false\n");
}
```

- `__is_ref_array(` *type* `)`

  Returns `true` if passed a platform array. For more information, see Arrays.

```
using namespace System;
int main() {
array<int>^ x = gcnew array<int>(10);
Console::WriteLine(__is_ref_array(array<int>));
}
```

- `__is_ref_class(` *type* `)`

  Returns `true` if passed a reference class. For more information on user-defined reference types, see Classes and Structs.

```
using namespace System;
ref class R {};
int main() {
Console::WriteLine(__is_ref_class(Buffer));
Console::WriteLine(__is_ref_class(R));
}
```

- `__is_sealed(` *type* `)`

  Returns `true` if passed a platform or native type marked sealed. For more information, see sealed.

```
ref class R sealed{};
int main() {
System::Console::WriteLine(__is_sealed(R));
}
```

- `__is_simple_value_class(` *type* `)`

  Returns `true` if passed a value type that contains no references to the garbage-collected heap. For more information on user-defined value types, see Classes and Structs.

```
using namespace System;
ref class R {};
value struct V {};
value struct V2 {
R ^ r;   // not a simnple value type
};

int main() {
Console::WriteLine(__is_simple_value_class(V));
Console::WriteLine(__is_simple_value_class(V2));
}
```

- `__is_union(` *type* `)`

  Returns `true` if a type is a union.

```
#include <stdio.h>
union A {
int i;
float f;
};

int main() {
__is_union(A) == true ?
printf("true\n") : printf("false\n");
}
```

- `__is_value_class(` *type* `)`

  Returns `true` if passed a value type. For more information on user-defined value types, see Classes and Structs.

```
value struct V {};

int main() {
System::Console::WriteLine(__is_value_class(V));
}
```

Windows Runtime

Remarks

The `__has_finalizer(` *type* `)` type trait is not supported because this platform does not support finalizers.

Requirements

Compiler option: `/ZW`

Common Language Runtime

Remarks

(There are no platform-specific remarks for this feature.)

Requirements

Compiler option: `/clr`

Examples

Example

The following code example shows how to use a class template to expose a compiler type trait for a `/clr` compilation. For more information, see Windows Runtime and Managed Templates.

```
// compiler_type_traits.cpp
// compile with: /clr
using namespace System;

template <class T>
ref struct is_class {
   literal bool value = __is_ref_class(T);
};

ref class R {};

int main () {
   if (is_class<R>::value)
      Console::WriteLine("R is a ref class");
   else
      Console::WriteLine("R is not a ref class");
}
```

```
R is a ref class
```

See also

Component Extensions for .NET and UWP

Context-Sensitive Keywords (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

Context-sensitive keywords are language elements that are recognized only in specific contexts. Outside the specific context, a context-sensitive keyword can be a user-defined symbol.

All Runtimes

Remarks

The following is a list of context-sensitive keywords:

- abstract
- delegate
- event
- finally
- for each, in
- initonly
- `internal`
- literal
- override
- property
- sealed
- `where` (part of Generics)

For readability purposes, you may want to limit your use of context-sensitive keywords as user-defined symbols.

Windows Runtime

Remarks

(There are no platform-specific remarks for this feature.)

Requirements

Compiler option: `/ZW`

Common Language Runtime

Remarks

(There are no platform-specific remarks for this feature.)

Requirements

Compiler option: `/clr`

Examples

The following code example shows that in the appropriate context, the `property` context-sensitive keyword can be used to define a property and a variable.

```
// context_sensitive_keywords.cpp
// compile with: /clr
public ref class C {
   int MyInt;
public:
   C() : MyInt(99) {}

   property int Property_Block {   // context-sensitive keyword
      int get() { return MyInt; }
   }
};

int main() {
   int property = 0;               // variable name
   C ^ MyC = gcnew C();
   property = MyC->Property_Block;
   System::Console::WriteLine(++property);
}
```

```
100
```

See also

Component Extensions for .NET and UWP

delegate (C++/CLI and C++/CX)

5/13/2022 • 3 minutes to read • Edit Online

Declares a type that represents a function pointer.

All Runtimes

Both the Windows Runtime and common language runtime support delegates.

Remarks

delegate is a context-sensitive keyword. For more information, see Context-Sensitive Keywords.

To detect at compile time if a type is a delegate, use the `__is_delegate()` type trait. For more information, see Compiler Support for Type Traits.

Windows Runtime

C++/CX supports delegates with the following syntax.

Syntax

```
access
delegate
return-type
delegate-type-identifier
(
[ parameters ]
)
```

Parameters

access
(optional) The accessibility of the delegate, which can be `public` (the default) or `private`. The function prototype can also be qualified with the `const` or `volatile` keywords.

return-type
The return type of the function prototype.

delegate-type-identifier
The name of the declared delegate type.

parameters
(Optional) The types and identifiers of the function prototype.

Remarks

Use the *delegate-type-identifier* to declare an event with the same prototype as the delegate. For more information, see Delegates (C++/CX).

Requirements

Compiler option: `/ZW`

Common Language Runtime

The common language runtime supports delegates with the following syntax.

Syntax

```
access
delegate
function_declaration
```

Parameters

access
(optional) The accessibility of the delegate outside of the assembly can be public or private. The default is private. Inside a class, a delegate can have any accessibility.

function_declaration
The signature of the function that can be bound to the delegate. The return type of a delegate can be any managed type. For interoperability reasons, it is recommended that the return type of a delegate be a CLS type.

To define an unbound delegate, the first parameter in *function_declaration* should be the type of the `this` pointer for the object.

Remarks

Delegates are multicast: the "function pointer" can be bound to one or more methods within a managed class. The **delegate** keyword defines a multicast delegate type with a specific method signature.

A delegate can also be bound to a method of a value class, such as a static method.

A delegate has the following characteristics:

- It inherits from `System::MulticastDelegate`.
- It has a constructor that takes two arguments: a pointer to a managed class or NULL (in the case of binding to a static method) and a fully qualified method of the specified type.
- It has a method called `Invoke`, whose signature matches the declared signature of the delegate.

When a delegate is invoked, its function(s) are called in the order they were attached.

The return value of a delegate is the return value from its last attached member function.

Delegates cannot be overloaded.

Delegates can be bound or unbound.

When you instantiate a bound delegate, the first argument shall be an object reference. The second argument of a delegate instantiation shall either be the address of a method of a managed class object, or a pointer to a method of a value type. The second argument of a delegate instantiation must name the method with the full class scope syntax and apply the address-of operator.

When you instantiate an unbound delegate, the first argument shall either be the address of a method of a managed class object, or a pointer to a method of a value type. The argument must name the method with the full class scope syntax and apply the address-of operator.

When creating a delegate to a static or global function, only one parameter is required: the function (optionally, the address of the function).

For more information on delegates, see

- How to: Define and Use Delegates (C++/CLI)
- Generic Delegates (C++/CLI)

Requirements

Compiler option: `/clr`

Examples

The following example shows how to declare, initialize, and invoke delegates.

```
// mcppv2_delegate.cpp
// compile with: /clr
using namespace System;

// declare a delegate
public delegate void MyDel(int i);

ref class A {
public:
   void func1(int i) {
      Console::WriteLine("in func1 {0}", i);
   }

   void func2(int i) {
      Console::WriteLine("in func2 {0}", i);
   }

   static void func3(int i) {
      Console::WriteLine("in static func3 {0}", i);
   }
};

int main () {
   A ^ a = gcnew A;

   // declare a delegate instance
   MyDel^ DelInst;

   // test if delegate is initialized
   if (DelInst)
      DelInst(7);

   // assigning to delegate
   DelInst = gcnew MyDel(a, &A::func1);

   // invoke delegate
   if (DelInst)
      DelInst(8);

   // add a function
   DelInst += gcnew MyDel(a, &A::func2);

   DelInst(9);

   // remove a function
   DelInst -= gcnew MyDel(a, &A::func1);

   // invoke delegate with Invoke
   DelInst->Invoke(10);

   // make delegate to static function
   MyDel ^ StaticDelInst = gcnew MyDel(&A::func3);
   StaticDelInst(11);
}
```

```
in func1 8

in func1 9

in func2 9

in func2 10

in static func3 11
```

See also

Component Extensions for .NET and UWP

enum class (C++/CLI and C++/CX)

5/13/2022 • 4 minutes to read • Edit Online

Declares an enumeration at namespace scope, which is a user-defined type consisting of a set of named constants called enumerators.

All Runtimes

Remarks

C++/CX and C++/CLI support **public enum class** and **private enum class** which are similar to the standard C++ **enum class** but with the addition of the accessibility specifier. Under **/clr**, the C++11 **enum class** type is permitted but will generate warning C4472 which is intended to ensure that you really want the ISO enum type and not the C++/CX and C++/CLI type. For more information about the ISO Standard C++ `enum` keyword, see Enumerations.

Windows Runtime

Syntax

```
      access
      enum class
      enumeration-identifier
      [:underlying-type] { enumerator-list } [var];
accessenum structenumeration-identifier[:underlying-type] { enumerator-list } [var];
```

Parameters

access
The accessibility of the enumeration, which can be `public` or `private`.

enumeration-identifier
The name of the enumeration.

underlying-type
(Optional) The underlying type of the enumeration.

(Optional. Windows Runtime only) The underlying type of the enumeration, which can be `bool`, `char`, `char16`, `int16`, `uint16`, `int`, `uint32`, `int64`, or `uint64`.

enumerator-list
A comma-delimited list of enumerator names.

The value of each enumerator is a constant expression that is either defined implicitly by the compiler, or explicitly by the notation, *enumerator* `=` *constant-expression*. By default, the value of the first enumerator is zero if it is implicitly defined. The value of each subsequent implicitly-defined enumerator is the value of the previous enumerator + 1.

var
(Optional) The name of a variable of the enumeration type.

Remarks

For more information, and examples, see Enums.

Note that the compiler emits error messages if the constant expression that defines the value of an enumerator cannot be represented by the *underlying-type*. However, the compiler does not report an error for a value that is inappropriate for the underlying type. For example:

- If *underlying-type* is numeric, and an enumerator specifies the maximum value for that type, the value of the next implicitly defined enumeration cannot be represented.
- If *underlying-type* is `bool`, and more than two enumerators are implicitly defined, the enumerators after the first two cannot be represented.
- If *underlying-type* is `char16`, and the enumeration value ranges from 0xD800 through 0xDFFF, the value can be represented. However, the value logically incorrect because it represents half a Unicode surrogate pair and should not appear in isolation.

Requirements

Compiler option: `/ZW`

Common Language Runtime

Syntax

```
      access
      enum class
      name [:type] { enumerator-list } var;
accessenum structname [:type] { enumerator-list } var;
```

Parameters

access
The accessibility of the enum. Can be either `public` or `private`.

enumerator-list
A comma-separated list of the identifiers (enumerators) in the enumeration.

name
The name of the enumeration. Anonymous managed enumerations are not allowed.

type
(Optional) The underlying type of the *identifiers*. This can be any scalar type, such as signed or unsigned versions of `int`, `short`, or `long`. `bool` or `char` is also allowed.

var
(Optional) The name of a variable of the enumeration type.

Remarks

enum class and **enum struct** are equivalent declarations.

There are two types of enums: managed or C++/CX and standard.

A managed or C++/CX enum might be defined as follows,

```
public enum class day {sun, mon };
```

and is semantically equivalent to:

```
ref class day {
public:
   static const int sun = 0;
   static const int mon = 1;
};
```

A standard enum might be defined as follows:

```
enum day2 { sun, mon };
```

and is semantically equivalent to:

```
static const int sun = 0;
static const int mon = 1;
```

Managed enumerator names (*identifiers*) are not injected into the scope where the enumeration is defined; all references to the enumerators must be fully qualified (*name* `::` *identifier*). For this reason, you cannot define an anonymous managed enum.

The enumerators of a standard enum are strongly injected into the enclosing scope. That is, if there is another symbol with the same name as an enumerator in the enclosing scope, the compiler will generate an error.

In Visual Studio 2002 and Visual Studio 2003, enumerators were weakly injected (visible in the enclosing scope unless there was another identifier with the same name).

If a standard C++ enum is defined (without `class` or `struct`), compiling with `/clr` will cause the enumeration to be compiled as a managed enum. The enumeration still has the semantics of an unmanaged enumeration. Note, the compiler injects an attribute, `Microsoft::VisualC::NativeEnumAttribute` to identify a programmer's intent for the enum to be a native enum. Other compilers will simply see the standard enum as a managed enum.

A named, standard enum compiled with `/clr` will be visible in the assembly as a managed enum, and can be consumed by any other managed compiler. However, an unnamed standard enum will not be publicly visible from the assembly.

In Visual Studio 2002 and Visual Studio 2003, a standard enum used as the type in a function parameter:

```
// mcppv2_enum.cpp
// compile with: /clr
enum E { a, b };
void f(E) {System::Console::WriteLine("hi");}

int main() {
   E myi = b;
   f(myi);
}
```

would emit the following in MSIL for the function signature:

```
void f(int32);
```

However, in current versions of the compiler, the standard enum is emitted as a managed enum with a [NativeEnumAttribute] and the following in MSIL for the function signature:

```
void f(E)
```

For more information about native enums, see C++ Enumeration Declarations.

For more information on CLR enums, see:

- Underlying Type of an Enum

Requirements

Compiler option: `/clr`

Examples

```
// mcppv2_enum_2.cpp
// compile with: /clr
// managed enum
public enum class m { a, b };

// standard enum
public enum n { c, d };

// unnamed, standard enum
public enum { e, f } o;

int main()
{
   // consume managed enum
   m mym = m::b;
   System::Console::WriteLine("no automatic conversion to int: {0}", mym);
   System::Console::WriteLine("convert to int: {0}", (int)mym);

   // consume standard enum
   n myn = d;
   System::Console::WriteLine(myn);

   // consume standard, unnamed enum
   o = f;
   System::Console::WriteLine(o);
}
```

```
no automatic conversion to int: b

convert to int: 1

1

1
```

See also

Component Extensions for .NET and UWP

`event` keyword (C++/CLI and C++/CX)

5/13/2022 • 6 minutes to read • Edit Online

The `event` keyword declares an *event*, which is a notification to registered subscribers (*event handlers*) that something of interest has occurred.

All Runtimes

C++/CX supports declaring an *event member* or an *event block*. An event member is shorthand for declaring an event block. By default, an event member declares the `add`, `remove`, and `raise` functions that are declared explicitly in an event block. To customize the functions in an event member, declare an event block instead and then override the functions that you require.

Syntax

```
// event data member
modifier event delegate^ event_name;

// event block
modifier event delegate^ event_name
{
   modifier return_value add(delegate^ name);
   modifier void remove(delegate^ name);
   modifier void raise(parameters);
}
```

Parameters

modifier
A modifier that can be used on either the event declaration or an event accessor method. Possible values are `static` and `virtual`.

delegate
The delegate, whose signature the event handler must match.

event_name
The name of the event.

return_value
The return value of the event accessor method. To be verifiable, the return type must be `void`.

parameters
(optional) Parameters for the `raise` method, which match the signature of the *delegate* parameter.

Remarks

An event is an association between a delegate and an *event handler*. An event handler is a member function that responds when the event gets triggered. It allows clients from any class to register methods that match the signature and return type of the delegate.

There are two kinds of events declarations:

event data member
The compiler automatically creates storage for the event in the form of a member of the delegate type, and creates internal `add`, `remove`, and `raise` member functions. An event data member must be declared inside a

class. The return type of the return type of the delegate must match the return type of the event handler.

event block
An event block enables you to explicitly declare and customize the behavior of the `add`, `remove`, and `raise` methods.

You can use `operator +=` and `operator -=` to add and remove an event handler, or call the `add` and `remove` methods explicitly.

`event` is a context-sensitive keyword. For more information, see Context-sensitive keywords.

Windows Runtime

Remarks

For more information, see Events (C++/CX).

To add and later remove an event handler, save the `EventRegistrationToken` structure that's returned by the `add` operation. Then in the `remove` operation, use the saved `EventRegistrationToken` structure to identify the event handler to remove.

Requirements

Compiler option: `/ZW`

Common Language Runtime

The **event** keyword lets you declare an event. An event is a way for a class to provide notifications when something of interest happens.

Syntax

```
// event data member
modifier event delegate^ event_name;

// event block
modifier event delegate^ event_name
{
   modifier return_value add(delegate^ name);
   modifier void remove(delegate^ name);
   modifier void raise(parameters);
}
```

Parameters

modifier
A modifier that can be used on either the event declaration or an event accessor method. Possible values are `static` and `virtual`.

delegate
The delegate, whose signature the event handler must match.

event_name
The name of the event.

return_value
The return value of the event accessor method. To be verifiable, the return type must be `void`.

parameters
(optional) Parameters for the `raise` method, which match the signature of the *delegate* parameter.

Remarks

An event is an association between a delegate and an *event handler*. An event handler is a member function that responds when the event gets triggered. It allows clients from any class to register methods that match the signature and return type of the underlying delegate.

The delegate can have one or more associated methods. These methods get called when your code indicates that the event has occurred. An event in one program can be made available to other programs that target the .NET Framework common language runtime.

There are two kinds of event declarations:

event data members
The compiler creates storage for data member events as a member of the delegate type. An event data member must be declared inside a class. It's also known as a *trivial event*. See the code sample for an example.

event blocks
Event blocks let you customize the behavior of the `add`, `remove`, and `raise` methods, by implementing `add`, `remove`, and `raise` methods. The signature of the `add`, `remove`, and `raise` methods must match the signature of the delegate. Event block events aren't data members. Any use as a data member generates a compiler error.

The return type of the event handler must match the return type of the delegate.

In the .NET Framework, you can treat a data member as if it were a method itself (that is, the `Invoke` method of its corresponding delegate). To do so, predefine the delegate type for declaring a managed event data member. In contrast, a managed event method implicitly defines the corresponding managed delegate if it isn't already defined. See the code sample at the end of this article for an example.

When declaring a managed event, you can specify `add` and `remove` accessors that will be called when event handlers are added or removed using operators `+=` and `-=`. The `add`, `remove`, and `raise` methods can be called explicitly.

The following steps must be taken to create and use events in Microsoft C++:

1. Create or identify a delegate. If you're defining your own event, you must also ensure that there's a delegate to use with the `event` keyword. If the event is predefined, in the .NET Framework for example, then consumers of the event need only know the name of the delegate.

2. Create a class that contains:

   - An event created from the delegate.

   - (Optional) A method that verifies that an instance of the delegate declared with the `event` keyword exists. Otherwise, this logic must be placed in the code that fires the event.

   - Methods that call the event. These methods can be overrides of some base class functionality.

   This class defines the event.

3. Define one or more classes that connect methods to the event. Each of these classes will associate one or more methods with the event in the base class.

4. Use the event:

   - Create an object of the class that contains the event declaration.

   - Create an object of the class that contains the event definition.

For more information on C++/CLI events, see Events in an Interface.

Requirements

Compiler option: `/clr`

Examples

The following code example demonstrates declaring pairs of delegates, events, and event handlers. It shows how to subscribe (add), invoke, and then unsubscribe (remove) the event handlers.

```
// mcppv2_events.cpp
// compile with: /clr
using namespace System;

// declare delegates
delegate void ClickEventHandler(int, double);
delegate void DblClickEventHandler(String^);

// class that defines events
ref class EventSource {
public:
   event ClickEventHandler^ OnClick;   // declare the event OnClick
   event DblClickEventHandler^ OnDblClick;   // declare OnDblClick

   void FireEvents() {
      // raises events
      OnClick(7, 3.14159);
      OnDblClick("Hello");
   }
};

// class that defines methods that will called when event occurs
ref class EventReceiver {
public:
   void OnMyClick(int i, double d) {
      Console::WriteLine("OnClick: {0}, {1}", i, d);
   }

   void OnMyDblClick(String^ str) {
      Console::WriteLine("OnDblClick: {0}", str);
   }
};

int main() {
   EventSource ^ MyEventSource = gcnew EventSource();
   EventReceiver^ MyEventReceiver = gcnew EventReceiver();

   // hook handler to event
   MyEventSource->OnClick += gcnew ClickEventHandler(MyEventReceiver, &EventReceiver::OnMyClick);
   MyEventSource->OnDblClick += gcnew DblClickEventHandler(MyEventReceiver, &EventReceiver::OnMyDblClick);

   // invoke events
   MyEventSource->FireEvents();

   // unhook handler to event
   MyEventSource->OnClick -= gcnew ClickEventHandler(MyEventReceiver, &EventReceiver::OnMyClick);
   MyEventSource->OnDblClick -= gcnew DblClickEventHandler(MyEventReceiver, &EventReceiver::OnMyDblClick);
}
```

```
OnClick: 7, 3.14159

OnDblClick: Hello
```

The following code example demonstrates the logic used to generate the `raise` method of a trivial event. If the event has one or more subscribers, calling the `raise` method implicitly or explicitly calls the delegate. If the delegate's return type isn't `void` and if there are zero event subscribers, the `raise` method returns the default value for the delegate type. If there are no event subscribers, calling the `raise` method immediately returns and no exception is raised. If the delegate return type isn't `void`, the delegate type is returned.

