How to: Define and consume classes and structs (C++/CLI)

Article • 10/17/2022

This article shows how to define and consume user-defined reference types and value types in C++/CLI.

Object instantiation

Reference (ref) types can only be instantiated on the managed heap, not on the stack or on the native heap. Value types can be instantiated on the stack or the managed heap.

C++

```cpp
// mcppv2_ref_class2.cpp
// compile with: /clr
ref class MyClass {
public:
   int i;

   // nested class
   ref class MyClass2 {
   public:
      int i;
   };

   // nested interface
   interface struct MyInterface {
      void f();
   };
};

ref class MyClass2 : public MyClass::MyInterface {
public:
   virtual void f() {
      System::Console::WriteLine("test");
   }
};

public value struct MyStruct {
   void f() {
      System::Console::WriteLine("test");
   }
};

int main() {
   // instantiate ref type on garbage-collected heap
   MyClass ^ p_MyClass = gcnew MyClass;
   p_MyClass -> i = 4;

   // instantiate value type on garbage-collected heap
   MyStruct ^ p_MyStruct = gcnew MyStruct;
   p_MyStruct -> f();

   // instantiate value type on the stack
   MyStruct p_MyStruct2;
   p_MyStruct2.f();

   // instantiate nested ref type on garbage-collected heap
   MyClass::MyClass2 ^ p_MyClass2 = gcnew MyClass::MyClass2;
   p_MyClass2 -> i = 5;
}
```

Implicitly abstract classes

An *implicitly abstract class* can't be instantiated. A class is implicitly abstract when:

- the base type of the class is an interface, and
- the class doesn't implement all of the interface's member functions.

You may be unable to construct objects from a class that's derived from an interface. The reason might be that the class is implicitly abstract. For more information about abstract classes, see abstract.

The following code example demonstrates that the `MyClass` class can't be instantiated because function `MyClass::func2` isn't implemented. To enable the example to compile, uncomment `MyClass::func2`.

C++

```cpp
// mcppv2_ref_class5.cpp
// compile with: /clr
interface struct MyInterface {
   void func1();
   void func2();
};

ref class MyClass : public MyInterface {
public:
   void func1(){}
   // void func2(){}
};

int main() {
   MyClass ^ h_MyClass = gcnew MyClass;   // C2259
                                          // To resolve, uncomment
MyClass::func2.
}
```

Type visibility

You can control the visibility of common language runtime (CLR) types. When your assembly is referenced, you control whether types in the assembly are visible or not visible outside the assembly.

`public` indicates that a type is visible to any source file that contains a `#using` directive for the assembly that contains the type. `private` indicates that a type isn't visible to source files that contain a `#using` directive for the assembly that contains the type. However, private types are visible within the same assembly. By default, the visibility for a class is `private`.

By default before Visual Studio 2005, native types had public accessibility outside the assembly. Enable Compiler Warning (level 1) C4692 to help you see where private native types are used incorrectly. Use the make_public pragma to give public accessibility to a native type in a source code file that you can't modify.

For more information, see #using Directive.

The following sample shows how to declare types and specify their accessibility, and then access those types inside the assembly. If an assembly that has private types is referenced by using `#using`, only public types in the assembly are visible.

C++

```cpp
// type_visibility.cpp
// compile with: /clr
using namespace System;
// public type, visible inside and outside assembly
public ref struct Public_Class {
   void Test(){Console::WriteLine("in Public_Class");}
};

// private type, visible inside but not outside assembly
private ref struct Private_Class {
   void Test(){Console::WriteLine("in Private_Class");}
};

// default accessibility is private
ref class Private_Class_2 {
public:
   void Test(){Console::WriteLine("in Private_Class_2");}
};
```

```
int main() {
   Public_Class ^ a = gcnew Public_Class;
   a->Test();

   Private_Class ^ b = gcnew Private_Class;
   b->Test();

   Private_Class_2 ^ c = gcnew Private_Class_2;
   c->Test();
}
```