```
// trivial_events.cpp
// compile with: /clr /c
using namespace System;
public delegate int Del();
public ref struct C {
   int i;
   event Del^ MyEvent;

   void FireEvent() {
      i = MyEvent();
   }
};

ref struct EventReceiver {
   int OnMyClick() { return 0; }
};

int main() {
   C c;
   c.i = 687;

   c.FireEvent();
   Console::WriteLine(c.i);
   c.i = 688;

   EventReceiver^ MyEventReceiver = gcnew EventReceiver();
   c.MyEvent += gcnew Del(MyEventReceiver, &EventReceiver::OnMyClick);
   Console::WriteLine(c.i);
}
```

```
0

688
```

See also

Component extensions for .NET and UWP

Exception Handling (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

Applications compiled with the `/ZW` compiler option or `/clr` compiler option both use *exceptions* to handle unexpected errors during program execution. The following topics discuss exception handling in either C++/CX or C++/CLI applications.

In This Section

Basic Concepts in Using Managed Exceptions
Describes throwing exceptions and using `try` / `catch` blocks.

Differences in Exception Handling Behavior Under /clr
Discusses the differences from the standard behavior of C++ exception handling.

finally
Discusses how to use the finally keyword.

How to: Define and Install a Global Exception Handler
Demonstrates how unhandled exceptions can be captured.

How to: Catch Exceptions in Native Code Thrown from MSIL
Discusses how to catch CLR and C++ exceptions in native code.

How to: Define and Install a Global Exception Handler
Demonstrates how to catch all unhandled exceptions.

Related Sections

Exception Handling
Describes exception handling in standard C++.

See also

Component Extensions for .NET and UWP

Explicit Overrides (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

This topic discusses how to explicitly override a member of a base class or interface. A named (explicit) override should only be used to override a method with a derived method that has a different name.

All Runtimes

Syntax

```
overriding-function-declarator = type::function [,type::function] { overriding-function-definition }
overriding-function-declarator = function { overriding-function-definition }
```

Parameters

overriding-function-declarator
The return type, name, and argument list of the overriding function. Note that the overriding function does not have to have the same name as the function being overridden.

type
The base type that contains a function to override.

function
A comma-delimited list of one or more function names to override.

overriding-function-definition
The function body statements that define the overriding function.

Remarks

Use explicit overrides to create an alias for a method signature, or to provide different implementations for methods with the same signature.

For information about modifying the behavior of inherited types and inherited type members, see Override Specifiers.

Windows Runtime

Requirements

Compiler option: `/ZW`

Common Language Runtime

Remarks

For information about explicit overrides in native code or code compiled with `/clr:oldSyntax`, see Explicit Overrides.

Requirements

Compiler option: `/clr`

Examples

The following code example shows a simple, implicit override and implementation of a member in a base interface, not using explicit overrides.

```
// explicit_override_1.cpp
// compile with: /clr
interface struct I1 {
   virtual void f();
};

ref class X : public I1 {
public:
   virtual void f() {
      System::Console::WriteLine("X::f override of I1::f");
   }
};

int main() {
   I1 ^ MyI = gcnew X;
   MyI -> f();
}
```

```
X::f override of I1::f
```

The following code example shows how to implement all interface members with a common signature, using explicit override syntax.

```
// explicit_override_2.cpp
// compile with: /clr
interface struct I1 {
   virtual void f();
};

interface struct I2 {
   virtual void f();
};

ref struct X : public I1, I2 {
   virtual void f() = I1::f, I2::f {
      System::Console::WriteLine("X::f override of I1::f and I2::f");
   }
};

int main() {
   I1 ^ MyI = gcnew X;
   I2 ^ MyI2 = gcnew X;
   MyI -> f();
   MyI2 -> f();
}
```

```
X::f override of I1::f and I2::f
X::f override of I1::f and I2::f
```

The following code example shows how a function override can have a different name from the function it is implementing.

```
// explicit_override_3.cpp
// compile with: /clr
interface struct I1 {
   virtual void f();
};

ref class X : public I1 {
public:
   virtual void g() = I1::f {
      System::Console::WriteLine("X::g");
   }
};

int main() {
   I1 ^ a = gcnew X;
   a->f();
}
```

```
X::g
```

The following code example shows an explicit interface implementation that implements a type safe collection.

```
// explicit_override_4.cpp
// compile with: /clr /LD
using namespace System;
ref class R : ICloneable {
   int X;

   virtual Object^ C() sealed = ICloneable::Clone {
      return this->Clone();
   }

public:
   R() : X(0) {}
   R(int x) : X(x) {}

   virtual R^ Clone() {
      R^ r = gcnew R;
      r->X = this->X;
      return r;
   }
};
```

See also

Component Extensions for .NET and UWP

ref new, gcnew (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

The **ref new** aggregate keyword allocates an instance of a type that is garbage collected when the object becomes inaccessible, and that returns a handle (^) to the allocated object.

All Runtimes

Memory for an instance of a type that is allocated by **ref new** is deallocated automatically.

A **ref new** operation throws `OutOfMemoryException` if it is unable to allocate memory.

For more information about how memory for native C++ types is allocated and deallocated, see the new and delete operators.

Windows Runtime

Use **ref new** to allocate memory for Windows Runtime objects whose lifetime you want to administer automatically. The object is automatically deallocated when its reference count goes to zero, which occurs after the last copy of the reference has gone out of scope. For more information, see Ref classes and structs.

Requirements

Compiler option: `/ZW`

Common Language Runtime

Memory for a managed type (reference or value type) is allocated by **gcnew**, and deallocated by using garbage collection.

Requirements

Compiler option: `/clr`

Examples

The following example uses **gcnew** to allocate a Message object.

```
// mcppv2_gcnew_1.cpp
// compile with: /clr
ref struct Message {
   System::String^ sender;
   System::String^ receiver;
   System::String^ data;
};

int main() {
   Message^ h_Message  = gcnew Message;
  //...
}
```

The following example uses **gcnew** to create a boxed value type for use like a reference type.

```
// example2.cpp : main project file.
// compile with /clr
using namespace System;
value class Boxed {
    public:
        int i;
};
int main()
{
    Boxed^ y = gcnew Boxed;
    y->i = 32;
    Console::WriteLine(y->i);
    return 0;
}
```

```
32
```

See also

Component Extensions for .NET and UWP

Generics (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

Generics are parameterized types and methods. In this section, find out which generic features both the Windows Runtime and the common language runtime support, and which ones only the common language runtime supports. You'll also find out how to author your own generic methods and types in C++/CLI, and how to use generic types authored in a .NET Framework language in C++/CLI. Finally, this section provides a comparison of generics and C++ templates.

In This Section

Supported by the Windows Runtime and the Common Language Runtime

Overview of Generics in C++/CLI
Information about what generics are, the motivation for the language feature, and definitions of terms that are used to describe generics. Also, information about the use of reference types and value types as type parameters for generics.

Generic Interfaces (C++/CLI)
Information about defining and using generic interfaces.

Generic Delegates (C++/CLI)
Information about defining and using generic delegates.

Constraints on Generic Type Parameters (C++/CLI)
Information about using constraints in generic types.

Consuming Generics (C++/CLI)
Information about using generics defined in .NET assemblies, possibly authored in other languages, in C++/CLI.

Generics and Templates (C++/CLI)
A comparison of generics and templates, when to use each, and how to combine them usefully.

Supported by the Common Language Runtime

Generic Functions (C++/CLI)
Information about defining and using generic functions and methods.

Generic Classes (C++/CLI)
Information about defining and using generic classes.

Related Sections

Using the for each, in keyword on a generic collection.

See also

Component Extensions for .NET and UWP

Overview of Generics in C++/CLI

5/13/2022 • 6 minutes to read • Edit Online

Generics are parameterized types supported by the common language runtime. A parameterized type is a type that is defined with an unknown type parameter that is specified when the generic is used.

Why Generics?

C++ supports templates and both templates and generics support parameterized types to create typed collection classes. However, templates provide compile-time parameterization. You cannot reference an assembly containing a template definition and create new specializations of the template. Once compiled, a specialized template looks like any other class or method. In contrast, generics are emitted in MSIL as a parameterized type known by the runtime to be a parameterized type; source code that references an assembly containing a generic type can create specializations of the generic type. For more information on the comparison of standard C++ templates and generics, see Generics and Templates (C++/CLI).

Generic Functions and Types

Class types, as long as they are managed types, may be generic. An example of this might be a `List` class. The type of an object in the list would be the type parameter. If you needed a `List` class for many different types of objects, before generics you might have used a `List` that takes `System::Object` as the item type. But that would allow any object (including objects of the wrong type) to be used in the list. Such a list would be called an untyped collection class. At best, you could check the type at runtime and throw an exception. Or, you might have used a template, which would lose its generic quality once compiled into an assembly. Consumers of your assembly could not create their own specializations of the template. Generics allow you to create typed collection classes, say `List<int>` (read as "List of int") and `List<double>` ("List of double") which would generate a compile-time error if you tried to put a type that the collection was not designed to accept into the typed collection. In addition, these types remain generic after they are compiled.

A description of the syntax of generic classes may be found in Generic Classes (C++/CLI). A new namespace, System.Collections.Generic, introduces a set of parameterized collection types including Dictionary<TKey,TValue>, List<T> and LinkedList<T>.

Both instance and static class member functions, delegates, and global functions may also be generic. Generic functions may be necessary if the function's parameters are of an unknown type, or if the function itself must work with generic types. In many cases where `System::Object` may have been used in the past as a parameter for an unknown object type, a generic type parameter may be used instead, allowing for more type-safe code. Any attempt to pass in a type that the function was not designed for would be flagged as an error at compile time. Using `System::Object` as a function parameter, the inadvertent passing of an object that the function wasn't intended to deal with would not be detected, and you would have to cast the unknown object type to a specific type in the function body, and account for the possibility of an InvalidCastException. With a generic, code attempting to pass an object to the function would cause a type conflict so the function body is guaranteed to have the correct type.

The same benefits apply to collection classes built on generics. Collection classes in the past would use `System::Object` to store elements in a collection. Insertion of objects of a type that the collection was not designed for was not flagged at compile time, and often not even when the objects were inserted. Usually, an object would be cast to some other type when it was accessed in the collection. Only when the cast failed would the unexpected type be detected. Generics solves this problem at compile time by detecting any code that inserts a type that doesn't match (or implicitly convert to) the type parameter of the generic collection.

For a description of the syntax, see Generic Functions (C++/CLI).

Terminology Used With Generics

Type Parameters

A generic declaration contains one or more unknown types known as *type parameters*. Type parameters are given a name which stands for the type within the body of the generic declaration. The type parameter is used as a type within the body of the generic declaration. The generic declaration for `List<T>` contains the type parameter T.

Type Arguments

The *type argument* is the actual type used in place of the type parameter when the generic is specialized for a specific type or types. For example, `int` is the type argument in `List<int>`. Value types and handle types are the only types allowed in as a generic type argument.

Constructed Type

A type constructed from a generic type is referred to as a *constructed type*. A type not fully specified, such as `List<T>` is an *open constructed type*; a type fully specified, such as `List<double>,` is a *closed constructed type* or *specialized type*. Open constructed types may be used in the definition of other generic types or methods and may not be fully specified until the enclosing generic is itself specified. For example, the following is a use of an open constructed type as a base class for a generic:

```
// generics_overview.cpp
// compile with: /clr /c
generic <typename T>

ref class List {};

generic <typename T>

ref class Queue : public List<T> {};
```

Constraint

A constraint is a restriction on the types that may be used as a type parameter. For example, a given generic class could accept only classes that inherit from a specified class, or implement a specified interface. For more information, see Constraints on Generic Type Parameters (C++/CLI).

Reference Types and Value Types

Handles types and value types may be used as type arguments. In the generic definition, in which either type may be used, the syntax is that of reference types. For example, the `->` operator is used to access members of the type of the type parameter whether or not the type eventually used is a reference type or a value type. When a value type is used as the type argument, the runtime generates code that uses the value types directly without boxing the value types.

When using a reference type as a generic type argument, use the handle syntax. When using a value type as a generic type argument, use the name of the type directly.

```
// generics_overview_2.cpp
// compile with: /clr
generic <typename T>

ref class GenericType {};
ref class ReferenceType {};

value struct ValueType {};

int main() {
    GenericType<ReferenceType^> x;
    GenericType<ValueType> y;
}
```

Type Parameters

Type parameters in a generic class are treated like other identifiers. However, because the type is not known, there are restrictions on their use. For example, you cannot use members and methods of the type parameter class unless the type parameter is known to support these members. That is, to access a member through the type parameter, you must add the type that contains the member to the type parameter's constraint list.

```
// generics_overview_3.cpp
// compile with: /clr
interface class I {
   void f1();
   void f2();
};

ref struct R : public I {
   virtual void f1() {}
   virtual void f2() {}
   virtual void f3() {}
};

generic <typename T>
where T : I
void f(T t) {
   t->f1();
   t->f2();
   safe_cast<R^>(t)->f3();
}

int main() {
   f(gcnew R());
}
```

These restrictions apply to operators as well. An unconstrained generic type parameter may not use the `==` and `!=` operators to compare two instances of the type parameter, in case the type does not support these operators. These checks are necessary for generics, but not for templates, because generics may be specialized at runtime with any class that satisfies the constraints, when it is too late to check for the use of invalid members.

A default instance of the type parameter may be created by using the `()` operator. For example:

`T t = T();`

where `T` is a type parameter in a generic class or method definition, initializes the variable to its default value. If `T` is a ref class it will be a null pointer; if `T` is a value class, the object is initialized to zero. This is called a *default initializer*.

See also

Generics

Generic functions (C++/CLI)

5/13/2022 • 3 minutes to read • Edit Online

A generic function is a function that is declared with type parameters. When called, actual types are used instead of the type parameters.

All platforms

Remarks

This feature doesn't apply to all platforms.

Windows Runtime

Remarks

This feature isn't supported in the Windows Runtime.

Requirements

Compiler option: `/ZW`

Common Language Runtime

A generic function is a function that's declared with type parameters. When called, actual types are used instead of the type parameters.

Syntax

```
[attributes] [modifiers]
return-type identifier<type-parameter-identifier-list>
[type-parameter-constraints-clauses]

([formal-parameters])
{function-body}
```

Parameters

`attributes`
(Optional) Additional declarative information. For more information on attributes and attribute classes, see attributes.

`modifiers`
(Optional) A modifier for the function, such as `static`. `virtual` isn't allowed since virtual methods may not be generic.

`return-type`
The type returned by the method. If the return type is void, no return value is required.

`identifier`
The function name.

`type-parameter-identifier-list`
Comma-separated identifiers list.

`formal-parameters`

(Optional) Parameter list.

`type-parameter-constraints-clauses`
This set specifies restrictions on the types that may be used as type arguments, and takes the form specified in Constraints on Generic Type Parameters (C++/CLI).

`function-body`
The body of the method, which may refer to the type parameter identifiers.

Remarks

Generic functions are functions declared with a generic type parameter. They may be methods in a `class` or `struct`, or standalone functions. A single generic declaration implicitly declares a family of functions that differ only in the substitution of a different actual type for the generic type parameter.

A `class` or `struct` constructor may not be declared with generic type parameters.

When called, the generic type parameter is replaced by an actual type. The actual type may be explicitly specified in angled brackets using syntax similar to a template function call. If called without the type parameters, the compiler will attempt to deduce the actual type from the parameters supplied in the function call. The compiler reports an error if the intended type argument cannot be deduced from the parameters used.

Requirements

Compiler option: `/clr`

Examples

The following code sample demonstrates a generic function.

```
// generics_generic_function_1.cpp
// compile with: /clr
generic <typename ItemType>
void G(int i) {}

ref struct A {
   generic <typename ItemType>
   void G(ItemType) {}

   generic <typename ItemType>
   static void H(int i) {}
};

int main() {
   A myObject;

   // generic function call
   myObject.G<int>(10);

   // generic function call with type parameters deduced
   myObject.G(10);

   // static generic function call
   A::H<int>(10);

   // global generic function call
   G<int>(10);
}
```

Generic functions can be overloaded based on signature or *arity*, the number of type parameters on a function. Also, generic functions can be overloaded with non-generic functions of the same name, as long as the functions differ in some type parameters. For example, the following functions can be overloaded:

```
// generics_generic_function_2.cpp
// compile with: /clr /c
ref struct MyClass {
   void MyMythod(int i) {}

   generic <class T>
   void MyMythod(int i) {}

   generic <class T, class V>
   void MyMythod(int i) {}
};
```

The following example uses a generic function to find the first element in an array. It declares `MyClass`, which inherits from the base class `MyBaseClass`. `MyClass` contains a generic function, `MyFunction`, which calls another generic function, `MyBaseClassFunction`, within the base class. In `main`, the generic function, `MyFunction`, is called using different type arguments.

```
// generics_generic_function_3.cpp
// compile with: /clr
using namespace System;

ref class MyBaseClass {
protected:
   generic <class ItemType>
   ItemType MyBaseClassFunction(ItemType item) {
      return item;
   }
};

ref class MyClass: public MyBaseClass {
public:
   generic <class ItemType>
   ItemType MyFunction(ItemType item) {
      return MyBaseClass::MyBaseClassFunction<ItemType>(item);
   }
};

int main() {
   MyClass^ myObj = gcnew MyClass();

   // Call MyFunction using an int.
   Console::WriteLine("My function returned an int: {0}",
                        myObj->MyFunction<int>(2003));

   // Call MyFunction using a string.
   Console::WriteLine("My function returned a string: {0}",
   myObj->MyFunction<String^>("Hello generic functions!"));
}
```

```
My function returned an int: 2003
My function returned a string: Hello generic functions!
```

See also

Component Extensions for .NET and UWP
Generics

Generic Classes (C++/CLI)

5/13/2022 • 13 minutes to read • Edit Online

A generic class is declared using the following form:

Syntax

```
[attributes]
generic <class-key type-parameter-identifier(s)>
[constraint-clauses]
[accessibility-modifiers] ref class identifier  [modifiers]
[: base-list]
{
class-body
} [declarators] [;]
```

Remarks

In the above syntax, the following terms are used:

attributes
(Optional) Additional declarative information. For more information on attributes and attribute classes, see Attributes.

class-key
Either `class` or `typename`

type-parameter-identifier(s), Comma-separated list of identifiers specifying the names of the type parameters.

constraint-clauses
A list (not comma-separated) of **where** clauses specifying the constraints for the type parameters. Takes the form:

> **where** *type-parameter-identifier* **:** *constraint-list* **...**

constraint-list
class-or-interface[`,` ...]

accessibility-modifiers
Accessibility modifiers for the generic class. For the Windows Runtime, the only allowed modifier is `private`. For the common language runtime, the allowed modifiers are `private` and `public`.

identifier
The name of the generic class, any valid C++ identifier.

modifiers
(Optional) Allowed modifiers include **sealed** and **abstract**.

base-list
A list that contains the one base class and any implemented interfaces, all separated by commas.

class-body
The body of the class, containing fields, member functions, etc.

declarators
Declarations of any variables of this type. For example: `^` *identifier*[`,` ...]

You can declare generic classes such as these (note that the keyword `class` may be used instead of `typename`). In this example, `ItemType`, `KeyType` and `ValueType` are unknown types that are specified at the point where the type. `HashTable<int, int>` is a constructed type of the generic type `HashTable<KeyType, ValueType>`. A number of different constructed types can be constructed from a single generic type. Constructed types constructed from generic classes are treated like any other ref class type.