Output

```Output
in Public_Class
in Private_Class
in Private_Class_2
```

Now, let's rewrite the previous sample so that it's built as a DLL.

```C++
// type_visibility_2.cpp
// compile with: /clr /LD
using namespace System;
// public type, visible inside and outside the assembly
public ref struct Public_Class {
   void Test(){Console::WriteLine("in Public_Class");}
};

// private type, visible inside but not outside the assembly
private ref struct Private_Class {
   void Test(){Console::WriteLine("in Private_Class");}
};

// by default, accessibility is private
ref class Private_Class_2 {
public:
   void Test(){Console::WriteLine("in Private_Class_2");}
};
```

The next sample shows how to access types outside the assembly. In this sample, the client consumes the component that's built in the previous sample.

```C++
// type_visibility_3.cpp
// compile with: /clr
```

```
#using "type_visibility_2.dll"
int main() {
   Public_Class ^ a = gcnew Public_Class;
   a->Test();

   // private types not accessible outside the assembly
   // Private_Class ^ b = gcnew Private_Class;
   // Private_Class_2 ^ c = gcnew Private_Class_2;
}
```

Output

```
Output

in Public_Class
```

Member visibility

You can make access to a member of a public class from within the same assembly different than access to it from outside the assembly by using pairs of the access specifiers `public`, `protected`, and `private`

This table summarizes the effect of the various access specifiers:

Expand table

Specifier	Effect
`public`	Member is accessible inside and outside the assembly. For more information, see public.
`private`	Member is inaccessible, both inside and outside the assembly. For more information, see private.
`protected`	Member is accessible inside and outside the assembly, but only to derived types. For more information, see protected.
`internal`	Member is public inside the assembly but private outside the assembly. `internal` is a context-sensitive keyword. For more information, see Context-Sensitive Keywords.
`public protected` -or- `protected public`	Member is public inside the assembly but protected outside the assembly.
`private protected` -or- `protected private`	Member is protected inside the assembly but private outside the assembly.

The following sample shows a public type that has members that are declared using the different access specifiers. Then, it shows access to those members from inside the assembly.

C++

```cpp
// compile with: /clr
using namespace System;
// public type, visible inside and outside the assembly
public ref class Public_Class {
public:
   void Public_Function(){System::Console::WriteLine("in
Public_Function");}

private:
   void Private_Function(){System::Console::WriteLine("in
Private_Function");}

protected:
   void Protected_Function(){System::Console::WriteLine("in
Protected_Function");}

internal:
   void Internal_Function(){System::Console::WriteLine("in
Internal_Function");}

protected public:
   void Protected_Public_Function(){System::Console::WriteLine("in
Protected_Public_Function");}

public protected:
   void Public_Protected_Function(){System::Console::WriteLine("in
Public_Protected_Function");}

private protected:
   void Private_Protected_Function(){System::Console::WriteLine("in
Private_Protected_Function");}

protected private:
   void Protected_Private_Function(){System::Console::WriteLine("in
Protected_Private_Function");}
};

// a derived type, calls protected functions
ref struct MyClass : public Public_Class {
   void Test() {
      Console::WriteLine("=======================");
      Console::WriteLine("in function of derived class");
      Protected_Function();
      Protected_Private_Function();
      Private_Protected_Function();
      Console::WriteLine("exiting function of derived class");
      Console::WriteLine("=======================");
   }
};

int main() {
   Public_Class ^ a = gcnew Public_Class;
   MyClass ^ b = gcnew MyClass;
   a->Public_Function();
   a->Protected_Public_Function();
   a->Public_Protected_Function();

   // accessible inside but not outside the assembly
   a->Internal_Function();

   // call protected functions
   b->Test();

   // not accessible inside or outside the assembly
   // a->Private_Function();
}
```

Output

```
Output

in Public_Function
in Protected_Public_Function
in Public_Protected_Function
in Internal_Function
=======================
in function of derived class
in Protected_Function
in Protected_Private_Function
in Private_Protected_Function
exiting function of derived class
=======================
```

Now let's build the previous sample as a DLL.

```
C++

// compile with: /clr /LD
using namespace System;
// public type, visible inside and outside the assembly
public ref class Public_Class {
public:
   void Public_Function(){System::Console::WriteLine("in
Public_Function");}

private:
   void Private_Function(){System::Console::WriteLine("in
Private_Function");}
protected:
   void Protected_Function(){System::Console::WriteLine("in
Protected_Function");}

internal:
   void Internal_Function(){System::Console::WriteLine("in
Internal_Function");}

protected public:
   void Protected_Public_Function(){System::Console::WriteLine("in
Protected_Public_Function");}

public protected:
   void Public_Protected_Function(){System::Console::WriteLine("in
Public_Protected_Function");}

private protected:
   void Private_Protected_Function(){System::Console::WriteLine("in
Private_Protected_Function");}

protected private:
   void Protected_Private_Function(){System::Console::WriteLine("in
Protected_Private_Function");}
};

// a derived type, calls protected functions
ref struct MyClass : public Public_Class {
   void Test() {
      Console::WriteLine("=======================");
      Console::WriteLine("in function of derived class");
      Protected_Function();
      Protected_Private_Function();
      Private_Protected_Function();
      Console::WriteLine("exiting function of derived class");
      Console::WriteLine("=======================");
   }
};
```

The following sample consumes the component that's created in the previous sample. It shows how to access the members from outside the assembly.

```C++
// compile with: /clr
#using "type_member_visibility_2.dll"
using namespace System;
// a derived type, calls protected functions
ref struct MyClass : public Public_Class {
   void Test() {
      Console::WriteLine("=======================");
      Console::WriteLine("in function of derived class");
      Protected_Function();
      Protected_Public_Function();
      Public_Protected_Function();
      Console::WriteLine("exiting function of derived class");
      Console::WriteLine("=======================");
   }
};

int main() {
   Public_Class ^ a = gcnew Public_Class;
   MyClass ^ b = gcnew MyClass;
   a->Public_Function();

   // call protected functions
   b->Test();

   // can't be called outside the assembly
   // a->Private_Function();
   // a->Internal_Function();
   // a->Protected_Private_Function();
   // a->Private_Protected_Function();
}
```

Output

Output

```
in Public_Function
=======================
in function of derived class
in Protected_Function
in Protected_Public_Function
in Public_Protected_Function
exiting function of derived class
=======================
```

Public and private native classes

A native type can be referenced from a managed type. For example, a function in a managed type can take a parameter whose type is a native struct. If the managed type and function are public in an assembly, then the native type must also be public.

C++

```
// native type
public struct N {
   N(){}
   int i;
};
```

Next, create the source code file that consumes the native type:

```C++
// compile with: /clr /LD
#include "mcppv2_ref_class3.h"
// public managed type
public ref struct R {
   // public function that takes a native type
   void f(N nn) {}
};
```

Now, compile a client:

```C++
// compile with: /clr
#using "mcppv2_ref_class3.dll"

#include "mcppv2_ref_class3.h"

int main() {
   R ^r = gcnew R;
   N n;
   r->f(n);
}
```

Static constructors

A CLR type—for example, a class or struct—can have a static constructor that can be used to initialize static data members. A static constructor is called at most once, and is called before any static member of the type is accessed the first time.

An instance constructor always runs after a static constructor.

The compiler can't inline a call to a constructor if the class has a static constructor. The compiler can't inline a call to any member function if the class is a value type, has a static constructor, and doesn't have an instance constructor. The CLR may inline the call, but the compiler can't.

Define a static constructor as a private member function, because it's meant to be called only by the CLR.

For more information about static constructors, see How to: Define an Interface Static Constructor (C++/CLI) .

C++

```
// compile with: /clr
using namespace System;

ref class MyClass {
private:
   static int i = 0;

   static MyClass() {
      Console::WriteLine("in static constructor");
      i = 9;
   }

public:
   static void Test() {
      i++;
      Console::WriteLine(i);
   }
};

int main() {
   MyClass::Test();
   MyClass::Test();
}
```

Output

Output

```
in static constructor
10
11
```

Semantics of the `this` pointer

When you're using C++\CLI to define types, the `this` pointer in a reference type is of type *handle*. The `this` pointer in a value type is of type *interior pointer*.

These different semantics of the `this` pointer can cause unexpected behavior when a default indexer is called. The next example shows the correct way to access a default indexer in both a ref type and a value type.