```
// generic_classes_1.cpp
// compile with: /clr
using namespace System;
generic <typename ItemType>
ref struct Stack {
   // ItemType may be used as a type here
   void Add(ItemType item) {}
};

generic <typename KeyType, typename ValueType>
ref class HashTable {};

// The keyword class may be used instead of typename:
generic <class ListItem>
ref class List {};

int main() {
   HashTable<int, Decimal>^ g1 = gcnew HashTable<int, Decimal>();
}
```

Both value types (either built-in types such as `int` or `double`, or user-defined value types) and reference types may be used as a generic type argument. The syntax within the generic definition is the same regardless. Syntactically, the unknown type is treated as if it were a reference type. However, the runtime is able to determine that if the type actually used is a value type and substitute the appropriate generated code for direct access to members. Value types used as generic type arguments are not boxed and so do not suffer the performance penalty associated with boxing. The syntax used within the body of the generic should be `T^` and `->` instead of `.`. Any use of ref new, gcnew for the type parameter will be appropriately interpreted by the runtime as the simple creation of a value type if the type argument is a value type.

You can also declare a generic class with Constraints on Generic Type Parameters (C++/CLI) on the types that can be used for the type parameter. In the following example any type used for `ItemType` must implement the `IItem` interface. Attempting to use `int`, for example, which does not implement `IItem`, would produce a compile-time error because the type argument does not satisfy the constraint.

```
// generic_classes_2.cpp
// compile with: /clr /c
interface class IItem {};
generic <class ItemType>
where ItemType : IItem
ref class Stack {};
```

Generic classes in the same namespace cannot be overloaded by only changing the number or the types of type parameters. However, if each class lives in a different namespace, they can be overloaded. For example, consider the following two classes, `MyClass` and `MyClass<ItemType>`, in the namespaces `A` and `B`. The two classes can then be overloaded in a third namespace C:

```
// generic_classes_3.cpp
// compile with: /clr /c
namespace A {
   ref class MyClass {};
}

namespace B {
   generic <typename ItemType>
   ref class MyClass2 { };
}

namespace C {
   using namespace A;
   using namespace B;

   ref class Test {
      static void F() {
         MyClass^ m1 = gcnew MyClass();   // OK
         MyClass2<int>^ m2 = gcnew MyClass2<int>();   // OK
      }
   };
}
```

The base class and base interfaces cannot be type parameters. However, the base class can involve the type parameter as an argument, as in the following case:

```
// generic_classes_4.cpp
// compile with: /clr /c
generic <typename ItemType>
interface class IInterface {};

generic <typename ItemType>
ref class MyClass : IInterface<ItemType> {};
```

Constructors and destructors are executed once for each object instance (as usual); static constructors are executed once for each constructed type.

Fields in Generic Classes

This section demonstrates the use of instance and static fields in generic classes.

Instance Variables

Instance variables of a generic class can have types and variable initializers that include any type parameters from the enclosing class.

Example: Different generic classes

In the following example, three different instances of the generic class, MyClass<ItemType>, are created by using the appropriate type arguments (`int`, `double`, and **string**).

```
// generics_instance_fields1.cpp
// compile with: /clr
// Instance fields on generic classes
using namespace System;

generic <typename ItemType>
ref class MyClass {
// Field of the type ItemType:
public :
   ItemType field1;
   // Constructor using a parameter of the type ItemType:
   MyClass(ItemType p) {
   field1 = p;
   }
};

int main() {
   // Instantiate an instance with an integer field:
   MyClass<int>^ myObj1 = gcnew MyClass<int>(123);
   Console::WriteLine("Integer field = {0}", myObj1->field1);

   // Instantiate an instance with a double field:
   MyClass<double>^ myObj2 = gcnew MyClass<double>(1.23);
   Console::WriteLine("Double field = {0}", myObj2->field1);

   // Instantiate an instance with a String field:
   MyClass<String^>^ myObj3 = gcnew MyClass<String^>("ABC");
   Console::WriteLine("String field = {0}", myObj3->field1);
   }
```

```
Integer field = 123
Double field = 1.23
String field = ABC
```

Static Variables

On the creation of a new generic type, new instances of any static variables are created and any static constructor for that type is executed.

Static variables can use any type parameters from the enclosing class.

Example: Use static variables

The following example demonstrates using static fields and a static constructor within a generic class.

```
// generics_static2.cpp
// compile with: /clr
using namespace System;

interface class ILog {
   void Write(String^ s);
};

ref class DateTimeLog : ILog {
public:
   virtual void Write(String^ s) {
      Console::WriteLine( "{0}\t{1}", DateTime::Now, s);
   }
};

ref class PlainLog : ILog {
public:
   virtual void Write(String^ s) { Console::WriteLine(s); }
};

generic <typename LogType>
where LogType : ILog
ref class G {
   static LogType s_log;

public:
   G(){}
   void SetLog(LogType log) { s_log = log; }
   void F() { s_log->Write("Test1"); }
   static G() { Console::WriteLine("Static constructor called."); }
};

int main() {
   G<PlainLog^>^ g1 = gcnew G<PlainLog^>();
   g1->SetLog(gcnew PlainLog());
   g1->F();

   G<DateTimeLog^>^ g2 = gcnew G<DateTimeLog^>();
   g2->SetLog(gcnew DateTimeLog());

   // prints date
   // g2->F();
}
```

```
Static constructor called.
Static constructor called.
Static constructor called.
Test1
```

Methods in Generic Classes

Methods in generic classes can be generic themselves; non-generic methods will be implicitly parameterized by the class type parameter.

The following special rules apply to methods within generic classes:

- Methods in generic classes can use type parameters as parameters, return types, or local variables.
- Methods in generic classes can use open or closed constructed types as parameters, return types, or local variables.

Non-Generic Methods in Generic Classes

Methods in generic classes that have no additional type parameters are usually referred to as non-generic although they are implicitly parameterized by the enclosing generic class.

The signature of a non-generic method can include one or more type parameters of the enclosing class, either directly or in an open constructed type. For example:

`void MyMethod(MyClass<ItemType> x) {}`

The body of such methods can also use these type parameters.

Example: Declare non-generic method

The following example declares a non-generic method, `ProtectData`, inside a generic class, `MyClass<ItemType>`. The method uses the class type parameter `ItemType` in its signature in an open constructed type.

```
// generics_non_generic_methods1.cpp
// compile with: /clr
// Non-generic methods within a generic class.
using namespace System;

generic <typename ItemType>
ref class MyClass {
public:
   String^ name;
   ItemType data;

   MyClass(ItemType x) {
      data = x;
   }

   // Non-generic method using the type parameter:
   virtual void ProtectData(MyClass<ItemType>^ x) {
      data = x->data;
   }
};

// ItemType defined as String^
ref class MyMainClass: MyClass<String^> {
public:
   // Passing "123.00" to the constructor:
   MyMainClass(): MyClass<String^>("123.00") {
      name = "Jeff Smith";
   }

   virtual void ProtectData(MyClass<String^>^ x) override {
      x->data = String::Format("${0}**", x->data);
   }

   static void Main() {
      MyMainClass^ x1 = gcnew MyMainClass();

      x1->ProtectData(x1);
      Console::WriteLine("Name: {0}", x1->name);
      Console::WriteLine("Amount: {0}", x1->data);
   }
};

int main() {
   MyMainClass::Main();
}
```

```
Name: Jeff Smith
Amount: $123.00**
```

Generic Methods in Generic Classes

You can declare generic methods in both generic and non-generic classes. For example:

Example: Declare generic and non-generic methods

```
// generics_method2.cpp
// compile with: /clr /c
generic <typename Type1>
ref class G {
public:
   // Generic method having a type parameter
   // from the class, Type1, and its own type
   // parameter, Type2
   generic <typename Type2>
   void Method1(Type1 t1, Type2 t2) { F(t1, t2); }

   // Non-generic method:
   // Can use the class type param, Type1, but not Type2.
   void Method2(Type1 t1) { F(t1, t1); }

   void F(Object^ o1, Object^ o2) {}
};
```

The non-generic method is still generic in the sense that it is parameterized by the class's type parameter, but it has no additional type parameters.

All types of methods in generic classes can be generic, including static, instance, and virtual methods.

Example: Declare and use generic methods

The following example demonstrates declaring and using generic methods within generic classes:

```
// generics_generic_method2.cpp
// compile with: /clr
using namespace System;
generic <class ItemType>
ref class MyClass {
public:
   // Declare a generic method member.
   generic <class Type1>
   String^ MyMethod(ItemType item, Type1 t) {
      return String::Concat(item->ToString(), t->ToString());
   }
};

int main() {
   // Create instances using different types.
   MyClass<int>^ myObj1 = gcnew MyClass<int>();
   MyClass<String^>^ myObj2 = gcnew MyClass<String^>();
   MyClass<String^>^ myObj3 = gcnew MyClass<String^>();

   // Calling MyMethod using two integers.
   Console::WriteLine("MyMethod returned: {0}",
            myObj1->MyMethod<int>(1, 2));

   // Calling MyMethod using an integer and a string.
   Console::WriteLine("MyMethod returned: {0}",
            myObj2->MyMethod<int>("Hello #", 1));

   // Calling MyMethod using two strings.
   Console::WriteLine("MyMethod returned: {0}",
       myObj3->MyMethod<String^>("Hello ", "World!"));

   // generic methods can be called without specifying type arguments
   myObj1->MyMethod<int>(1, 2);
   myObj2->MyMethod<int>("Hello #", 1);
   myObj3->MyMethod<String^>("Hello ", "World!");
}
```

```
MyMethod returned: 12
MyMethod returned: Hello #1
MyMethod returned: Hello World!
```

Using Nested Types in Generic Classes

Just as with ordinary classes, you can declare other types inside a generic class. The nested class declaration is implicitly parameterized by the type parameters of the outer class declaration. Thus, a distinct nested class is defined for each constructed outer type. For example, in the declaration,

```
// generic_classes_5.cpp
// compile with: /clr /c
generic <typename ItemType>
ref struct Outer {
   ref class Inner {};
};
```

The type `Outer<int>::Inner` is not the same as the type `Outer<double>::Inner`.

As with generic methods in generic classes, additional type parameters can be defined for the nested type. If you use the same type parameter names in the inner and outer class, the inner type parameter will hide the outer type parameter.

```
// generic_classes_6.cpp
// compile with: /clr /c
generic <typename ItemType>
ref class Outer {
   ItemType outer_item;   // refers to outer ItemType

   generic <typename ItemType>
   ref class Inner {
      ItemType inner_item;   // refers to Inner ItemType
   };
};
```

Since there is no way to refer to the outer type parameter, the compiler will produce a warning in this situation.

When constructed nested generic types are named, the type parameter for the outer type is not included in the type parameter list for the inner type, even though the inner type is implicitly parameterized by the outer type's type parameter. In the above case, a name of a constructed type would be `Outer<int>::Inner<string>`.

Example: Build and read linked list

The following example demonstrates building and reading a linked list using nested types in generic classes.

```
// generics_linked_list.cpp
// compile with: /clr
using namespace System;
generic <class ItemType>
ref class LinkedList {
// The node class:
public:
   ref class Node {
   // The link field:
   public:
      Node^ next;
      // The data field:
      ItemType item;
   } ^first, ^current;
};

ref class ListBuilder {
public:
   void BuildIt(LinkedList<double>^ list) {
      /* Build the list */
      double m[5] = {0.1, 0.2, 0.3, 0.4, 0.5};
      Console::WriteLine("Building the list:");

      for (int n=0; n<=4; n++) {
         // Create a new node:
         list->current = gcnew LinkedList<double>::Node();

         // Assign a value to the data field:
         list->current->item = m[n];

         // Set the link field "next" to be the same as
         // the "first" field:
         list->current->next = list->first;

         // Redirect "first" to the new node:
         list->first = list->current;

         // Display node's data as it builds:
         Console::WriteLine(list->current->item);
      }
   }
   void ReadIt(LinkedList<double>^ list) {
      // Read the list
      // Make "first" the "current" link field:
      list->current = list->first;
      Console::WriteLine("Reading nodes:");

      // Read nodes until current == null:
      while (list->current != nullptr) {
         // Display the node's data field:
         Console::WriteLine(list->current->item);

         // Move to the next node:
         list->current = list->current->next;
      }
   }
};

int main() {
   // Create a list:
   LinkedList<double>^ aList = gcnew LinkedList<double>();

   // Initialize first node:
   aList->first = nullptr;

   // Instantiate the class, build, and read the list:
   ListBuilder^ myListBuilder = gcnew ListBuilder();
   myListBuilder->BuildIt(aList);
   myListBuilder->ReadIt(aList);
}
```

```
Building the list:
0.1
0.2
0.3
0.4
0.5
Reading nodes:
0.5
0.4
0.3
0.2
0.1
```

Properties, Events, Indexers and Operators in Generic Classes

- Properties, events, indexers and operators can use the type parameters of the enclosing generic class as return values, parameters, or local variables, such as when `ItemType` is a type parameter of a class:

  ```
  public ItemType MyProperty {}
  ```

- Properties, events, indexers and operators cannot themselves be parameterized.

Example: Declare instance property

This example shows declarations of an instance property within a generic class.

```
// generics_generic_properties1.cpp
// compile with: /clr
using namespace System;

generic <typename ItemType>
ref class MyClass {
private:
   property ItemType myField;

public:
   property ItemType MyProperty {
      ItemType get() {
         return myField;
      }
      void set(ItemType value) {
         myField = value;
      }
   }
};

int main() {
   MyClass<String^>^ c = gcnew MyClass<String^>();
   MyClass<int>^ c1 = gcnew MyClass<int>();

   c->MyProperty = "John";
   c1->MyProperty = 234;

   Console::Write("{0}, {1}", c->MyProperty, c1->MyProperty);
}
```

```
John, 234
```

Example: Generic class with event

The next example shows a generic class with an event.

```
// generics_generic_with_event.cpp
// compile with: /clr
// Declare a generic class with an event and
// invoke events.
using namespace System;

// declare delegates
generic <typename ItemType>
delegate void ClickEventHandler(ItemType);

// generic class that defines events
generic <typename ItemType>
ref class EventSource {
public:
   // declare the event OnClick
   event ClickEventHandler<ItemType>^ OnClick;
   void FireEvents(ItemType item) {
      // raises events
      OnClick(item);
   }
};

// generic class that defines methods that will called when
// event occurs
generic <typename ItemType>
ref class EventReceiver {
public:
   void OnMyClick(ItemType item) {
   Console::WriteLine("OnClick: {0}", item);
   }
};

int main() {
   EventSource<String^>^ MyEventSourceString =
                   gcnew EventSource<String^>();
   EventSource<int>^ MyEventSourceInt = gcnew EventSource<int>();
   EventReceiver<String^>^ MyEventReceiverString =
                   gcnew EventReceiver<String^>();
   EventReceiver<int>^ MyEventReceiverInt = gcnew EventReceiver<int>();

   // hook handler to event
   MyEventSourceString->OnClick += gcnew ClickEventHandler<String^>(
       MyEventReceiverString, &EventReceiver<String^>::OnMyClick);
   MyEventSourceInt->OnClick += gcnew ClickEventHandler<int>(
             MyEventReceiverInt, &EventReceiver<int>::OnMyClick);

   // invoke events
   MyEventSourceString->FireEvents("Hello");
   MyEventSourceInt->FireEvents(112);

   // unhook handler to event
   MyEventSourceString->OnClick -= gcnew ClickEventHandler<String^>(
        MyEventReceiverString, &EventReceiver<String^>::OnMyClick);
   MyEventSourceInt->OnClick -= gcnew ClickEventHandler<int>(
        MyEventReceiverInt, &EventReceiver<int>::OnMyClick);
}
```

Generic Structs

The rules for declaring and using generic structs are the same as those for generic classes, except for the differences noted in the Visual C++ language reference.

Example: Declare generic struct

The following example declares a generic struct, `MyGenStruct`, with one field, `myField`, and assigns values of different types (`int`, `double`, `String^`) to this field.

```
// generics_generic_struct1.cpp
// compile with: /clr
using namespace System;

generic <typename ItemType>
ref struct MyGenStruct {
public:
   ItemType myField;

   ItemType AssignValue(ItemType item) {
      myField = item;
      return myField;
   }
};

int main() {
   int myInt = 123;
   MyGenStruct<int>^ myIntObj = gcnew MyGenStruct<int>();
   myIntObj->AssignValue(myInt);
   Console::WriteLine("The field is assigned the integer value: {0}",
            myIntObj->myField);

   double myDouble = 0.123;
   MyGenStruct<double>^ myDoubleObj = gcnew MyGenStruct<double>();
   myDoubleObj->AssignValue(myDouble);
   Console::WriteLine("The field is assigned the double value: {0}",
            myDoubleObj->myField);

   String^ myString = "Hello Generics!";
   MyGenStruct<String^>^ myStringObj = gcnew MyGenStruct<String^>();
   myStringObj->AssignValue(myString);
   Console::WriteLine("The field is assigned the string: {0}",
            myStringObj->myField);
}
```

```
The field is assigned the integer value: 123
The field is assigned the double value: 0.123
The field is assigned the string: Hello Generics!
```

See also

Generics

Generic Interfaces (C++/CLI)

5/13/2022 • 4 minutes to read • Edit Online

The restrictions that apply to type parameters on classes are the same as those that apply to type parameters on interfaces (see Generic Classes (C++/CLI)).

The rules that control function overloading are the same for functions within generic classes or generic interfaces.

Explicit interface member implementations work with constructed interface types in the same way as with simple interface types (see the following examples).

For more information on interfaces, see interface class.

Syntax

```
[attributes] generic <class-key type-parameter-identifier[, ...]>
[type-parameter-constraints-clauses][accesibility-modifiers] interface class identifier [: base-list] {
interface-body} [declarators] ;
```

Remarks

attributes
(Optional) Additional declarative information. For more information on attributes and attribute classes, see **Attributes**.

class-key
`class` or `typename`

type-parameter-identifier(s)
Comma-separated identifiers list.

type-parameter-constraints-clauses
Takes the form specified in Constraints on Generic Type Parameters (C++/CLI)

accessibility-modifiers
(Optional) Accessibility modifiers (e.g. **public, private**).

identifier
The interface name.

base-list
(Optional) A list that contains one or more explicit base interfaces separated by commas.

interface-body
Declarations of the interface members.

declarators
(Optional) Declarations of variables based on this type.

Example: How to declare and instantiate a generic interface

The following example demonstrates how to declare and instantiate a generic interface. In the example, the generic interface `IList<ItemType>` is declared. It is then implemented by two generic classes, `List1<ItemType>` and `List2<ItemType>`, with different implementations.

```
// generic_interface.cpp
// compile with: /clr
using namespace System;

// An exception to be thrown by the List when
// attempting to access elements beyond the
// end of the list.
ref class ElementNotFoundException : Exception {};

// A generic List interface
generic <typename ItemType>
public interface class IList {
   ItemType MoveFirst();
   bool Add(ItemType item);
   bool AtEnd();
   ItemType Current();
   void MoveNext();
};

// A linked list implementation of IList
generic <typename ItemType>
public ref class List1 : public IList<ItemType> {
   ref class Node {
      ItemType m_item;

   public:
      ItemType get_Item() { return m_item; };
      void set_Item(ItemType value) { m_item = value; };

      Node^ next;

      Node(ItemType item) {
         m_item = item;
         next = nullptr;
      }
   };

   Node^ first;
   Node^ last;
   Node^ current;

   public:
   List1() {
      first = nullptr;
      last = first;
      current = first;
   }

   virtual ItemType MoveFirst() {
      current = first;
      if (first != nullptr)
        return first->get_Item();
      else
         return ItemType();
   }

   virtual bool Add(ItemType item) {
      if (last != nullptr) {
         last->next = gcnew Node(item);
         last = last->next;
      }
      else {
         first = gcnew Node(item);
         last = first;
         current = first;
         current = first;
      }
      return true;
   }

   virtual bool AtEnd() {
      if (current == nullptr )
        return true;
      else
        return false;
   }

   virtual ItemType Current() {
       if (current != nullptr)
         return current->get_Item();
       else
         throw gcnew ElementNotFoundException();
   }

   virtual void MoveNext() {
      if (current != nullptr)
       current = current->next;
      else
        throw gcnew ElementNotFoundException();
   }
};

// An array implementation of IList
generic <typename ItemType>
ref class List2 : public IList<ItemType> {
   array<ItemType>^ item_array;
   int count;
   int current;

   public:

   List2() {
      // not yet possible to declare an
      // array of a generic type parameter
      item_array = gcnew array<ItemType>(256);
      count = current = 0;
   }

   virtual ItemType MoveFirst() {
      current = 0;
      return item_array[0];
   }

   virtual bool Add(ItemType item) {
      if (count < 256)
         item_array[count++] = item;
      else
        return false;
      return true;
   }

   virtual bool AtEnd() {
      if (current >= count)
        return true;
      else
        return false;
   }

   virtual ItemType Current() {
      if (current < count)
        return item_array[current];
      else
        throw gcnew ElementNotFoundException();
   }
   virtual void MoveNext() {
      if (current < count)
         ++current;
      else
         throw gcnew ElementNotFoundException();
   }
};

// Add elements to the list and display them.
generic <typename ItemType>
void AddStringsAndDisplay(IList<ItemType>^ list, ItemType item1, ItemType item2) {
   list->Add(item1);
   list->Add(item2);
   for (list->MoveFirst(); ! list->AtEnd(); list->MoveNext())
   Console::WriteLine(list->Current());
}

int main() {
   // Instantiate both types of list.

   List1<String^>^ list1 = gcnew List1<String^>();
   List2<String^>^ list2 = gcnew List2<String^>();

   // Use the linked list implementation of IList.
   AddStringsAndDisplay<String^>(list1, "Linked List", "List1");

   // Use the array implementation of the IList.
   AddStringsAndDisplay<String^>(list2, "Array List", "List2");
}
```

```
Linked List
List1
Array List
List2
```

Example: Declare a generic interface

This example declares a generic interface, `IMyGenIface`, and two non-generic interfaces, `IMySpecializedInt` and `ImySpecializedString`, that specialize `IMyGenIface`. The two specialized interfaces are then implemented by two classes, `MyIntClass` and `MyStringClass`. The example shows how to specialize generic interfaces, instantiate generic and non-generic interfaces, and call the explicitly implemented members on the interfaces.

```cpp
// generic_interface2.cpp
// compile with: /clr
// Specializing and implementing generic interfaces.
using namespace System;

generic <class ItemType>
public interface class IMyGenIface {
   void Initialize(ItemType f);
};

public interface class IMySpecializedInt: public IMyGenIface<int> {
   void Display();
};

public interface class IMySpecializedString: public IMyGenIface<String^> {
   void Display();
};

public ref class MyIntClass: public IMySpecializedInt {
   int myField;

public:
   virtual void Initialize(int f) {
      myField = f;
   }

   virtual void Display() {
      Console::WriteLine("The integer field contains: {0}", myField);
   }
};

public ref struct MyStringClass: IMySpecializedString {
   String^ myField;

public:
   virtual void Initialize(String^ f) {
      myField = f;
    }

   virtual void Display() {
      Console::WriteLine("The String field contains: {0}", myField);
   }
};

int main() {
   // Instantiate the generic interface.
   IMyGenIface<int>^ myIntObj = gcnew MyIntClass();

   // Instantiate the specialized interface "IMySpecializedInt."
   IMySpecializedInt^ mySpIntObj = (IMySpecializedInt^) myIntObj;

   // Instantiate the generic interface.
   IMyGenIface<String^>^ myStringObj = gcnew MyStringClass();

   // Instantiate the specialized interface "IMySpecializedString."
   IMySpecializedString^ mySpStringObj =
            (IMySpecializedString^) myStringObj;

   // Call the explicitly implemented interface members.
   myIntObj->Initialize(1234);
   mySpIntObj->Display();

   myStringObj->Initialize("My string");
   mySpStringObj->Display();
}
```

```
The integer field contains: 1234
The String field contains: My string
```

See also

Generics

Generic Delegates (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

You can use generic type parameters with delegates. For more information on delegates, see delegate (C++/CLI and C++/CX).

Syntax

```
[attributes]
generic < [class | typename] type-parameter-identifiers>
[type-parameter-constraints-clauses]
[accessibility-modifiers] delegate result-type identifier
([formal-parameters]);
```

Parameters

attributes
(Optional) Additional declarative information. For more information on attributes and attribute classes, see Attributes.

type-parameter-identifier(s)
Comma-separated list of identifiers for the type parameters.

type-parameter-constraints-clauses
Takes the form specified in Constraints on Generic Type Parameters (C++/CLI)

accessibility-modifiers
(Optional) Accessibility modifiers (e.g. `public`, `private`).

result-type
The return type of the delegate.

identifier
The name of the delegate.

formal-parameters
(Optional) The parameter list of the delegate.

Examples

The delegate type parameters are specified at the point where a delegate object is created. Both the delegate and method associated with it must have the same signature. The following is an example of a generic delegate declaration.

```
// generics_generic_delegate1.cpp
// compile with: /clr /c
generic <class ItemType>
delegate ItemType GenDelegate(ItemType p1, ItemType% p2);
```

The following sample shows that

- You cannot use the same delegate object with different constructed types. Create different delegate objects for different types.

- A generic delegate can be associated with a generic method.
- When a generic method is called without specifying type arguments, the compiler tries to infer the type arguments for the call.

```
// generics_generic_delegate2.cpp
// compile with: /clr
generic <class ItemType>
delegate ItemType GenDelegate(ItemType p1, ItemType% p2);

generic <class ItemType>
ref struct MyGenClass {
   ItemType MyMethod(ItemType i, ItemType % j) {
      return ItemType();
   }
};

ref struct MyClass {
   generic <class ItemType>
   static ItemType MyStaticMethod(ItemType i, ItemType % j) {
      return ItemType();
   }
};

int main() {
   MyGenClass<int> ^ myObj1 = gcnew MyGenClass<int>();
   MyGenClass<double> ^ myObj2 = gcnew MyGenClass<double>();
   GenDelegate<int>^ myDelegate1 =
      gcnew GenDelegate<int>(myObj1, &MyGenClass<int>::MyMethod);

   GenDelegate<double>^ myDelegate2 =
      gcnew GenDelegate<double>(myObj2, &MyGenClass<double>::MyMethod);

   GenDelegate<int>^ myDelegate =
      gcnew GenDelegate<int>(&MyClass::MyStaticMethod<int>);
}
```

The following example declares a generic delegate `GenDelegate<ItemType>`, and then instantiates it by associating it to the method `MyMethod` that uses the type parameter `ItemType`. Two instances of the delegate (an integer and a double) are created and invoked.

```
// generics_generic_delegate.cpp
// compile with: /clr
using namespace System;

// declare generic delegate
generic <typename ItemType>
delegate ItemType GenDelegate (ItemType p1, ItemType% p2);

// Declare a generic class:
generic <typename ItemType>
ref class MyGenClass {
public:
   ItemType MyMethod(ItemType p1, ItemType% p2) {
      p2 = p1;
      return p1;
    }
};

int main() {
   int i = 0, j = 0;
   double m = 0.0, n = 0.0;

   MyGenClass<int>^ myObj1 = gcnew MyGenClass<int>();
   MyGenClass<double>^ myObj2 = gcnew MyGenClass<double>();

   // Instantiate a delegate using int.
   GenDelegate<int>^ MyDelegate1 =
      gcnew GenDelegate<int>(myObj1, &MyGenClass<int>::MyMethod);

   // Invoke the integer delegate using MyMethod.
   i = MyDelegate1(123, j);

   Console::WriteLine(
      "Invoking the integer delegate: i = {0}, j = {1}", i, j);

   // Instantiate a delegate using double.
   GenDelegate<double>^ MyDelegate2 =
      gcnew GenDelegate<double>(myObj2, &MyGenClass<double>::MyMethod);

   // Invoke the integer delegate using MyMethod.
   m = MyDelegate2(0.123, n);

   Console::WriteLine(
      "Invoking the double delegate: m = {0}, n = {1}", m, n);
}
```

```
Invoking the integer delegate: i = 123, j = 123
Invoking the double delegate: m = 0.123, n = 0.123
```

See also

Generics

Constraints on generic type parameters (C++/CLI)

5/13/2022 • 5 minutes to read • Edit Online

In generic type or method declarations, you can qualify a type parameter with *constraints*. A constraint is a requirement that types used as type arguments must satisfy. For example, a constraint might be that the type argument must implement a certain interface or inherit from a specific class.

Constraints are optional; not specifying a constraint on a parameter is equivalent to constraining that parameter to Object.

Syntax

```
where type-parameter: constraint-list
```

Parameters

`type-parameter`
One of the type parameters, to be constrained.

`constraint-list`
`constraint-list` is a comma-separated list of constraint specifications. The list can include interfaces to be implemented by the type parameter.

The list can also include a class. For the type argument to satisfy a base class constraint, it must be the same class as the constraint or derive from the constraint.

You can also specify `gcnew()` to indicate the type argument must have a public parameterless constructor; or `ref class` to indicate the type argument must be a reference type, including any class, interface, delegate, or array type; or `value class` to indicate the type argument must be a value type. Any value type except `Nullable<T>` can be specified.

You can also specify a generic parameter as a constraint. The type argument supplied for the type you're constraining must be or derive from the type of the constraint. This parameter is called a *naked type constraint*.

Remarks

The constraint clause consists of `where` followed by a type parameter, a colon (`:`), and the constraint, which specifies the nature of the restriction on the type parameter. `where` is a context-sensitive keyword. For more information, see Context-sensitive keywords. Separate multiple `where` clauses with a space.

Constraints are applied to type parameters to place limitations on the types that can be used as arguments for a generic type or method.

Class and interface constraints specify that the argument types must be or inherit from a specified class or implement a specified interface.

The application of constraints to a generic type or method allows code in that type or method to take advantage of the known features of the constrained types. For example, you can declare a generic class such that the type parameter implements the `IComparable<T>` interface:

```
// generics_constraints_1.cpp
// compile with: /c /clr
using namespace System;
generic <typename T>
where T : IComparable<T>
ref class List {};
```

This constraint requires that a type argument used for `T` implements `IComparable<T>` at compile time. It also allows interface methods, such as `CompareTo`, to be called. No cast is needed on an instance of the type parameter to call interface methods.

Static methods in the type argument's class can't be called through the type parameter; they can be called only through the actual named type.

A constraint can't be a value type, including built-in types such as `int` or `double`. Since value types cannot have derived classes, only one class could ever satisfy the constraint. In that case, the generic can be rewritten with the type parameter replaced by the specific value type.

Constraints are required in some cases since the compiler won't allow the use of methods or other features of an unknown type unless the constraints imply that the unknown type supports the methods or interfaces.

Multiple constraints for the same type parameter can be specified in a comma-separated list

```
// generics_constraints_2.cpp
// compile with: /c /clr
using namespace System;
using namespace System::Collections::Generic;
generic <typename T>
where T : List<T>, IComparable<T>
ref class List {};
```

With multiple type parameters, use one **where** clause for each type parameter. For example:

```
// generics_constraints_3.cpp
// compile with: /c /clr
using namespace System;
using namespace System::Collections::Generic;

generic <typename K, typename V>
   where K: IComparable<K>
   where V: IComparable<K>
ref class Dictionary {};
```

Use constraints in your code according to the following rules:

- If multiple constraints are listed, the constraints may be listed in any order.
- Constraints can also be class types, such as abstract base classes. However, constraints can't be value types or `sealed` classes.
- Constraints can't themselves be type parameters, but they can involve the type parameters in an open constructed type. For example:

```
// generics_constraints_4.cpp
// compile with: /c /clr
generic <typename T>
ref class G1 {};

generic <typename Type1, typename Type2>
where Type1 : G1<Type2>   // OK, G1 takes one type parameter
ref class G2{};
```

Examples

The following example demonstrates using constraints to call instance methods on type parameters.

```
// generics_constraints_5.cpp
// compile with: /clr
using namespace System;

interface class IAge {
   int Age();
};

ref class MyClass {
public:
   generic <class ItemType> where ItemType : IAge
   bool isSenior(ItemType item) {
      // Because of the constraint,
      // the Age method can be called on ItemType.
      if (item->Age() >= 65)
         return true;
      else
         return false;
   }
};

ref class Senior : IAge {
public:
   virtual int Age() {
      return 70;
   }
};

ref class Adult: IAge {
public:
   virtual int Age() {
      return 30;
   }
};

int main() {
   MyClass^ ageGuess = gcnew MyClass();
   Adult^ parent = gcnew Adult();
   Senior^ grandfather = gcnew Senior();

   if (ageGuess->isSenior<Adult^>(parent))
      Console::WriteLine("\"parent\" is a senior");
   else
      Console::WriteLine("\"parent\" is not a senior");

   if (ageGuess->isSenior<Senior^>(grandfather))
      Console::WriteLine("\"grandfather\" is a senior");
   else
      Console::WriteLine("\"grandfather\" is not a senior");
}
```

```
"parent" is not a senior
"grandfather" is a senior
```

When a generic type parameter is used as a constraint, it's called a *naked type constraint*. Naked type constraints are useful when a member function with its own type parameter needs to constrain that parameter to the type parameter of the containing type.

In the following example, `T` is a naked type constraint in the context of the `Add` method.

Naked type constraints can also be used in generic class definitions. The usefulness of naked type constraints with generic classes is limited because the compiler can assume nothing about a naked type constraint except that it derives from Object. Use naked type constraints on generic classes in scenarios in which you wish to enforce an inheritance relationship between two type parameters.

```
// generics_constraints_6.cpp
// compile with: /clr /c
generic <class T>
ref struct List {
   generic <class U>
   where U : T
   void Add(List<U> items)  {}
};

generic <class A, class B, class C>
where A : C
ref struct SampleClass {};
```

See also

Generics

Consuming Generics (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

Generics authored in one .NET (or UWP) language may be used in other languages. Unlike templates, a generic in a compiled assembly still remains generic. Thus, one may instantiate the generic type in a different assembly and even in a different language than the assembly in which the generic type was defined.

Example: Generic class defined in C#

This example shows a generic class defined in C#.

```
// consuming_generics_from_other_NET_languages.cs
// compile with: /target:library
// a C# program
public class CircularList<ItemType> {
   class ListNode    {
      public ItemType m_item;
      public ListNode next;
      public ListNode(ItemType item) {
         m_item = item;
      }
   }

   ListNode first, last;

   public CircularList() {}

   public void Add(ItemType item) {
      ListNode newnode = new ListNode(item);
      if (first == null) {
         first = last = newnode;
         first.next = newnode;
         last.next = first;
      }
      else {
         newnode.next = first;
         first = newnode;
         last.next = first;
      }
   }

   public void Remove(ItemType item) {
      ListNode iter = first;
      if (first.m_item.Equals( item )) {
         first =
         last.next = first.next;
      }
      for ( ; iter != last ; iter = iter.next )
         if (iter.next.m_item.Equals( item )) {
              if (iter.next == last)
                  last = iter;
              iter.next = iter.next.next;
              return;
          }
   }

   public void PrintAll() {
      ListNode iter = first;
      do {
         System.Console.WriteLine( iter.m_item );
         iter = iter.next;
      } while (iter != last);
   }
}
```

Example: Consume assembly authored in C#

This example consumes the assembly authored in C#.

```
// consuming_generics_from_other_NET_languages_2.cpp
// compile with: /clr
#using <consuming_generics_from_other_NET_languages.dll>
using namespace System;
class NativeClass {};
ref class MgdClass {};

int main() {
   CircularList<int>^ circ1 = gcnew CircularList<int>();
   CircularList<MgdClass^>^ circ2 = gcnew CircularList<MgdClass^>();

   for (int i = 0 ; i < 100 ; i += 10)
      circ1->Add(i);
   circ1->Remove(50);
   circ1->PrintAll();
}
```

The example produces this output:

```
90
80
70
60
40
30
20
10
```

See also

Generics

Generics and Templates (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

Generics and templates are both language features that provide support for parameterized types. However, they are different and have different uses. This topic provides an overview of the many differences.

For more information, see Windows Runtime and Managed Templates.

Comparing Templates and Generics

Key differences between generics and C++ templates:

- Generics are generic until the types are substituted for them at runtime. Templates are specialized at compile time so they are not still parameterized types at runtime
- The common language runtime specifically supports generics in MSIL. Because the runtime knows about generics, specific types can be substituted for generic types when referencing an assembly containing a generic type. Templates, in contrast, resolve into ordinary types at compile time and the resulting types may not be specialized in other assemblies.
- Generics specialized in two different assemblies with the same type arguments are the same type. Templates specialized in two different assemblies with the same type arguments are considered by the runtime to be different types.
- Generics are generated as a single piece of executable code which is used for all reference type arguments (this is not true for value types, which have a unique implementation per value type). The JIT compiler knows about generics and is able to optimize the code for the reference or value types that are used as type arguments. Templates generate separate runtime code for each specialization.
- Generics do not allow non-type template parameters, such as `template <int i> C {}`. Templates allow them.
- Generics do not allow explicit specialization (that is, a custom implementation of a template for a specific type). Templates do.
- Generics do not allow partial specialization (a custom implementation for a subset of the type arguments). Templates do.
- Generics do not allow the type parameter to be used as the base class for the generic type. Templates do.
- Templates support template-template parameters (e.g. `template<template<class T> class X> class MyClass`), but generics do not.

Combining Templates and Generics

The basic difference in generics has implications for building applications that combine templates and generics. For example, suppose you have a template class that you want to create a generic wrapper for to expose that template to other languages as a generic. You cannot have the generic take a type parameter that it then passes though to the template, since the template needs to have that type parameter at compile time, but the generic won't resolve the type parameter until runtime. Nesting a template inside a generic won't work either because there's no way to expand the templates at compile time for arbitrary generic types that could be instantiated at runtime.

Example

Description

The following example shows a simple example of using templates and generics together. In this example, the template class passes its parameter through to the generic type. The reverse is not possible.

This idiom could be used when you want to build on an existing generic API with template code that is local to a C++/CLI assembly, or when you need to add an extra layer of parameterization to a generic type, to take advantage of certain features of templates not supported by generics.

Code

```
// templates_and_generics.cpp
// compile with: /clr
using namespace System;

generic <class ItemType>
ref class MyGeneric {
   ItemType m_item;

public:
   MyGeneric(ItemType item) : m_item(item) {}
   void F() {
      Console::WriteLine("F");
   }
};

template <class T>
public ref class MyRef {
MyGeneric<T>^ ig;

public:
   MyRef(T t) {
      ig = gcnew MyGeneric<T>(t);
      ig->F();
    }
};

int main() {
   // instantiate the template
   MyRef<int>^ mref = gcnew MyRef<int>(11);
}
```

```
F
```

See also

Generics

How to: Improve Performance with Generics (C++/CLI)

5/13/2022 • 3 minutes to read • Edit Online

With generics, you can create reusable code based on a type parameter. The actual type of the type parameter is deferred until called by client code. For more information on generics, see Generics.

This article will discuss how generics can help increase the performance of an application that uses collections.

Example: Two main drawbacks of .NET Framework collections

The .NET Framework comes with many collection classes in the System.Collections namespace. Most of these collections operate on objects of type System.Object. This allows collections to store any type, since all types in the .NET Framework, even value types, derive from System.Object. However, there are two drawbacks to this approach.

First, if the collection is storing value types such as integers, the value must be boxed before being added to the collection and unboxed when the value is retrieved from the collection. These are expensive operations.

Second, there is no way to control which types can be added to a collection. It is perfectly legal to add an integer and a string to the same collection, even though this is probably not what was intended. Therefore, in order for your code to be type safe, you have to check that the type retrieved from the collection really is what was expected.

The following code example shows the two main drawbacks of the .NET Framework collections before generics.

```
// perf_pre_generics.cpp
// compile with: /clr

using namespace System;
using namespace System::Collections;

int main()
{
    // This Stack can contain any type.
    Stack ^s = gcnew Stack();

    // Push an integer to the Stack.
    // A boxing operation is performed here.
    s->Push(7);

    // Push a String to the same Stack.
    // The Stack now contains two different data types.
    s->Push("Seven");

    // Pop the items off the Stack.
    // The item is returned as an Object, so a cast is
    // necessary to convert it to its proper type.
    while (s->Count> 0)
    {
        Object ^o = s->Pop();
        if (o->GetType() == Type::GetType("System.String"))
        {
            Console::WriteLine("Popped a String: {0}", (String ^)o);
        }
        else if (o->GetType() == Type::GetType("System.Int32"))
        {
            Console::WriteLine("Popped an int: {0}", (int)o);
        }
        else
        {
            Console::WriteLine("Popped an unknown type!");
        }
    }
}
```

```
Popped a String: Seven
Popped an int: 7
```

Example: Benefit of using generic collection

The new System.Collections.Generic namespace contains many of the same collections found in the System.Collections namespace, but they have been modified to accept generic type parameters. This eliminates the two drawbacks of non-generic collections: the boxing and unboxing of value types and the inability to specify the types to be stored in the collections. Operations on the two collections are identical; they differ only in how they are instantiated.

Compare the example written above with this example that uses a generic Stack<T> collection. On large collections that are frequently accessed, the performance of this example will be significantly greater than the preceding example.