For more information, see Handle to Object Operator (^) and interior_ptr (C++/CLI)

C++

```
// compile with: /clr
using namespace System;

ref struct A {
   property Double default[Double] {
      Double get(Double data) {
         return data*data;
      }
   }

   A() {
      // accessing default indexer
      Console::WriteLine("{0}", this[3.3]);
   }
};

value struct B {
   property Double default[Double] {
      Double get(Double data) {
         return data*data;
      }
   }
   void Test() {
      // accessing default indexer
      Console::WriteLine("{0}", this->default[3.3]);
   }
};

int main() {
   A ^ mya = gcnew A();
   B ^ myb = gcnew B();
   myb->Test();
}
```

Output

Output

```
10.89
10.89
```

Hide-by-signature functions

In standard C++, a function in a base class gets hidden by a function that has the same name in a derived class, even if the derived-class function doesn't have the same kind or number of parameters. It's known as *hide-by-name* semantics. In a reference type, a function in a base class only gets hidden by a function in a derived class if both the name and the parameter list are the same. It's known as *hide-by-signature* semantics.

A class is considered a hide-by-signature class when all of its functions are marked in the metadata as `hidebysig`. By default, all classes that are created under **`/clr`** have `hidebysig` functions. When a class has `hidebysig` functions, the compiler doesn't hide functions by name in any direct base classes, but if the compiler encounters a hide-by-name class in an inheritance chain, it continues that hide-by-name behavior.

Under hide-by-signature semantics, when a function is called on an object, the compiler identifies the most derived class that contains a function that could satisfy the function call. If there's only one function in the class that satisfies the call, the compiler calls that function. If there's more than one function in the class that could satisfy the call, the compiler uses overload resolution rules to determine which function to call. For more information about overload rules, see Function Overloading.

For a given function call, a function in a base class might have a signature that makes it a slightly better match than a function in a derived class. However, if the function was explicitly called on an object of the derived class, the function in the derived class is called.

Because the return value isn't considered part of a function's signature, a base-class function gets hidden if it has the same name and takes the same kind and number of arguments as a derived-class function, even if it differs in the type of the return value.

The following sample shows that a function in a base class isn't hidden by a function in a derived class.

C++

```cpp
// compile with: /clr
using namespace System;
ref struct Base {
   void Test() {
      Console::WriteLine("Base::Test");
   }
};

ref struct Derived : public Base {
   void Test(int i) {
      Console::WriteLine("Derived::Test");
   }
};

int main() {
   Derived ^ t = gcnew Derived;
   // Test() in the base class will not be hidden
   t->Test();
}
```

Output

```Output
Base::Test
```

The next sample shows that the Microsoft C++ compiler calls a function in the most derived class—even if a conversion is required to match one or more of the parameters—and not call a function in a base class that is a better match for the function call.

```C++
// compile with: /clr
using namespace System;
ref struct Base {
   void Test2(Single d) {
      Console::WriteLine("Base::Test2");
   }
};

ref struct Derived : public Base {
   void Test2(Double f) {
      Console::WriteLine("Derived::Test2");
   }
};

int main() {
   Derived ^ t = gcnew Derived;
   // Base::Test2 is a better match, but the compiler
   // calls a function in the derived class if possible
   t->Test2(3.14f);
}
```

Output

```Output
Derived::Test2
```

The following sample shows that it's possible to hide a function even if the base class has the same signature as the derived class.

```C++
// compile with: /clr
using namespace System;
ref struct Base {
   int Test4() {
      Console::WriteLine("Base::Test4");
      return 9;
   }
};

ref struct Derived : public Base {
   char Test4() {
      Console::WriteLine("Derived::Test4");
      return 'a';
   }
};

int main() {
   Derived ^ t = gcnew Derived;

   // Base::Test4 is hidden
   int i = t->Test4();
   Console::WriteLine(i);
}
```

Output

Output

```
Derived::Test4
97
```

Copy constructors

The C++ standard says that a copy constructor is called when an object is moved, such that an object is created and destroyed at the same address.

However, when a function that's compiled to MSIL calls a native function where a native class—or more than one—is passed by value and where the native class has a copy constructor or a destructor, no copy constructor is called and the object is destroyed at a different address than where it was created. This behavior could cause problems if the class has a pointer into itself, or if the code is tracking objects by address.

For more information, see /clr (Common Language Runtime Compilation).

The following sample demonstrates when a copy constructor isn't generated.