```
// perf_post_generics.cpp
// compile with: /clr

#using <System.dll>

using namespace System;
using namespace System::Collections::Generic;

int main()
{
    // This Stack can only contain integers.
    Stack<int> ^s = gcnew Stack<int>();

    // Push an integer to the Stack.
    // A boxing operation is performed here.
    s->Push(7);
    s->Push(14);

    // You can no longer push a String to the same Stack.
    // This will result in compile time error C2664.
    //s->Push("Seven");

    // Pop an item off the Stack.
    // The item is returned as the type of the collection, so no
    // casting is necessary and no unboxing is performed for
    // value types.
    int i = s->Pop();
    Console::WriteLine(i);

    // You can no longer retrieve a String from the Stack.
    // This will result in compile time error C2440.
    //String ^str = s->Pop();
}
```

```
14
```

See also

Generics

`interface class` (C++/CLI and C++/CX)

5/13/2022 • 3 minutes to read • Edit Online

Declares an interface. For information on native interfaces, see `__interface`.

All runtimes

Syntax

```
interface_access interface class name : inherit_access base_interface {};
interface_access interface struct name : inherit_access base_interface {};
```

Parameters

`interface_access`
The accessibility of an interface outside the assembly. Possible values are `public` and `private`. `private` is the default. Nested interfaces can't have an *`interface_access`* specifier.

`name`
The name of the interface.

`inherit_access`
The accessibility of *`base_interface`*. The only permitted accessibility for a base interface is `public` (the default).

`base_interface`
(Optional) A base interface for interface *`name`*.

Remarks

`interface struct` is equivalent to `interface class`.

An interface can contain declarations for functions, events, and properties. All interface members have public accessibility. An interface can also contain static data members, functions, events, and properties, and these static members must be defined in the interface.

An interface defines how a class may be implemented. An interface isn't a class and classes can only implement interfaces. When a class defines a function declared in an interface, the function is implemented, not overridden. Therefore, name lookup doesn't include interface members.

A `class` or `struct` that derives from an interface must implement all members of the interface. When implementing interface *`name`*, you must also implement the interfaces in the *`base_interface`* list.

For more information, see:

- Interface static constructor
- Generic interfaces (C++/CLI)

For information on other CLR types, see Classes and Structs.

You can detect at compile time if a type is an interface with `__is_interface_class(type)`. For more information, see Compiler support for type traits.

In the development environment, you can get F1 help on these keywords by highlighting the keyword (for example, `interface class`) and pressing **F1**.

Windows Runtime

Remarks

(There are no remarks for this language feature that apply to only the Windows Runtime.)

Requirements

Compiler option: `/ZW`

Common Language Runtime

Remarks

(There are no remarks for this language feature that apply to only the common language runtime.)

Requirements

Compiler option: `/clr`

Examples

The following code example demonstrates how an interface can define the behavior of a clock function.

```
// mcppv2_interface_class.cpp
// compile with: /clr
using namespace System;

public delegate void ClickEventHandler(int, double);

// define interface with nested interface
public interface class Interface_A {
   void Function_1();

   interface class Interface_Nested_A {
      void Function_2();
   };
};

// interface with a base interface
public interface class Interface_B : Interface_A {
   property int Property_Block;
   event ClickEventHandler^ OnClick;
   static void Function_3() { Console::WriteLine("in Function_3"); }
};

// implement nested interface
public ref class MyClass : public Interface_A::Interface_Nested_A {
public:
   virtual void Function_2() { Console::WriteLine("in Function_2"); }
};

// implement interface and base interface
public ref class MyClass2 : public Interface_B {
private:
   int MyInt;

public:
   // implement non-static function
   virtual void Function_1() { Console::WriteLine("in Function_1"); }

   // implement property
   property int Property_Block {
      virtual int get() { return MyInt; }
      virtual void set(int value) { MyInt = value; }
   }
   // implement event
   virtual event ClickEventHandler^ OnClick;
   void FireEvents() {
      OnClick(7, 3.14159);
   }
};

// class that defines method called when event occurs
ref class EventReceiver {
public:
   void OnMyClick(int i, double d) {
      Console::WriteLine("OnClick: {0}, {1}", i, d);
   }
};

int main() {
   // call static function in an interface
   Interface_B::Function_3();

   // instantiate class that implements nested interface
   MyClass ^ x = gcnew MyClass;
   x->Function_2();

   // instantiate class that implements interface with base interface
   MyClass2 ^ y = gcnew MyClass2;
   y->Function_1();
   y->Property_Block = 8;
   Console::WriteLine(y->Property_Block);

   EventReceiver^ MyEventReceiver = gcnew EventReceiver();

   // hook handler to event
   y->OnClick += gcnew ClickEventHandler(MyEventReceiver, &EventReceiver::OnMyClick);

   // invoke events
   y->FireEvents();

   // unhook handler to event
   y->OnClick -= gcnew ClickEventHandler(MyEventReceiver, &EventReceiver::OnMyClick);

   // call implemented function via interface handle
   Interface_A^ hi = gcnew MyClass2();
   hi->Function_1();
}
```

```
in Function_3

in Function_2

in Function_1

8

OnClick: 7, 3.14159

in Function_1
```

The following code sample shows two ways to implement functions with the same signature declared in multiple interfaces and where those interfaces are used by a class.

```
// mcppv2_interface_class_2.cpp
// compile with: /clr /c
interface class I {
   void Test();
   void Test2();
};

interface class J : I {
   void Test();
   void Test2();
};

ref struct R : I, J {
   // satisfies the requirement to implement Test in both interfaces
   virtual void Test() {}

   // implement both interface functions with explicit overrides
   virtual void A() = I::Test2 {}
   virtual void B() = J::Test2 {}
};
```

See also

Component Extensions for .NET and UWP

`literal` (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

A variable (data member) marked as `literal` in a `/clr` compilation is a compile-time constant. It's the native equivalent of a C# `const` variable.

All Platforms

Remarks

(There are no remarks for this language feature that apply to all runtimes.)

Windows Runtime

Remarks

(There are no remarks for this language feature that apply to only the Windows Runtime.)

Common Language Runtime

Remarks

A data member marked as `literal` must be initialized when declared. And, the value must be a constant integral, enum, or string type. Conversion from the type of the initialization expression to the type of the `literal` data member can't require a user-defined conversion.

No memory is allocated for the `literal` field at runtime; the compiler only inserts its value in the metadata for the class. The `literal` value is treated as a compile-time constant. The closest equivalent in Standard C++ is `constexpr`, but a data member can't be `constexpr` in C++/CLI.

A variable marked as `literal` differs from one marked `static const`. A `static const` data member isn't made available in metadata to other compilers. For more information, see `static` and `const`.

`literal` is a context-sensitive keyword. For more information, see Context-sensitive keywords.

Examples

This example shows that a `literal` variable implies `static`.

```
// mcppv2_literal.cpp
// compile with: /clr
ref struct X {
   literal int i = 4;
};

int main() {
   int value = X::i;
}
```

The following sample shows the effect of `literal` in metadata:

```
// mcppv2_literal2.cpp
// compile with: /clr /LD
public ref struct A {
   literal int lit = 0;
   static const int sc = 1;
};
```

Notice the difference in the metadata for `sc` and `lit`: the `modopt` directive is applied to `sc`, meaning it can be ignored by other compilers.

```
.field public static int32 modopt([mscorlib]System.Runtime.CompilerServices.IsConst) sc = int32(0x00000001)
```

```
.field public static literal int32 lit = int32(0x00000000)
```

The following sample, authored in C#, references the metadata created in the previous sample and shows the effect of `literal` and `static const` variables:

```
// mcppv2_literal3.cs
// compile with: /reference:mcppv2_literal2.dll
// A C# program
class B {
   public static void Main() {
      // OK
      System.Console.WriteLine(A.lit);
      System.Console.WriteLine(A.sc);

      // C# does not enforce C++ const
      A.sc = 9;
      System.Console.WriteLine(A.sc);

      // C# enforces const for a literal
      A.lit = 9;   // CS0131

      // you can assign a C++ literal variable to a C# const variable
      const int i = A.lit;
      System.Console.WriteLine(i);

      // but you cannot assign a C++ static const variable
      // to a C# const variable
      const int j = A.sc;   // CS0133
      System.Console.WriteLine(j);
   }
}
```

Requirements

Compiler option: `/clr`

See also

Component Extensions for .NET and UWP

Windows Runtime and Managed Templates (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

Templates enable you to define a prototype of a Windows Runtime or common language runtime type, and then instantiate variations of that type by using different template type parameters.

All Runtimes

You can create templates from value or reference types. For more information about creating value or reference types, see Classes and Structs.

For more information about standard C++ class templates, see Class Templates.

Windows Runtime

(There are no remarks for this language feature that apply to only the Windows Runtime.)

Requirements

Compiler option: `/ZW`

Common Language Runtime

There are some limitations to creating class templates from managed types, which are demonstrated in the following code examples.

Requirements

Compiler option: `/clr`

Examples

It is possible to instantiate a generic type with a managed type template parameter, but you cannot instantiate a managed template with a generic type template parameter. This is because generic types are resolved at runtime. For more information, see Generics and Templates (C++/CLI).

```
// managed_templates.cpp
// compile with: /clr /c

generic<class T>
ref class R;

template<class T>
ref class Z {
   // Instantiate a generic with a template parameter.
   R<T>^ r;    // OK
};

generic<class T>
ref class R {
   // Cannot instantiate a template with a generic parameter.
   Z<T>^ z;   // C3231
};
```

A generic type or function cannot be nested in a managed template.

```
// managed_templates_2.cpp
// compile with: /clr /c

template<class T> public ref class R {
   generic<class T> ref class W {};   // C2959
};
```

You cannot access templates defined in a referenced assembly with C++/CLI language syntax, but you can use reflection. If a template is not instantiated, it's not emitted in the metadata. If a template is instantiated, only referenced member functions will appear in metadata.

```
// managed_templates_3.cpp
// compile with: /clr

// Will not appear in metadata.
template<class T> public ref class A {};

// Will appear in metadata as a specialized type.
template<class T> public ref class R {
public:
   // Test is referenced, will appear in metadata
   void Test() {}

   // Test2 is not referenced, will not appear in metadata
   void Test2() {}
};

// Will appear in metadata.
generic<class T> public ref class G { };

public ref class S { };

int main() {
   R<int>^ r = gcnew R<int>;
   r->Test();
}
```

You can change the managed modifier of a class in a partial specialization or explicit specialization of a class template.

```
// managed_templates_4.cpp
// compile with: /clr /c

// class template
// ref class
template <class T>
ref class A {};

// partial template specialization
// value type
template <class T>
value class A <T *> {};

// partial template specialization
// interface
template <class T>
interface class A<T%> {};

// explicit template specialization
// native class
template <>
class A <int> {};
```

See also

Component Extensions for .NET and UWP

new (new slot in vtable) (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

The `new` keyword indicates that a virtual member will get a new slot in the vtable.

All Runtimes

(There are no remarks for this language feature that apply to all runtimes.)

Windows Runtime

Not supported in Windows Runtime.

Common Language Runtime

Remarks

In a `/clr` compilation, `new` indicates that a virtual member will get a new slot in the vtable; that the function does not override a base class method.

`new` causes the newslot modifier to be added to the IL for the function. For more information about newslot, see:

- MethodInfo.GetBaseDefinition()
- System.Reflection.MethodAttributes

Requirements

Compiler option: `/clr`

Examples

The following sample shows the effect of `new`.

```
// newslot.cpp
// compile with: /clr
ref class C {
public:
   virtual void f() {
      System::Console::WriteLine("C::f() called");
   }

   virtual void g() {
      System::Console::WriteLine("C::g() called");
   }
};

ref class D : public C {
public:
   virtual void f() new {
      System::Console::WriteLine("D::f() called");
   }

   virtual void g() override {
      System::Console::WriteLine("D::g() called");
   }
};

ref class E : public D {
public:
   virtual void f() override {
      System::Console::WriteLine("E::f() called");
   }
};

int main() {
   D^ d = gcnew D;
   C^ c = gcnew D;

   c->f();   // calls C::f
   d->f();   // calls D::f

   c->g();   // calls D::g
   d->g();   // calls D::g

   D ^ e = gcnew E;
   e->f();   // calls E::f
}
```

```
C::f() called

D::f() called

D::g() called

D::g() called

E::f() called
```

See also

Component Extensions for .NET and UWP
Override Specifiers

nullptr (C++/CLI and C++/CX)

5/13/2022 • 4 minutes to read • Edit Online

The `nullptr` keyword represents a *null pointer value*. Use a null pointer value to indicate that an object handle, interior pointer, or native pointer type does not point to an object.

Use `nullptr` with either managed or native code. The compiler emits appropriate but different instructions for managed and native null pointer values. For information about using the ISO standard C++ version of this keyword, see nullptr.

The **__nullptr** keyword is a Microsoft-specific keyword that has the same meaning as `nullptr`, but applies to only native code. If you use `nullptr` with native C/C++ code and then compile with the /clr compiler option, the compiler cannot determine whether `nullptr` indicates a native or managed null pointer value. To make your intention clear to the compiler, use `nullptr` to specify a managed value or **__nullptr** to specify a native value.

The `nullptr` keyword is equivalent to **Nothing** in Visual Basic and **null** in C#.

Usage

The `nullptr` keyword can be used anywhere a handle, native pointer, or function argument can be used.

The `nullptr` keyword is not a type and is not supported for use with:

- sizeof
- typeid
- `throw nullptr` (although `throw (Object^)nullptr;` will work)

The `nullptr` keyword can be used in the initialization of the following pointer types:

- Native pointer
- Windows Runtime handle
- Managed handle
- Managed interior pointer

The `nullptr` keyword can be used to test if a pointer or handle reference is null before the reference is used.

Function calls among languages that use null pointer values for error checking should be interpreted correctly.

You cannot initialize a handle to zero; only `nullptr` can be used. Assignment of constant 0 to an object handle produces a boxed `Int32` and a cast to `Object^`.

Example: `nullptr` keyword

The following code example demonstrates that the `nullptr` keyword can be used wherever a handle, native pointer, or function argument can be used. And the example demonstrates that the `nullptr` keyword can be used to check a reference before it is used.

```
// mcpp_nullptr.cpp
// compile with: /clr
value class V {};
ref class G {};
void f(System::Object ^) {}

int main() {
// Native pointer.
   int *pN = nullptr;
// Managed handle.
   G ^pG = nullptr;
   V ^pV1 = nullptr;
// Managed interior pointer.
   interior_ptr<V> pV2 = nullptr;
// Reference checking before using a pointer.
   if (pN == nullptr) {}
   if (pG == nullptr) {}
   if (pV1 == nullptr) {}
   if (pV2 == nullptr) {}
// nullptr can be used as a function argument.
   f(nullptr);   // calls f(System::Object ^)
}
```

Example: Use `nullptr` and zero interchangeably

The following code example shows that `nullptr` and zero can be used interchangeably on native pointers.

```
// mcpp_nullptr_1.cpp
// compile with: /clr
class MyClass {
public:
   int i;
};

int main() {
   MyClass * pMyClass = nullptr;
   if ( pMyClass == nullptr)
      System::Console::WriteLine("pMyClass == nullptr");

   if ( pMyClass == 0)
      System::Console::WriteLine("pMyClass == 0");

   pMyClass = 0;
   if ( pMyClass == nullptr)
      System::Console::WriteLine("pMyClass == nullptr");

   if ( pMyClass == 0)
      System::Console::WriteLine("pMyClass == 0");
}
```

```
pMyClass == nullptr

pMyClass == 0

pMyClass == nullptr

pMyClass == 0
```

Example: Interpret `nullptr` as a handle

The following code example shows that `nullptr` is interpreted as a handle to any type or a native pointer to any type. In case of function overloading with handles to different types, an ambiguity error will be generated. The `nullptr` would have to be explicitly cast to a type.

```
// mcpp_nullptr_2.cpp
// compile with: /clr /LD
void f(int *){}
void f(int ^){}

void f_null() {
   f(nullptr);   // C2668
   // try one of the following lines instead
   f((int *) nullptr);
   f((int ^) nullptr);
}
```

Example: Cast `nullptr`

The following code example shows that casting `nullptr` is allowed and returns a pointer or handle to the cast type that contains the `nullptr` value.

```
// mcpp_nullptr_3.cpp
// compile with: /clr /LD
using namespace System;
template <typename T>
void f(T) {}   // C2036 cannot deduce template type because nullptr can be any type

int main() {
   f((Object ^) nullptr);   // T = Object^, call f(Object ^)

   // Delete the following line to resolve.
   f(nullptr);

   f(0);   // T = int, call f(int)
}
```

Example: Pass `nullptr` as a function parameter

The following code example shows that `nullptr` can be used as a function parameter.

```
// mcpp_nullptr_4.cpp
// compile with: /clr
using namespace System;
void f(Object ^ x) {
   Console::WriteLine("test");
}

int main() {
   f(nullptr);
}
```

```
test
```

Example: Default initialization

The following code example shows that when handles are declared and not explicitly initialized, they are default initialized to `nullptr`.

```
// mcpp_nullptr_5.cpp
// compile with: /clr
using namespace System;
ref class MyClass {
public:
   void Test() {
      MyClass ^pMyClass;   // gc type
      if (pMyClass == nullptr)
         Console::WriteLine("NULL");
   }
};

int main() {
   MyClass ^ x = gcnew MyClass();
   x -> Test();
}
```

```
NULL
```

Example: Assign `nullptr` to a native pointer

The following code example shows that `nullptr` can be assigned to a native pointer when you compile with `/clr`.

```
// mcpp_nullptr_6.cpp
// compile with: /clr
int main() {
   int * i = 0;
   int * j = nullptr;
}
```

Requirements

Compiler option: (Not required; supported by all code generation options, including `/ZW` and `/clr`)

See also

Component Extensions for .NET and UWP
nullptr

Override Specifiers (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

Override specifiers modify how inherited types and members of inherited types behave in derived types.

All Runtimes

Remarks

For more information about override specifiers, see:

- abstract
- new (new slot in vtable)
- override
- sealed
- Override Specifiers and Native Compilations

`abstract` and `sealed` are also valid on type declarations, where they do not act as override specifiers.

For information about explicitly overriding base class functions, see Explicit Overrides.

Windows Runtime

(There are no remarks for this language feature that apply to only the Windows Runtime.)

Requirements

Compiler option: `/ZW`

Common Language Runtime

(There are no remarks for this language feature that apply to only the common language runtime.)

Requirements

Compiler option: `/clr`

See also

Component Extensions for .NET and UWP

override (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

The **override** context-sensitive keyword indicates that a member of a type overrides a base class or a base interface member.

Remarks

The **override** keyword is valid when compiling for native targets (default compiler option), Windows Runtime targets (`/ZW` compiler option), or common language runtime targets (`/clr` compiler option).

For more information about override specifiers, see override Specifier and Override Specifiers and Native Compilations.

For more information about context-sensitive keywords, see Context-Sensitive Keywords.

Examples

The following code example shows that **override** can also be used in native compilations.

```
// override_keyword_1.cpp
// compile with: /c
struct I1 {
   virtual void f();
};

struct X : public I1 {
   virtual void f() override {}
};
```

Windows Runtime example

The following code example shows that **override** can be used in Windows Runtime compilations.

```
// override_keyword_2.cpp
// compile with: /ZW /c
ref struct I1 {
   virtual void f();
};

ref struct X : public I1 {
   virtual void f() override {}
};
```

Requirements

Compiler option: `/ZW`

C++/CLI example

The following code example shows that **override** can be used in common language runtime compilations.

```
// override_keyword_3.cpp
// compile with: /clr /c
ref struct I1 {
   virtual void f();
};

ref struct X : public I1 {
   virtual void f() override {}
};
```

Requirements

Compiler option: `/clr`

See also

override Specifier
Override Specifiers

partial (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

The **partial** keyword enables different parts of the same ref class to be authored independently and in different files.

All Runtimes

(This language feature applies only to the Windows Runtime.)

Windows Runtime

For a ref class that has two partial definitions, the **partial** keyword is applied to the first occurrence of the definition, and this is typically done by auto-generated code, so that a human coder doesn't use the keyword very often. For all subsequent partial definitions of the class, omit the **partial** modifier from the *class-key* keyword and class identifier. When the compiler encounters a previously defined ref class and class identifier but no **partial** keyword, it internally combines all of the parts of the ref class definition into one definition.

Syntax

```
partial class-key identifier {
   /* The first part of the partial class definition.
      This is typically auto-generated */
}
// ...
class-key identifier {
   /* The subsequent part(s) of the class definition. The same
      identifier is specified, but the "partial" keyword is omitted. */
}
```

Parameters

class-key
A keyword that declares a class or struct that is supported by the Windows Runtime. Either **ref class**, **value class**, **ref struct**, or **value struct**.

identifier
The name of the defined type.

Remarks

A partial class supports scenarios where you modify one part of a class definition in one file, and automatic code-generating software—for example, the XAML designer—modifies code in the same class in another file. By using a partial class, you can prevent the automatic code generator from overwriting your code. In a Visual Studio project, the **partial** modifier is applied automatically to the generated file.

Contents: With two exceptions, a partial class definition can contain anything that the full class definition could contain if the **partial** keyword was omitted. However, you can't specify class accessibility (for example, `public partial class X { ... };`), or a **declspec**.

Access specifiers used in a partial class definition for *identifier* do not affect the default accessibility in a subsequent partial or full class definition for *identifier*. Inline definitions of static data members are allowed.

Declaration: A partial definition of a class *identifier* only introduces the name *identifier*, but *identifier* cannot be used in a way that requires a class definition. The name *identifier* can't be used to know the size of *identifier*, or to use a base or member of *identifier* until after the compiler encounters the full definition of *identifier*.

Number and ordering: There can be zero or more partial class definitions for *identifier*. Every partial class definition of *identifier* must lexically precede the one full definition of *identifier* (if there is a full definition; otherwise, the class can't be used except as if forward-declared) but need not precede forward declarations of *identifier*. All class-keys must match.

Full definition: At the point of the full definition of the class *identifier*, the behavior is the same as if the definition of *identifier* had declared all base classes, members, etc. in the order in which they were encountered and defined in the partial classes.

Templates: A partial class cannot be a template.

Generics: A partial class can be a generic if the full definition could be generic. But every partial and full class must have exactly the same generic parameters, including formal parameter names.

For more information about how to use the **partial** keyword, see Partial Classes (C++/CX).

Requirements

Compiler option: `/ZW`

Common Language Runtime

(This language feature does not apply to the Common Language Runtime.)

See also

Partial Classes (C++/CX)

`property` (C++/CLI and C++/CX)

5/13/2022 • 5 minutes to read • Edit Online

Declares a *property*, which is a member function that behaves and is accessed like a data member or an array element.

All runtimes

You can declare one of the following types of properties.

- **simple property**
  By default, creates a `set` accessor that assigns the property value, a `get` accessor that retrieves the property value, and a compiler-generated private data member that contains the property value.

- **property block**
  Use a property block to create user-defined `get` or `set` accessors. The property is read and write if both the `get` and `set` accessors are defined, read-only if only the `get` accessor is defined, and write-only if only the `set` accessor is defined.

  You need to explicitly declare a data member to contain the property value.

- **indexed property**
  A property block that you can use to get and set a property value that is specified by one or more indexes.

  You can create an indexed property that has either a user-defined property name or a *default* property name. The name of a default index property is the name of the class in which the property is defined. To declare a default property, specify the `default` keyword instead of a property name.

Explicitly declare a data member to contain the property value. For an indexed property, the data member is typically an array or a collection.

Syntax

```
property type property_name;

property type property_name {
   access-modifier type get() inheritance-modifier {property_body};
   access-modifier void set(type value) inheritance-modifier {property_body};
}

property type property_name[index_list] {
   access-modifier type get(index_list) inheritance-modifier {property_body};
   access-modifier void set(index_list, value) inheritance-modifier {property_body};
}

property type default[index_list] {
   access-modifier type get(index_list) inheritance-modifier {property_body};
   access-modifier void set(index_list, value) inheritance-modifier {property_body};
}
```

Parameters

`type`
The data type of the property value, and of the property itself.

`property_name`
The name of the property.

`access-modifier`
An access qualifier. Valid qualifiers are `static` and `virtual`.

The `get` or `set` accessors need not agree on the `virtual` qualifier, but they must agree on the `static` qualifier.

`inheritance-modifier`
An inheritance qualifier. Valid qualifiers are `abstract` and `sealed`.

`index_list`
A comma-delimited list of one or more indexes. Each index consists of an index type, and an optional identifier that can be used in the property method body.

`value`
The value to assign to the property in a `set` operation, or retrieve in a `get` operation.

`property_body`
The property method body of the `set` or `get` accessor. The `property_body` can use the `index_list` to access the underlying property data member, or as parameters in user-defined processing.

Windows Runtime

For more information, see Properties (C++/CX).

Requirements

Compiler option: `/ZW`

Common Language Runtime

Syntax

```
modifier property type property_name;

modifier property type property_name {
   modifier void set(type);
   modifier type get();
}
modifier property type property_name[index-list, value] {
   modifier void set(index-list, value);
   modifier type get(index-list);

modifier property type default[index];
}
```

Parameters

`modifier`
A modifier that can be used on either a property declaration or a get/set accessor method. Possible values are `static` and `virtual`.

`type`
The type of the value that is represented by the property.

`property_name`
Parameter(s) for the `raise` method; must match the signature of the delegate.

`index_list`
A comma-delimited list of one or more indexes, specified in square brackets (the subscript operator, `[]`). For each index, specify a type and optionally an identifier that can be used in the property method body.

Remarks

The first syntax example shows a *simple property*, which implicitly declares both a `set` and `get` method. The compiler automatically creates a private field to store the value of the property.

The second syntax example shows a *property block*, which explicitly declares both a `set` and `get` method.

The third syntax example shows a customer-defined *index property*. An index property takes parameters in addition to the value to be set or retrieved. Specify a name for the property. Unlike a simple property, the `set` and `get` methods of an index property must be explicitly defined, and so you must specify a name for the property.

The fourth syntax example shows a *default* property, which provides array-like access to an instance of the type. The keyword, `default`, serves only to specify a default property. The name of the default property is the name of the type in which the property is defined.

The `property` keyword can appear in a class, interface, or value type. A property can have a `get` function (read-only), a `set` function (write-only), or both (read-write).

A property name can't match the name of the managed class that contains it. The return type of the getter function must match the type of the last parameter of a corresponding setter function.

To client code, a property has the appearance of an ordinary data member, and can be written to or read from by using the same syntax as a data member.

The `get` and `set` methods need not agree on the `virtual` modifier.

The accessibility of the `get` and `set` method can differ.

The definition of a property method can appear outside the class body, just like an ordinary method.

The `get` and the `set` method for a property shall agree on the `static` modifier.

A property is scalar if its `get` and `set` methods fit the following description:

- The `get` method has no parameters, and has return type `T`.
- The `set` method has a parameter of type `T`, and return type `void`.

There shall be only one scalar property declared in a scope with the same identifier. Scalar properties cannot be overloaded.

When a property data member is declared, the compiler injects a data member—sometimes referred to as the "backing store"—in the class. However, the name of the data member is of a form such that you can't reference the member in the source as if it were an actual data member of the containing class. Use ildasm.exe to view the metadata for your type and see the compiler-generated name for the property's backing store.

Different accessibility is allowed for the accessor methods in a property block. That is, the `set` method can be `public` and the `get` method can be `private`. However, it's an error for an accessor method to have a less restrictive accessibility than what is on the declaration of the property itself.

`property` is a context-sensitive keyword. For more information, see Context-sensitive keywords.

Requirements

Compiler option: `/clr`

Examples

The following example shows the declaration and use of a property data member and a property block. It also shows that a property accessor can be defined out of class.

```
// mcppv2_property.cpp
// compile with: /clr
using namespace System;
public ref class C {
   int MyInt;
public:

   // property data member
   property String ^ Simple_Property;

   // property block
   property int Property_Block {

      int get();

      void set(int value) {
         MyInt = value;
      }
   }
};

int C::Property_Block::get() {
   return MyInt;
}

int main() {
   C ^ MyC = gcnew C();
   MyC->Simple_Property = "test";
   Console::WriteLine(MyC->Simple_Property);

   MyC->Property_Block = 21;
   Console::WriteLine(MyC->Property_Block);
}
```

```
test

21
```

See also

Component Extensions for .NET and UWP

safe_cast (C++/CLI and C++/CX)

5/13/2022 • 3 minutes to read • Edit Online

The **safe_cast** operation returns the specified expression as the specified type, if successful; otherwise, throws `InvalidCastException`.

All Runtimes

(There are no remarks for this language feature that apply to all runtimes.)

Syntax

```
[default]:: safe_cast< type-id >( expression )
```

Windows Runtime

safe_cast allows you to change the type of a specified expression. In situations where you fully expect a variable or parameter to be convertible to a certain type, you can use **safe_cast** without a **try-catch** block to detect programming errors during development. For more information, see Casting (C++/CX).

Syntax

```
[default]:: safe_cast< type-id >( expression )
```

Parameters

type-id
The type to convert *expression* to. A handle to a reference or value type, a value type, or a tracking reference to a reference or value type.

expression
An expression that evaluates to a handle to a reference or value type, a value type, or a tracking reference to a reference or value type.

Remarks

safe_cast throws `InvalidCastException` if it cannot convert *expression* to the type specified by *type-id*. To catch `InvalidCastException`, specify the /EH (Exception Handling Model) compiler option, and use a **try/catch** statement.

Requirements

Compiler option: `/ZW`

Examples

The following code example demonstrates how to use **safe_cast** with the Windows Runtime.

```
// safe_cast_ZW.cpp
// compile with: /ZW /EHsc

using namespace default;
using namespace Platform;

interface class I1 {};
interface class I2 {};
interface class I3 {};

ref class X : public I1, public I2 {};

int main(Array<String^>^ args) {
   I1^ i1 = ref new X;
   I2^ i2 = safe_cast<I2^>(i1);   // OK, I1 and I2 have common type: X
   // I2^ i3 = static_cast<I2^>(i1);   C2440 use safe_cast instead
   try {
      I3^ i4 = safe_cast<I3^>(i1);   // Fails because i1 is not derived from I3.
   }
   catch(InvalidCastException^ ic) {
   wprintf(L"Caught expected exception: %s\n", ic->Message);
   }
}
```

```
Caught expected exception: InvalidCastException
```

Common Language Runtime

safe_cast allows you to change the type of an expression and generate verifiable MSIL code.

Syntax

```
[cli]:: safe_cast< type-id >( expression )
```

Parameters

type-id
A handle to a reference or value type, a value type, or a tracking reference to a reference or value type.

expression
An expression that evaluates to a handle to a reference or value type, a value type, or a tracking reference to a reference or value type.

Remarks

The expression `safe_cast<` *type-id* `>(` *expression* `)` converts the operand *expression* to an object of type *type-id*.

The compiler will accept a static_cast in most places that it will accept a **safe_cast**. However, **safe_cast** is guaranteed to produce verifiable MSIL, where as a `static_cast` could produce unverifiable MSIL. See Pure and Verifiable Code (C++/CLI) and Peverify.exe (PEVerify Tool) for more information on verifiable code.

Like `static_cast`, **safe_cast** invokes user-defined conversions.

For more information about casts, see Casting Operators.

safe_cast does not apply a `const_cast` (cast away `const`).

safe_cast is in the cli namespace. See Platform, default, and cli Namespaces for more information.

For more information on **safe_cast**, see:

- C-Style Casts with /clr (C++/CLI)
- How to: Use safe_cast in C++/CLI

Requirements

Compiler option: `/clr`

Examples

One example of where the compiler will not accept a `static_cast` but will accept a **safe_cast** is for casts between unrelated interface types. With **safe_cast**, the compiler will not issue a conversion error and will perform a check at runtime to see if the cast is possible

```
// safe_cast.cpp
// compile with: /clr
using namespace System;

interface class I1 {};
interface class I2 {};
interface class I3 {};

ref class X : public I1, public I2 {};

int main() {
   I1^ i1 = gcnew X;
   I2^ i2 = safe_cast<I2^>(i1);   // OK, I1 and I2 have common type: X
   // I2^ i3 = static_cast<I2^>(i1);   C2440 use safe_cast instead
   try {
      I3^ i4 = safe_cast<I3^>(i1);   // fail at runtime, no common type
   }
   catch(InvalidCastException^) {
      Console::WriteLine("Caught expected exception");
   }
}
```

```
Caught expected exception
```

See also

Component Extensions for .NET and UWP

String (C++/CLI and C++/CX)

5/13/2022 • 4 minutes to read • Edit Online

The Windows Runtime and common language runtime represent strings as objects whose allocated memory is managed automatically. That is, you are not required to explicitly discard the memory for a string when the string variable goes out of scope or your application ends. To indicate that the lifetime of a string object is to be managed automatically, declare the string type with the handle-to-object (^) modifier.

Windows Runtime

The Windows Runtime architecture requires that the `String` data type be located in the `Platform` namespace. For your convenience, Visual C++ also provides the `string` data type, which is a synonym for `Platform::String`, in the `default` namespace.

Syntax

```
// compile with /ZW
using namespace Platform;
using namespace default;
   Platform::String^ MyString1 = "The quick brown fox";
   String^ MyString2 = "jumped over the lazy dog.";
   String^ MyString3 = "Hello, world!";
```

Requirements

Compiler option: `/ZW`

Common Language Runtime

When compiling with `/clr`, the compiler will convert string literals to strings of type String. To preserve backward compatibility with existing code there are two exceptions to this:

- Exception handling. When a string literal is thrown, the compiler will catch it as a string literal.
- Template deduction. When a string literal is passed as a template argument, the compiler will not convert it to a String. Note, string literals passed as a generic argument will be promoted to String.

The compiler also has built-in support for three operators, which you can override to customize their behavior:

- System::String^ operator +(System::String, System::String);
- System::String^ operator +(System::Object, System::String);
- System::String^ operator +(System::String, System::Object);

When passed a String, the compiler will box, if necessary, and then concatenate the object (with ToString) with the string.

NOTE

The caret ("^") indicates that the declared variable is a handle to a C++/CLI managed object.

For more information see String and Character Literals.

Requirements

Compiler option: **/clr**

Examples

The following code example demonstrates concatenating and comparing strings.

```
// string_operators.cpp
// compile with: /clr
// In the following code, the caret ("^") indicates that the
// declared variable is a handle to a C++/CLI managed object.
using namespace System;

int main() {
   String^ a = gcnew String("abc");
   String^ b = "def";   // same as gcnew form
   Object^ c = gcnew String("ghi");

   char d[100] = "abc";

   // variables of System::String returning a System::String
   Console::WriteLine(a + b);
   Console::WriteLine(a + c);
   Console::WriteLine(c + a);

   // accessing a character in the string
   Console::WriteLine(a[2]);

   // concatenation of three System::Strings
   Console::WriteLine(a + b + c);

   // concatenation of a System::String and string literal
   Console::WriteLine(a + "zzz");

   // you can append to a System::String^
   Console::WriteLine(a + 1);
   Console::WriteLine(a + 'a');
   Console::WriteLine(a + 3.1);

   // test System::String^ for equality
   a += b;
   Console::WriteLine(a);
   a = b;
   if (a == b)
      Console::WriteLine("a and b are equal");

   a = "abc";
   if (a != b)
      Console::WriteLine("a and b are not equal");

   // System:String^ and tracking reference
   String^% rstr1 = a;
   Console::WriteLine(rstr1);

   // testing an empty System::String^
   String^ n;
   if (n == nullptr)
      Console::WriteLine("n is empty");
}
```

```
abcdef

abcghi

ghiabc

c

abcdefghi

abczzz

abc1

abc97

abc3.1

abcdef

a and b are equal

a and b are not equal

abc

n is empty
```

The following sample shows that you can overload the compiler-provided operators, and that the compiler will find a function overload based on the String type.

```
// string_operators_2.cpp
// compile with: /clr
using namespace System;

// a string^ overload will be favored when calling with a String
void Test_Overload(const char * a) {
   Console::WriteLine("const char * a");
}
void Test_Overload(String^ a) {
   Console::WriteLine("String^ a");
}

// overload will be called instead of compiler defined operator
String^ operator +(String^ a, String^ b) {
   return ("overloaded +(String^ a, String^ b)");
}

// overload will be called instead of compiler defined operator
String^ operator +(Object^ a, String^ b) {
   return ("overloaded +(Object^ a, String^ b)");
}

// overload will be called instead of compiler defined operator
String^ operator +(String^ a, Object^ b) {
   return ("overloaded +(String^ a, Object^ b)");
}

int main() {
   String^ a = gcnew String("abc");
   String^ b = "def";   // same as gcnew form
   Object^ c = gcnew String("ghi");

   char d[100] = "abc";

   Console::WriteLine(a + b);
   Console::WriteLine(a + c);
   Console::WriteLine(c + a);

   Test_Overload("hello");
   Test_Overload(d);
}
```

```
overloaded +(String^ a, String^ b)

overloaded +(String^ a, Object^ b)

overloaded +(Object^ a, String^ b)

String^ a

const char * a
```

The following sample shows that the compiler distinguishes between native strings and String strings.

```
// string_operators_3.cpp
// compile with: /clr
using namespace System;
int func() {
   throw "simple string";   // const char *
};

int func2() {
   throw "string" + "string";   // returns System::String
};

template<typename T>
void func3(T t) {
   Console::WriteLine(T::typeid);
}

int main() {
   try {
      func();
   }
   catch(char * e) {
      Console::WriteLine("char *");
   }

   try {
      func2();
   }
   catch(String^ str) {
      Console::WriteLine("String^ str");
   }

   func3("string");   // const char *
   func3("string" + "string");   // returns System::String
}
```

```
char *

String^ str

System.SByte*

System.String
```

See also

Component Extensions for .NET and UWP
String and Character Literals
/clr (Common Language Runtime Compilation)

sealed (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

sealed is a context-sensitive keyword for ref classes that indicates that a virtual member cannot be overridden, or that a type cannot be used as a base type.

NOTE

The ISO C++11 Standard language introduced the final keyword. Use **final** on standard classes, and **sealed** on ref classes.

All Runtimes

Syntax

```
ref class identifier sealed {...};
virtual return-type identifier() sealed {...};
```

Parameters

identifier
The name of the function or class.

return-type
The type that's returned by a function.

Remarks

In the first syntax example, a class is sealed. In the second example, a virtual function is sealed.

Use the **sealed** keyword for ref classes and their virtual member functions. For more information, see Override Specifiers and Native Compilations.

You can detect at compile time whether a type is sealed by using the `__is_sealed(type)` type trait. For more information, see Compiler Support for Type Traits.

sealed is a context-sensitive keyword. For more information, see Context-Sensitive Keywords.

Windows Runtime

See Ref classes and structs.

Requirements

Compiler option: `/ZW`

Common Language Runtime

(There are no remarks for this language feature that apply to only the common language runtime.)

Requirements

Compiler option: `/clr`

Examples

This following code example shows the effect of **sealed** on a virtual member.

```
// sealed_keyword.cpp
// compile with: /clr
interface struct I1 {
   virtual void f();
   virtual void g();
};

ref class X : I1 {
public:
   virtual void f() {
      System::Console::WriteLine("X::f override of I1::f");
   }

   virtual void g() sealed {
      System::Console::WriteLine("X::f override of I1::g");
   }
};

ref class Y : public X {
public:
   virtual void f() override {
      System::Console::WriteLine("Y::f override of I1::f");
   }

   /*
   // the following override generates a compiler error
   virtual void g() override {
      System::Console::WriteLine("Y::g override of I1::g");
   }
   */
};

int main() {
   I1 ^ MyI = gcnew X;
   MyI -> f();
   MyI -> g();

   I1 ^ MyI2 = gcnew Y;
   MyI2 -> f();
}
```

```
X::f override of I1::f
X::f override of I1::g
Y::f override of I1::f
```

The next code example shows how to mark a class as sealed.

```
// sealed_keyword_2.cpp
// compile with: /clr
interface struct I1 {
   virtual void f();
};

ref class X sealed : I1 {
public:
   virtual void f() override {}
};

ref class Y : public X {   // C3246 base class X is sealed
public:
   virtual void f() override {}
};
```

See also

Component Extensions for .NET and UWP

typeid (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

Gets a value that indicates the type of an object.

NOTE

This topic refers to the C++ Component Extensions version of typeid. For the ISO C++ version of this keyword, see typeid Operator.

All Runtimes

Syntax

```
T::typeid
```

Parameters

T
A type name.

Windows Runtime

Syntax

```
Platform::Type^ type = T::typeid;
```

Parameters

T
A type name.

Remarks

In C++/CX, typeid returns a Platform::Type that is constructed from runtime type information.

Requirements

Compiler option: `/ZW`

Common Language Runtime

Syntax

```
System::Type^ type = T::typeid;
```

Parameters

type
The name of a type (abstract declarator) for which you want the `System::Type` object.

Remarks

`typeid` is used to get the Type for a type at compile time.

`typeid` is similar to getting the `System::Type` for a type at run time using GetType or GetType. However, `typeid` only accepts a type name as a parameter. If you want to use an instance of a type to get its `System::Type` name, use `GetType`.

`typeid` must be able to evaluate a type name (type) at compile time, whereas GetType evaluates the type to return at run time.

`typeid` can take a native type name or common language runtime alias for the native type name; see .NET Framework Equivalents to C++ Native Types (C++/CLI) for more information.

`typeid` also works with native types, although it will still return a `System::Type`. To get a type_info structure, use `typeid` Operator.

Requirements

Compiler option: `/clr`

Examples

The following example compares the typeid keyword to the `GetType()` member.