C++

```
// compile with: /clr
#include<stdio.h>

struct S {
   int i;
   static int n;

   S() : i(n++) {
      printf_s("S object %d being constructed, this=%p\n", i, this);
   }

   S(S const& rhs) : i(n++) {
      printf_s("S object %d being copy constructed from S object "
               "%d, this=%p\n", i, rhs.i, this);
   }

   ~S() {
      printf_s("S object %d being destroyed, this=%p\n", i, this);
   }
};

int S::n = 0;

#pragma managed(push,off)
void f(S s1, S s2) {
   printf_s("in function f\n");
}
#pragma managed(pop)

int main() {
   S s;
   S t;
   f(s,t);
}
```

Output

```
Output

S object 0 being constructed, this=0018F378
S object 1 being constructed, this=0018F37C
S object 2 being copy constructed from S object 1, this=0018F380
S object 3 being copy constructed from S object 0, this=0018F384
S object 4 being copy constructed from S object 2, this=0018F2E4
S object 2 being destroyed, this=0018F380
S object 5 being copy constructed from S object 3, this=0018F2E0
S object 3 being destroyed, this=0018F384
in function f
S object 5 being destroyed, this=0018F2E0
S object 4 being destroyed, this=0018F2E4
S object 1 being destroyed, this=0018F37C
S object 0 being destroyed, this=0018F378
```

Destructors and finalizers

Destructors in a reference type do a deterministic clean-up of resources. Finalizers clean up unmanaged resources, and can be called either deterministically by the destructor or nondeterministically by the garbage collector. For information about destructors in standard C++, see Destructors.

```C++
class classname {
   ~classname() {}   // destructor
   ! classname() {}   // finalizer
};
```

The CLR garbage collector deletes unused managed objects and releases their memory when they're no longer required. However, a type may use resources that the garbage collector doesn't know how to release. These resources are known as *unmanaged* resources (native file handles, for example). We recommend you release all unmanaged resources in the finalizer. The garbage collector releases managed resources nondeterministically, so it's not safe to refer to managed resources in a finalizer. That's because it's possible the garbage collector has already cleaned them up.

A Visual C++ finalizer isn't the same as the Finalize method. (CLR documentation uses finalizer and the Finalize method synonymously). The Finalize method is called by the garbage collector, which invokes each finalizer in a class inheritance chain. Unlike Visual C++ destructors, a derived-class finalizer call doesn't cause the compiler to invoke the finalizer in all base classes.

Because the Microsoft C++ compiler supports deterministic release of resources, don't try to implement the Dispose or Finalize methods. However, if you're familiar with these methods, here's how a Visual C++ finalizer and a destructor that calls the finalizer map to the Dispose pattern:

```C++
// Visual C++ code
ref class T {
   ~T() { this->!T(); }   // destructor calls finalizer
   !T() {}   // finalizer
};

// equivalent to the Dispose pattern
void Dispose(bool disposing) {
   if (disposing) {
      ~T();
   } else {
      !T();
```

```
    }
}
```

A managed type may also use managed resources that you'd prefer to release deterministically. You may not want the garbage collector to release an object nondeterministically at some point after the object is no longer required. The deterministic release of resources can significantly improve performance.

The Microsoft C++ compiler enables the definition of a destructor to deterministically clean up objects. Use the destructor to release all resources that you want to deterministically release. If a finalizer is present, call it from the destructor, to avoid code duplication.

```C++
// compile with: /clr /c
ref struct A {
   // destructor cleans up all resources
   ~A() {
      // clean up code to release managed resource
      // ...
      // to avoid code duplication,
      // call finalizer to release unmanaged resources
      this->!A();
   }

   // finalizer cleans up unmanaged resources
   // destructor or garbage collector will
   // clean up managed resources
   !A() {
      // clean up code to release unmanaged resources
      // ...
   }
};
```

If the code that consumes your type doesn't call the destructor, the garbage collector eventually releases all managed resources.

The presence of a destructor doesn't imply the presence of a finalizer. However, the presence of a finalizer implies that you must define a destructor and call the finalizer from that destructor. This call provides for the deterministic release of unmanaged resources.

Calling the destructor suppresses—by using SuppressFinalize—finalization of the object. If the destructor isn't called, your type's finalizer will eventually be called by the garbage collector.

You can improve performance by calling the destructor to deterministically clean up your object's resources, instead of letting the CLR nondeterministically finalize the object.