```
// keyword__typeid.cpp
// compile with: /clr
using namespace System;

ref struct G {
   int i;
};

int main() {
   G ^ pG = gcnew G;
   Type ^ pType = pG->GetType();
   Type ^ pType2 = G::typeid;

   if (pType == pType2)
      Console::WriteLine("typeid and GetType returned the same System::Type");
   Console::WriteLine(G::typeid);

   typedef float* FloatPtr;
   Console::WriteLine(FloatPtr::typeid);
}
```

```
typeid and GetType returned the same System::Type
G

System.Single*
```

The following sample shows that a variable of type System::Type can be used to get the attributes on a type. It also shows that for some types, you will have to create a typedef to use `typeid`.

```
// keyword__typeid_2.cpp
// compile with: /clr
using namespace System;
using namespace System::Security;
using namespace System::Security::Permissions;

typedef int ^ handle_to_int;
typedef int * pointer_to_int;

public ref class MyClass {};

class MyClass2 {};

[attribute(AttributeTargets::All)]
ref class AtClass {
public:
   AtClass(Type ^) {
      Console::WriteLine("in AtClass Type ^ constructor");
   }
};

[attribute(AttributeTargets::All)]
ref class AtClass2 {
public:
   AtClass2() {
      Console::WriteLine("in AtClass2 constructor");
   }
};

// Apply the AtClass and AtClass2 attributes to class B
[AtClass(MyClass::typeid), AtClass2]
[AttributeUsage(AttributeTargets::All)]
ref class B : Attribute {};

int main() {
   Type ^ MyType = B::typeid;

   Console::WriteLine(MyType->IsClass);

   array<Object^>^ MyArray = MyType -> GetCustomAttributes(true);
   for (int i = 0 ; i < MyArray->Length ; i++ )
      Console::WriteLine(MyArray[i]);

   if (int::typeid != pointer_to_int::typeid)
      Console::WriteLine("int::typeid != pointer_to_int::typeid, as expected");

   if (int::typeid == handle_to_int::typeid)
      Console::WriteLine("int::typeid == handle_to_int::typeid, as expected");
}
```

```
True

in AtClass2 constructor

in AtClass Type ^ constructor

AtClass2

System.AttributeUsageAttribute

AtClass

int::typeid != pointer_to_int::typeid, as expected

int::typeid == handle_to_int::typeid, as expected
```

See also

Component Extensions for .NET and UWP

User-Defined Attributes (C++/CLI and C++/CX)

5/13/2022 • 3 minutes to read • Edit Online

C++/CLI and C++/CX enable you to create platform-specific attributes that extend the metadata of an interface, class or structure, method, parameter, or enumeration. These attributes are distinct from the standard C++ attributes.

Windows Runtime

You can apply C++/CX attributes to properties, but not to constructors or methods.

Requirements

Compiler option: `/ZW`

Common Language Runtime

The information and syntax presented in this topic is meant to supersede the information presented in attribute.

You can define a custom attribute by defining a type and making Attribute a base class for the type and optionally applying the AttributeUsageAttribute attribute.

For more information, see:

- Attribute Targets
- Attribute Parameter Types

For information on signing assemblies in Visual C++, see Strong Name Assemblies (Assembly Signing) (C++/CLI).

Requirements

Compiler option: `/clr`

Examples

The following sample shows how to define a custom attribute.

```
// user_defined_attributes.cpp
// compile with: /clr /c
using namespace System;

[AttributeUsage(AttributeTargets::All)]
ref struct Attr : public Attribute {
   Attr(bool i){}
   Attr(){}
};

[Attr]
ref class MyClass {};
```

The following example illustrates some important features of custom attributes. For example, this example shows a common usage of the custom attributes: instantiating a server that can fully describe itself to clients.

```
// extending_metadata_b.cpp
// compile with: /clr
```

```
using namespace System;
using namespace System::Reflection;

public enum class Access { Read, Write, Execute };

// Defining the Job attribute:
[AttributeUsage(AttributeTargets::Class, AllowMultiple=true )]
public ref class Job : Attribute {
public:
   property int Priority {
      void set( int value ) { m_Priority = value; }
      int get() { return m_Priority; }
   }

   // You can overload constructors to specify Job attribute in different ways
   Job() { m_Access = Access::Read; }
   Job( Access a ) { m_Access = a; }
   Access m_Access;

protected:
   int m_Priority;
};

interface struct IService {
   void Run();
};

   // Using the Job attribute:
   // Here we specify that QueryService is to be read only with a priority of 2.
   // To prevent namespace collisions, all custom attributes implicitly
   // end with "Attribute".

[Job( Access::Read, Priority=2 )]
ref struct QueryService : public IService {
   virtual void Run() {}
};

// Because we said AllowMultiple=true, we can add multiple attributes
[Job(Access::Read, Priority=1)]
[Job(Access::Write, Priority=3)]
ref struct StatsGenerator : public IService {
   virtual void Run( ) {}
};

int main() {
   IService ^ pIS;
   QueryService ^ pQS = gcnew QueryService;
   StatsGenerator ^ pSG = gcnew StatsGenerator;

   //  use QueryService
   pIS = safe_cast<IService ^>( pQS );

   // use StatsGenerator
   pIS = safe_cast<IService ^>( pSG );

   // Reflection
   MemberInfo ^ pMI = pIS->GetType();
   array <Object ^ > ^ pObjs = pMI->GetCustomAttributes(false);

   // We can now quickly and easily view custom attributes for an
   // Object through Reflection */
   for( int i = 0; i < pObjs->Length; i++ ) {
      Console::Write("Service Priority = ");
      Console::WriteLine(static_cast<Job^>(pObjs[i])->Priority);
      Console::Write("Service Access = ");
      Console::WriteLine(static_cast<Job^>(pObjs[i])->m_Access);
   }
}
```

```
Service Priority = 0

Service Access = Write

Service Priority = 3

Service Access = Write

Service Priority = 1

Service Access = Read
```

The `Object^` type replaces the variant data type. The following example defines a custom attribute that takes an array of `Object^` as parameters.

Attribute arguments must be compile-time constants; in most cases, they should be constant literals.

See typeid for information on how to return a value of System::Type from a custom attribute block.

```
// extending_metadata_e.cpp
// compile with: /clr /c
using namespace System;
[AttributeUsage(AttributeTargets::Class | AttributeTargets::Method)]
public ref class AnotherAttr : public Attribute {
public:
   AnotherAttr(array<Object^>^) {}
   array<Object^>^ var1;
};

// applying the attribute
[ AnotherAttr( gcnew array<Object ^> { 3.14159, "pi" }, var1 = gcnew array<Object ^> { "a", "b" } ) ]
public ref class SomeClass {};
```

The runtime requires that the public part of the custom attribute class must be serializable. When authoring custom attributes, named arguments of your custom attribute are limited to compile-time constants. (Think of it as a sequence of bits appended to your class layout in the metadata.)

```
// extending_metadata_f.cpp
// compile with: /clr /c
using namespace System;
ref struct abc {};

[AttributeUsage( AttributeTargets::All )]
ref struct A : Attribute {
   A( Type^ ) {}
   A( String ^ ) {}
   A( int ) {}
};

[A( abc::typeid )]
ref struct B {};
```

See also

Component Extensions for .NET and UWP

Attribute Parameter Types (C++/CLI and C++/CX)

5/13/2022 • 2 minutes to read • Edit Online

Values passed to attributes must be known to the compiler at compile time. Attribute parameters can be of the following types:

- `bool`
- `char`, `unsigned char`
- `short`, `unsigned short`
- `int`, `unsigned int`
- `long`, `unsigned long`
- `__int64`, **unsigned __int64**
- `float`, `double`
- `wchar_t`
- `char*` or `wchar_t*` or `System::String*`
- `System::Type ^`
- `System::Object ^`
- `enum`

Example: Attribute parameter types

Code

```
// attribute_parameter_types.cpp
// compile with: /clr /c
using namespace System;
ref struct AStruct {};

[AttributeUsage(AttributeTargets::ReturnValue)]
ref struct Attr : public Attribute {
   Attr(AStruct ^ i){}
   Attr(bool i){}
   Attr(){}
};

ref struct MyStruct {
   static AStruct ^ x = gcnew AStruct;
   [returnvalue:Attr(x)] int Test() { return 0; }   // C3104
   [returnvalue:Attr] int Test2() { return 0; }   // OK
   [returnvalue:Attr(true)] int Test3() { return 0; }   // OK
};
```

Example: Unnamed arguments precede named arguments

Description

When specifying attributes, all unnamed (positional) arguments must precede any named arguments.

Code

```
// extending_metadata_c.cpp
// compile with: /clr /c
using namespace System;
[AttributeUsage(AttributeTargets::Class)]
ref class MyAttr : public Attribute {
public:
   MyAttr() {}
   MyAttr(int i) {}
   property int Priority;
   property int Version;
};

[MyAttr]
ref class ClassA {};   // No arguments

[MyAttr(Priority = 1)]
ref class ClassB {};   // Named argument

[MyAttr(123)]
ref class ClassC {};   // Positional argument

[MyAttr(123, Version = 1)]
ref class ClassD {};   // Positional and named
```

Example: One-dimensional array attribute parameter

Description

Attribute parameters can be one-dimensional arrays of the previous types.

Code

```
// extending_metadata_d.cpp
// compile with: /clr /c
using namespace System;

[AttributeUsage(AttributeTargets::Class)]
public ref struct ABC : public Attribute {
   ABC(array<int>^){}
   array<double> ^ param;
};

[ABC( gcnew array<int> {1,2,3}, param = gcnew array<double>{2.71, 3.14})]
ref struct AStruct{};
```

See also

User-Defined Attributes

Attribute Targets (C++/CLI and C++/CX)

5/13/2022 • 3 minutes to read • Edit Online

Attribute usage specifiers let you specify attribute targets. Each attribute is defined to apply to certain language elements. For example, an attribute might be defined to apply only to classes and structs. The following list shows the possible syntactic elements on which a custom attribute can be used. Combinations of these values (using logical or) may be used.

To specify attribute target, to pass one or more AttributeTargets enumerators to AttributeUsageAttribute when defining the attribute.

The following is a list of the valid attribute targets:

- `All` (applies to all constructs)

  ```
  using namespace System;
  [AttributeUsage(AttributeTargets::All)]
  ref class Attr : public Attribute {};

  [assembly:Attr];
  ```

- `Assembly` (applies to an assembly as a whole)

  ```
  using namespace System;
  [AttributeUsage(AttributeTargets::Assembly)]
  ref class Attr : public Attribute {};

  [assembly:Attr];
  ```

- `Module` (applies to a module as a whole)

  ```
  using namespace System;
  [AttributeUsage(AttributeTargets::Module)]
  ref class Attr : public Attribute {};

  [module:Attr];
  ```

- `Class`

  ```
  using namespace System;
  [AttributeUsage(AttributeTargets::Class)]
  ref class Attr : public System::Attribute {};

  [Attr]   // same as [class:Attr]
  ref class MyClass {};
  ```

- `Struct`

```
using namespace System;
[AttributeUsage(AttributeTargets::Struct)]
ref class Attr : public Attribute {};

[Attr]   // same as [struct:Attr]
value struct MyStruct{};
```

- `enum`

```
using namespace System;
[AttributeUsage(AttributeTargets::Enum)]
ref class Attr : public Attribute {};

[Attr]   // same as [enum:Attr]
enum struct MyEnum{e, d};
```

- `Constructor`

```
using namespace System;
[AttributeUsage(AttributeTargets::Constructor)]
ref class Attr : public Attribute {};

ref struct MyStruct{
[Attr] MyStruct(){}   // same as [constructor:Attr]
};
```

- `Method`

```
using namespace System;
[AttributeUsage(AttributeTargets::Method)]
ref class Attr : public Attribute {};

ref struct MyStruct{
[Attr] void Test(){}   // same as [method:Attr]
};
```

- `Property`

```
using namespace System;
[AttributeUsage(AttributeTargets::Property)]
ref class Attr : public Attribute {};

ref struct MyStruct{
[Attr] property int Test;   // same as [property:Attr]
};
```

- `Field`

```
using namespace System;
[AttributeUsage(AttributeTargets::Field)]
ref class Attr : public Attribute {};

ref struct MyStruct{
[Attr] int Test;   // same as [field:Attr]
};
```

- `Event`

```
using namespace System;
[AttributeUsage(AttributeTargets::Event)]
ref class Attr : public Attribute {};

delegate void ClickEventHandler(int, double);

ref struct MyStruct{
[Attr] event ClickEventHandler^ OnClick;   // same as [event:Attr]
};
```

- `Interface`

```
using namespace System;
[AttributeUsage(AttributeTargets::Interface)]
ref class Attr : public Attribute {};

[Attr]   // same as [event:Attr]
interface struct MyStruct{};
```

- `Parameter`

```
using namespace System;
[AttributeUsage(AttributeTargets::Parameter)]
ref class Attr : public Attribute {};

ref struct MyStruct{
void Test([Attr] int i);
void Test2([parameter:Attr] int i);
};
```

- `Delegate`

```
using namespace System;
[AttributeUsage(AttributeTargets::Delegate)]
ref class Attr : public Attribute {};

[Attr] delegate void Test();
[delegate:Attr] delegate void Test2();
```

- `ReturnValue`

```
using namespace System;
[AttributeUsage(AttributeTargets::ReturnValue)]
ref class Attr : public Attribute {};

ref struct MyStruct {
// Note required specifier
[returnvalue:Attr] int Test() { return 0; }
};
```

Typically, an attribute directly precedes the language element to which it applies. In some cases, however, the position of an attribute is not sufficient to determine the attribute's intended target. Consider this example:

```
[Attr] int MyFn(double x)...
```

Syntactically, there is no way to tell if the attribute is intended to apply to the method or to the method's return value (in this case, it defaults to the method). In such cases, an attribute usage specifier may be used. For example, to make the attribute apply to the return value, use the `returnvalue` specifier, as follows:

```
[returnvalue:Attr] int MyFn(double x)... // applies to return value
```

Attribute usage specifiers are required in the following situations:

- To specify an assembly- or module-level attribute.
- To specify that an attribute applies to a method's return value, not the method:

  ```
  [method:Attr] int MyFn(double x)...     // Attr applies to method
  [returnvalue:Attr] int MyFn(double x)...// Attr applies to return value
  [Attr] int MyFn(double x)...            // default: method
  ```

- To specify that an attribute applies to a property's accessor, not the property:

  ```
  [method:MyAttr(123)] property int Property()
  [property:MyAttr(123)] property int Property()
  [MyAttr(123)] property int get_MyPropy() // default: property
  ```

- To specify that an attribute applies to an event's accessor, not the event:

  ```
  delegate void MyDel();
  ref struct X {
     [field:MyAttr(123)] event MyDel* MyEvent;   //field
     [event:MyAttr(123)] event MyDel* MyEvent;   //event
     [MyAttr(123)] event MyDel* MyEvent;   // default: event
  }
  ```

An attribute usage specifier applies only to the attribute that immediately follows it; that is,

```
[returnvalue:Attr1, Attr2]
```

is different from

```
[returnvalue:Attr1, returnvalue:Attr2]
```

Example

Description

This sample shows how to specify multiple targets.

Code

```
using namespace System;
[AttributeUsage(AttributeTargets::Class | AttributeTargets::Struct, AllowMultiple = true )]
ref struct Attr : public Attribute {
   Attr(bool i){}
   Attr(){}
};

[Attr]
ref class MyClass {};

[Attr]
[Attr(true)]
value struct MyStruct {};
```

See also

User-Defined Attributes

Extensions That Are Specific to C++/CLI

5/13/2022 • 2 minutes to read • Edit Online

The following language features apply only to C++/CLI:

__identifier (C++/CLI)

C-Style Casts with /clr (C++/CLI)

interior_ptr (C++/CLI)

pin_ptr (C++/CLI)

Type Forwarding (C++/CLI)

Variable Argument Lists (...) (C++/CLI)

See also

Component Extensions for .NET and UWP

__identifier (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

Enables the use of C++ keywords as identifiers.

All Platforms

Syntax

```
__identifier(C++_keyword)
```

Remarks

Use of the **__identifier** keyword for identifiers that are not keywords is permitted, but strongly discouraged as a matter of style.

Windows Runtime

Requirements

Compiler option: `/ZW`

Examples

Example

In the following example, a class named `template` is created in C# and distributed as a DLL. In the C++/CLI program that uses the `template` class, the `__identifier` keyword conceals the fact that `template` is a standard C++ keyword.

```
// identifier_template.cs
// compile with: /target:library
public class template {
   public void Run() { }
}
```

```
// keyword__identifier.cpp
// compile with: /ZW
#using <identifier_template.dll>
int main() {
   __identifier(template)^ pTemplate = ref new __identifier(template)();
   pTemplate->Run();
}
```

Common Language Runtime

Remarks

The **__identifier** keyword is valid with the `/clr` compiler option.

Requirements

Compiler option: `/clr`

Examples

In the following example, a class named `template` is created in C# and distributed as a DLL. In the C++/CLI program that uses the `template` class, the `__identifier` keyword conceals the fact that `template` is a standard C++ keyword.

```
// identifier_template.cs
// compile with: /target:library
public class template {
   public void Run() { }
}
```

```
// keyword__identifier.cpp
// compile with: /clr
#using <identifier_template.dll>

int main() {
   __identifier(template) ^pTemplate = gcnew __identifier(template)();
   pTemplate->Run();
}
```

See also

Component Extensions for .NET and UWP
Component Extensions for .NET and UWP

C-Style Casts with /clr (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

The following topic applies only to the Common Language Runtime.

When used with CLR types, the compiler attempts to map C-style cast to one of the casts listed below, in the following order:

1. const_cast
2. safe_cast
3. safe_cast plus const_cast
4. static_cast
5. static_cast plus const_cast

If none of the casts listed above is valid, and if the type of the expression and the target type are CLR reference types, C-style cast maps to a runtime-check (castclass MSIL instruction). Otherwise, a C-style cast is considered invalid and the compiler issues an error.

Remarks

A C-style cast is not recommended. When compiling with /clr (Common Language Runtime Compilation), use safe_cast.

The following sample shows a C-style cast that maps to a `const_cast`.

```
// cstyle_casts_1.cpp
// compile with: /clr
using namespace System;

ref struct R {};
int main() {
   const R^ constrefR = gcnew R();
   R^ nonconstR = (R^)(constrefR);
}
```

The following sample shows a C-style cast that maps to a **safe_cast**.

```
// cstyle_casts_2.cpp
// compile with: /clr
using namespace System;
int main() {
   Object ^ o = "hello";
   String ^ s = (String^)o;
}
```

The following sample shows a C-style cast that maps to a **safe_cast** plus `const_cast`.

```
// cstyle_casts_3.cpp
// compile with: /clr
using namespace System;

ref struct R {};
ref struct R2 : public R {};

int main() {
   const R^ constR2 = gcnew R2();
   try {
   R2^ b2DR = (R2^)(constR2);
   }
   catch(InvalidCastException^ e) {
      System::Console::WriteLine("Invalid Exception");
   }
}
```

The following sample shows a C-style cast that maps to a `static_cast`.

```
// cstyle_casts_4.cpp
// compile with: /clr
using namespace System;

struct N1 {};
struct N2 {
   operator N1() {
      return N1();
   }
};

int main() {
   N2 n2;
   N1 n1 ;
   n1 = (N1)n2;
}
```

The following sample shows a C-style cast that maps to a `static_cast` plus `const_cast`.

```
// cstyle_casts_5.cpp
// compile with: /clr
using namespace System;
struct N1 {};

struct N2 {
   operator const N1*() {
      static const N1 n1;
      return &n1;
   }
};

int main() {
   N2 n2;
   N1* n1 = (N1*)(const N1*)n2;   // const_cast + static_cast
}
```

The following sample shows a C-style cast that maps to a run-time check.

```
// cstyle_casts_6.cpp
// compile with: /clr
using namespace System;

ref class R1 {};
ref class R2 {};

int main() {
   R1^ r  = gcnew R1();
   try {
      R2^ rr = ( R2^)(r);
   }
   catch(System::InvalidCastException^ e) {
      Console::WriteLine("Caught expected exception");
   }
}
```

The following sample shows an invalid C-style cast, which causes the compiler to issue an error.

```
// cstyle_casts_7.cpp
// compile with: /clr
using namespace System;
int main() {
   String^s = S"hello";
   int i = (int)s;   // C2440
}
```

Requirements

Compiler option: `/clr`

See also

Component Extensions for .NET and UWP

interior_ptr (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

An *interior pointer* declares a pointer to inside a reference type, but not to the object itself. An interior pointer can point to a reference handle, value type, boxed type handle, member of a managed type, or to an element of a managed array.

All Runtimes

(There are no remarks for this language feature that apply to all runtimes.)

Windows Runtime

(There are no remarks for this language feature that apply to only the Windows Runtime.)

Requirements

Compiler option: `/ZW`

Common Language Runtime

The following syntax example demonstrates an interior pointer.

Syntax

```
cli::interior_ptr<cv_qualifier type> var = &initializer;
```

Parameters

cv_qualifier
`const` or `volatile` qualifiers.

type
The type of *initializer*.

var
The name of the **interior_ptr** variable.

initializer
A member of a reference type, element of a managed array, or any other object that you can assign to a native pointer.

Remarks

A native pointer is not able to track an item as its location changes on the managed heap, which results from the garbage collector moving instances of an object. In order for a pointer to correctly refer to the instance, the runtime needs to update the pointer to the newly positioned object.

An **interior_ptr** represents a superset of the functionality of a native pointer. Therefore, anything that can be assigned to a native pointer can also be assigned to an **interior_ptr**. An interior pointer is permitted to perform the same set of operations as native pointers, including comparison and pointer arithmetic.

An interior pointer can only be declared on the stack. An interior pointer cannot be declared as a member of a class.

Since interior pointers exist only on the stack, taking the address of an interior pointer yields an unmanaged pointer.

interior_ptr has an implicit conversion to `bool`, which allows for its use in conditional statements.

For information on how to declare an interior pointer that points into an object that cannot be moved on the garbage-collected heap, see pin_ptr.

interior_ptr is in the cli namespace. See Platform, default, and cli Namespaces for more information.

For more information on interior pointers, see

- How to: Declare and Use Interior Pointers and Managed Arrays (C++/CLI)
- How to: Declare Value Types with the interior_ptr Keyword (C++/CLI)
- How to: Overload Functions with Interior Pointers and Native Pointers (C++/CLI)
- How to: Declare Interior Pointers with the const Keyword (C++/CLI)

Requirements

Compiler option: `/clr`

Examples

The following sample shows how to declare and use an interior pointer into a reference type.

```
// interior_ptr.cpp
// compile with: /clr
using namespace System;

ref class MyClass {
public:
   int data;
};

int main() {
   MyClass ^ h_MyClass = gcnew MyClass;
   h_MyClass->data = 1;
   Console::WriteLine(h_MyClass->data);

   interior_ptr<int> p = &(h_MyClass->data);
   *p = 2;
   Console::WriteLine(h_MyClass->data);

   // alternatively
   interior_ptr<MyClass ^> p2 = &h_MyClass;
   (*p2)->data = 3;
   Console::WriteLine((*p2)->data);
}
```

```
1
2
3
```

See also

Component Extensions for .NET and UWP

How to: Declare and Use Interior Pointers and Managed Arrays (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

The following C++/CLI sample shows how you can declare and use an interior pointer to an array.

IMPORTANT

This language feature is supported by the `/clr` compiler option, but not by the `/ZW` compiler option.

Example

Code

```
// interior_ptr_arrays.cpp
// compile with: /clr
#define SIZE 10

int main() {
   // declare the array
   array<int>^ arr = gcnew array<int>(SIZE);

   // initialize the array
   for (int i = 0 ; i < SIZE ; i++)
      arr[i] = i + 1;

   // create an interior pointer into the array
   interior_ptr<int> ipi = &arr[0];

   System::Console::WriteLine("1st element in arr holds: {0}", arr[0]);
   System::Console::WriteLine("ipi points to memory address whose value is: {0}", *ipi);

   ipi++;
   System::Console::WriteLine("after incrementing ipi, it points to memory address whose value is: {0}",
*ipi);
}
```

```
1st element in arr holds: 1
ipi points to memory address whose value is: 1
after incrementing ipi, it points to memory address whose value is: 2
```

See also

interior_ptr (C++/CLI)

How to: Declare Value Types with the interior_ptr Keyword (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

An **interior_ptr** can be used with a value type.

IMPORTANT

This language feature is supported by the `/clr` compiler option, but not by the `/ZW` compiler option.

Example: interior_ptr with value type

Description

The following C++/CLI sample shows how to use an **interior_ptr** with a value type.

Code

```
// interior_ptr_value_types.cpp
// compile with: /clr
value struct V {
   V(int i) : data(i){}
   int data;
};

int main() {
   V v(1);
   System::Console::WriteLine(v.data);

   // pointing to a value type
   interior_ptr<V> pv = &v;
   pv->data = 2;

   System::Console::WriteLine(v.data);
   System::Console::WriteLine(pv->data);

   // pointing into a value type
   interior_ptr<int> pi = &v.data;
   *pi = 3;
   System::Console::WriteLine(*pi);
   System::Console::WriteLine(v.data);
   System::Console::WriteLine(pv->data);
}
```

```
1
2
2
3
3
3
```

Example: this pointer

Description

In a value type, the `this` pointer evaluates to an interior_ptr.

In the body of a non-static member-function of a value type `V`, `this` is an expression of type `interior_ptr<V>` whose value is the address of the object for which the function is called.

Code

```
// interior_ptr_value_types_this.cpp
// compile with: /clr /LD
value struct V {
   int data;
   void f() {
      interior_ptr<V> pv1 = this;
      // V* pv2 = this;   error
   }
};
```

Example: address-of operator

Description

The following sample shows how to use the address-of operator with static members.

The address of a static Visual C++ type member yields a native pointer. The address of a static value type member is a managed pointer because value type member is allocated on the runtime heap and can be moved by the garbage collector.

Code

```
// interior_ptr_value_static.cpp
// compile with: /clr
using namespace System;
value struct V { int i; };

ref struct G {
   static V v = {22};
   static int i = 23;
   static String^ pS = "hello";
};

int main() {
   interior_ptr<int> p1 = &G::v.i;
   Console::WriteLine(*p1);

   interior_ptr<int> p2 = &G::i;
   Console::WriteLine(*p2);

   interior_ptr<String^> p3 = &G::pS;
   Console::WriteLine(*p3);
}
```

```
22
23
hello
```

See also

interior_ptr (C++/CLI)

How to: Overload Functions with Interior Pointers and Native Pointers (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

Functions can be overloaded depending on whether the parameter type is an interior pointer or a native pointer.

IMPORTANT

This language feature is supported by the `/clr` compiler option, but not by the `/ZW` compiler option.

Example

Code

```
// interior_ptr_overload.cpp
// compile with: /clr
using namespace System;

// C++ class
struct S {
   int i;
};

// managed class
ref struct G {
   int i;
};

// can update unmanaged storage
void f( int* pi ) {
   *pi = 10;
   Console::WriteLine("in f( int* pi )");
}

// can update managed storage
void f( interior_ptr<int> pi ) {
   *pi = 10;
   Console::WriteLine("in f( interior_ptr<int> pi )");
}

int main() {
   S *pS = new S;   // C++ heap
   G ^pG = gcnew G;   // common language runtime heap
   f( &pS->i );
   f( &pG->i );
};
```

```
in f( int* pi )
in f( interior_ptr<int> pi )
```

See also

interior_ptr (C++/CLI)

How to: Declare Interior Pointers with the const Keyword (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

The following sample shows how to use `const` in the declaration of an interior pointer.

IMPORTANT

This language feature is supported by the `/clr` compiler option, but not by the `/ZW` compiler option.

Example

```
// interior_ptr_const.cpp
// compile with: /clr
using namespace System;
value struct V {
   int i;
};

ref struct G {
   V v;
   String ^ msg;
};

interior_ptr<int> f( interior_ptr<V> pv ) {
   return &(pv->i);
}

int main() {
   int n = -1;
   int o = -1;
   interior_ptr<int> pn1 = &n;
   *pn1 = 50;

   V v;
   v.i = 101;
   V * npV = &v;   // ok: &v yields a pointer to the native heap

   interior_ptr<int> pn2 = &n;
   interior_ptr<V> pV = &(v);
   pn2 = f(pV);
   *pn2 = 50;

   G ^pG = gcnew G;
   pV = &(pG->v);   // ok: pV is an interior pointer

   interior_ptr<int const> pn3 = &n;
   // *pn3 = 5;   error because pn3 cannot be dereferenced and changed
   pn3 = &o;   // OK, can change the memory location

   interior_ptr<int> const pn4 = &n;
   *pn4 = 5;   // OK because you can dereference and change pn4
   // pn4 = &o;   error cannot change the memory location

   const interior_ptr<const int> pn5 = &n;
   // *pn5 = 5;   error cannot dereference and change pn5
   // pn5 = &o;   error cannot change the memory location

   const G ^ h_G = gcnew G;   // object is const, cannot modify any members of h_G or call any non-const
methods
   // h_G->msg = "test";   error h_G is const
   interior_ptr<String^ const> int_ptr_G = &(h_G->msg);

   G ^ const h_G2 = gcnew G;   // interior pointers to this obejct cannot be dereferenced and changed
   h_G2->msg = "test";
   interior_ptr<String^ const> int_ptr_G2 = &(h_G->msg);
};
```

See also

interior_ptr (C++/CLI)

pin_ptr (C++/CLI)

5/13/2022 • 4 minutes to read • Edit Online

Declares a *pinning pointer*, which is used only with the common language runtime.

All Runtimes

(There are no remarks for this language feature that apply to all runtimes.)

Windows Runtime

(This language feature is not supported in the Windows Runtime.)

Common Language Runtime

A *pinning pointer* is an interior pointer that prevents the object pointed to from moving on the garbage-collected heap. That is, the value of a pinning pointer is not changed by the common language runtime. This is required when you pass the address of a managed class to an unmanaged function so that the address will not change unexpectedly during resolution of the unmanaged function call.

Syntax

```
[cli::]pin_ptr<cv_qualifiertype>var = &initializer;
```

Parameters

cv_qualifier
`const` or `volatile` qualifiers. By default, a pinning pointer is `volatile`. It is redundant but not an error to declare a pinning pointer `volatile`.

type
The type of *initializer*.

var
The name of the **pin_ptr** variable.

initializer
A member of a reference type, element of a managed array, or any other object that you can assign to a native pointer.

Remarks

A **pin_ptr** represents a superset of the functionality of a native pointer. Therefore, anything that can be assigned to a native pointer can also be assigned to a **pin_ptr**. An interior pointer is permitted to perform the same set of operations as native pointers, including comparison and pointer arithmetic.

An object or sub-object of a managed class can be pinned, in which case the common language runtime will not move it during garbage collection. The principal use of this is to pass a pointer to managed data as an actual parameter of an unmanaged function call. During a collection cycle, the runtime will inspect the metadata created for the pinning pointer and will not move the item it points to.

Pinning an object also pins its value fields; that is, fields of primitive or value type. However, fields declared by tracking handle (`%`) are not pinned.

Pinning a sub-object defined in a managed object has the effect of pinning the whole object.

If the pinning pointer is reassigned to point to a new value, the previous instance pointed to is no longer considered pinned.

An object is pinned only while a **pin_ptr** points to it. The object is no longer pinned when its pinning pointer goes out of scope, or is set to nullptr. After the **pin_ptr** goes out of scope, the object that was pinned can be moved in the heap by the garbage collector. Any native pointers that still point to the object will not be updated, and de-referencing one of them could raise an unrecoverable exception.

If no pinning pointers point to the object (all pinning pointers went out of scope, were reassigned to point to other objects, or were assigned nullptr), the object is guaranteed not to be pinned.

A pinning pointer can point to a reference handle, value type or boxed type handle, member of a managed type, or an element of a managed array. It cannot point to a reference type.

Taking the address of a **pin_ptr** that points to a native object causes undefined behavior.

Pinning pointers can only be declared as non-static local variables on the stack.

Pinning pointers cannot be used as:

- function parameters
- the return type of a function
- a member of a class
- the target type of a cast.

pin_ptr is in the `cli` namespace. For more information, see Platform, default, and cli Namespaces.

For more information about interior pointers, see interior_ptr (C++/CLI).

For more information about pinning pointers, see How to: Pin Pointers and Arrays and How to: Declare Pinning Pointers and Value Types.

Requirements

Compiler option: `/clr`

Examples

The following example uses **pin_ptr** to constrain the position of the first element of an array.

```
// pin_ptr_1.cpp
// compile with: /clr
using namespace System;
#define SIZE 10

#pragma unmanaged
// native function that initializes an array
void native_function(int* p) {
   for(int i = 0 ; i < 10 ; i++)
    p[i] = i;
}
#pragma managed

public ref class A {
private:
   array<int>^ arr;   // CLR integer array

public:
   A() {
      arr = gcnew array<int>(SIZE);
   }

   void load() {
   pin_ptr<int> p = &arr[0];   // pin pointer to first element in arr
   int* np = p;   // pointer to the first element in arr
   native_function(np);   // pass pointer to native function
   }

   int sum() {
      int total = 0;
      for (int i = 0 ; i < SIZE ; i++)
         total += arr[i];
      return total;
   }
};

int main() {
   A^ a = gcnew A;
   a->load();   // initialize managed array using the native function
   Console::WriteLine(a->sum());
}
```

```
45
```

The following example shows that an interior pointer can be converted to a pinning pointer, and that the return type of the address-of operator (`&`) is an interior pointer when the operand is on the managed heap.

```
// pin_ptr_2.cpp
// compile with: /clr
using namespace System;

ref struct G {
   G() : i(1) {}
   int i;
};

ref struct H {
   H() : j(2) {}
   int j;
};

int main() {
   G ^ g = gcnew G;   // g is a whole reference object pointer
   H ^ h = gcnew H;

   interior_ptr<int> l = &(g->i);   // l is interior pointer

   pin_ptr<int> k = &(h->j);   // k is a pinning interior pointer

   k = l;   // ok
   Console::WriteLine(*k);
};
```

```
1
```

The following example shows that a pinning pointer can be cast to another type.

```
// pin_ptr_3.cpp
// compile with: /clr
using namespace System;

ref class ManagedType {
public:
   int i;
};

int main() {
   ManagedType ^mt = gcnew ManagedType;
   pin_ptr<int> pt = &mt->i;
   *pt = 8;
   Console::WriteLine(mt->i);

   char *pc = ( char* ) pt;
   *pc = 255;
   Console::WriteLine(mt->i);
}
```

```
8
255
```

How to: Pin Pointers and Arrays

5/13/2022 • 2 minutes to read • Edit Online

Pinning a sub-object defined in a managed object has the effect of pinning the entire object. For example, if any element of an array is pinned, then the whole array is also pinned. There are no extensions to the language for declaring a pinned array. To pin an array, declare a pinning pointer to its element type, and pin one of its elements.

Example

Code

```
// pin_ptr_array.cpp
// compile with: /clr
#include <stdio.h>
using namespace System;

int main() {
   array<Byte>^ arr = gcnew array<Byte>(4);
   arr[0] = 'C';
   arr[1] = '+';
   arr[2] = '+';
   arr[3] = '\0';
   pin_ptr<Byte> p = &arr[1];   // entire array is now pinned
   unsigned char * cp = p;

   printf_s("%s\n", cp); // bytes pointed at by cp
                         // will not move during call
}
```

```
++
```

See also

pin_ptr (C++/CLI)

How to: Declare Pinning Pointers and Value Types

5/13/2022 • 2 minutes to read • Edit Online

A value type can be implicitly boxed. You can then declare a pinning pointer to the value type object itself and use a **pin_ptr** to the boxed value type.

Example

Code

```
// pin_ptr_value.cpp
// compile with: /clr
value struct V {
   int i;
};

int main() {
   V ^ v = gcnew V;   // imnplicit boxing
   v->i=8;
   System::Console::WriteLine(v->i);
   pin_ptr<V> mv = &*v;
   mv->i = 7;
   System::Console::WriteLine(v->i);
   System::Console::WriteLine(mv->i);
}
```

```
8
7
7
```

See also

pin_ptr (C++/CLI)

Type Forwarding (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

Type forwarding allows you to move a type from one assembly (assembly A) into another assembly (assembly B), such that, it is not necessary to recompile clients that consume assembly A.

Windows Runtime

This feature is not supported in the Windows Runtime.

Common Language Runtime

The following code example demonstrates how to use type forwarding.

Syntax

```
#using "new.dll"
[assembly:TypeForwardedTo(type::typeid)];
```

Parameters

new
The assembly into which you are moving the type definition.

type
The type whose definition you are moving into another assembly.

Remarks

After a component (assembly) ships and is being used by client applications, you can use type forwarding to move a type from the component (assembly) into another assembly, ship the updated component (and any additional assemblies required), and the client applications will still work without being recompiled.

Type forwarding only works for components referenced by existing applications. When you rebuild an application, there must be the appropriate assembly references for any types used in the application.

When forwarding a type (Type A) from an assembly, you must add the `TypeForwardedTo` attribute for that type, as well as an assembly reference. The assembly that you reference must contain one of the following:

- The definition for Type A.
- A `TypeForwardedTo` attribute for Type A, as well as an assembly reference.

Examples of types that can be forwarded include:

- ref classes
- value classes
- enums
- interfaces

You cannot forward the following types:

- Generic types

- Native types
- Nested types (if you want to forward a nested type, you should forward the enclosing type)

You can forward a type to an assembly authored in any language targeting the common language runtime.

So, if a source code file that is used to build assembly A.dll contains a type definition (`ref class MyClass`), and you wanted to move that type definition to assembly B.dll, you would:

1. Move the `MyClass` type definition to a source code file used to build B.dll.
2. Build assembly B.dll
3. Delete the `MyClass` type definition from the source code used to build A.dll, and replace it with the following:

   ```
   #using "B.dll"
   [assembly:TypeForwardedTo(MyClass::typeid)];
   ```

4. Build assembly A.dll.
5. Use A.dll without recompiling client applications.

Requirements

Compiler option: `/clr`

Variable Argument Lists (...) (C++/CLI)

5/13/2022 • 2 minutes to read • Edit Online

This example shows how you can use the `...` syntax in C++/CLI to implement functions that have a variable number of arguments.

NOTE

This topic pertains to C++/CLI. For information about using the `...` in ISO Standard C++, see Ellipsis and variadic templates and Ellipsis and default arguments in Postfix expressions.

The parameter that uses `...` must be the last parameter in the parameter list.

Example

Code

```
// mcppv2_paramarray.cpp
// compile with: /clr
using namespace System;
double average( ... array<Int32>^ arr ) {
   int i = arr->GetLength(0);
   double answer = 0.0;

   for (int j = 0 ; j < i ; j++)
      answer += arr[j];

   return answer / i;
}

int main() {
   Console::WriteLine("{0}", average( 1, 2, 3, 6 ));
}
```

```
3
```

Code Example

The following example shows how to call from C# a Visual C++ function that takes a variable number of arguments.

```
// mcppv2_paramarray2.cpp
// compile with: /clr:safe /LD
using namespace System;

public ref class C {
public:
   void f( ... array<String^>^ a ) {}
};
```

The function `f` can be called from C# or Visual Basic, for example, as though it were a function that can take a variable number of arguments.

In C#, an argument that is passed to a `ParamArray` parameter can be called by a variable number of arguments. The following code sample is in C#.

```
// mcppv2_paramarray3.cs
// compile with: /r:mcppv2_paramarray2.dll
// a C# program

public class X {
   public static void Main() {
      // Visual C# will generate a String array to match the
      // ParamArray attribute
      C myc = new C();
      myc.f("hello", "there", "world");
   }
}
```

A call to `f` in Visual C++ can pass an initialized array or a variable-length array.

```
// mcpp_paramarray4.cpp
// compile with: /clr
using namespace System;

public ref class C {
public:
   void f( ... array<String^>^ a ) {}
};

int main() {
   C ^ myc = gcnew C();
   myc->f("hello", "world", "!!!");
}
```

See also

Arrays