Code that's written in Visual C++ and compiled by using `/clr` runs a type's destructor if:

- An object that's created by using stack semantics goes out of scope. For more information, see C++ Stack Semantics for Reference Types.
- An exception is thrown during the object's construction.
- The object is a member in an object whose destructor is running.
- You call the delete operator on a handle (Handle to Object Operator (^)).
- You explicitly call the destructor.

If a client that's written in another language consumes your type, the destructor gets called as follows:

- On a call to Dispose.
- On a call to `Dispose(void)` on the type.
- If the type goes out of scope in a C# `using` statement.

If you're not using stack semantics for reference types and create an object of a reference type on the managed heap, use try-finally syntax to ensure that an exception doesn't prevent the destructor from running.

```C++
// compile with: /clr
ref struct A {
   ~A() {}
};

int main() {
   A ^ MyA = gcnew A;
   try {
      // use MyA
   }
   finally {
      delete MyA;
   }
}
```

If your type has a destructor, the compiler generates a `Dispose` method that implements IDisposable. If a type that's written in Visual C++ and has a destructor that's consumed from another language, calling `IDisposable::Dispose` on that type causes the type's destructor to be called. When the type is consumed from a Visual C++ client, you can't directly call `Dispose`; instead, call the destructor by using the **`delete`** operator.

If your type has a finalizer, the compiler generates a `Finalize(void)` method that overrides Finalize.

If a type has either a finalizer or a destructor, the compiler generates a `Dispose(bool)` method, according to the design pattern. (For information, see Dispose Pattern). You can't explicitly author or call `Dispose(bool)` in Visual C++.

If a type has a base class that conforms to the design pattern, the destructors for all base classes are called when the destructor for the derived class is called. (If your type is written in Visual C++, the compiler ensures that your types implement this pattern.) In other words, the destructor of a reference class chains to its bases and members as specified by the C++ standard. First, the class's destructor is run. Then, the destructors for its members get run in the reverse of the order in which they were constructed. Finally, the destructors for its base classes get run in the reverse of the order in which they were constructed.

Destructors and finalizers aren't allowed inside value types or interfaces.

A finalizer can only be defined or declared in a reference type. Like a constructor and destructor, a finalizer has no return type.

After an object's finalizer runs, finalizers in any base classes are also called, beginning with the least derived type. Finalizers for data members aren't automatically chained to by a class's finalizer.

If a finalizer deletes a native pointer in a managed type, you must ensure that references to or through the native pointer aren't prematurely collected. Call the destructor on the managed type instead of using KeepAlive.

At compile time, you can detect whether a type has a finalizer or a destructor. For more information, see Compiler Support for Type Traits.

The next sample shows two types: one that has unmanaged resources, and one that has managed resources that get released deterministically.

```
C++
// compile with: /clr
#include <vcclr.h>
#include <stdio.h>
using namespace System;
using namespace System::IO;

ref class SystemFileWriter {
   FileStream ^ file;
   array<Byte> ^ arr;
   int bufLen;

public:
   SystemFileWriter(String ^ name) : file(File::Open(name,
FileMode::Append)),
                                     arr(gcnew array<Byte>(1024)) {}

   void Flush() {
      file->Write(arr, 0, bufLen);
      bufLen = 0;
   }

   ~SystemFileWriter() {
      Flush();
      delete file;
   }
};

ref class CRTFileWriter {
   FILE * file;
   array<Byte> ^ arr;
   int bufLen;

   static FILE * getFile(String ^ n) {
      pin_ptr<const wchar_t> name = PtrToStringChars(n);
      FILE * ret = 0;
      _wfopen_s(&ret, name, L"ab");
      return ret;
   }

public:
   CRTFileWriter(String ^ name) : file(getFile(name)), arr(gcnew ar-
ray<Byte>(1024) ) {}

   void Flush() {
      pin_ptr<Byte> buf = &arr[0];
      fwrite(buf, 1, bufLen, file);
      bufLen = 0;
   }

   ~CRTFileWriter() {
      this->!CRTFileWriter();
   }

   !CRTFileWriter() {
      Flush();
      fclose(file);
   }
};

int main() {
   SystemFileWriter w("systest.txt");
   CRTFileWriter ^ w2 = gcnew CRTFileWriter("crttest.txt");
}
```

See also

Classes and structs

Feedback