__clrcall

Article • 08/03/2021

Specifies that a function can only be called from managed code. Use **__clrcall** for all virtual functions that will only be called from managed code. However this calling convention cannot be used for functions that will be called from native code. The **__clrcall** modifier is Microsoft-specific.

Use **__clrcall** to improve performance when calling from a managed function to a virtual managed function or from managed function to managed function through pointer.

Entry points are separate, compiler-generated functions. If a function has both native and managed entry points, one of them will be the actual function with the function implementation. The other function will be a separate function (a thunk) that calls into the actual function and lets the common language runtime perform PInvoke. When marking a function as **__clrcall**, you indicate the function implementation must be MSIL and that the native entry point function will not be generated.

When taking the address of a native function if **__clrcall** is not specified, the compiler uses the native entry point. **__clrcall** indicates that the function is managed and there is no need to go through the transition from managed to native. In that case the compiler uses the managed entry point.

When `/clr` (not `/clr:pure` or `/clr:safe`) is used and **__clrcall** is not used, taking the address of a function always returns the address of the native entry point function. When **__clrcall** is used, the native entry point function is not created, so you get the address of the managed function, not an entry point thunk function. For more information, see Double Thunking. The **/clr:pure** and **/clr:safe** compiler options are deprecated in Visual Studio 2015 and unsupported in Visual Studio 2017.

/clr (Common Language Runtime Compilation) implies that all functions and function pointers are **__clrcall** and the compiler will not permit a function inside the compiland to be marked anything other than **__clrcall**. When **/clr:pure** is used, **__clrcall** can only be specified on function pointers and external declarations.

You can directly call **__clrcall** functions from existing C++ code that was compiled by using **/clr** as long as that function has an MSIL implementation. **__clrcall** functions cannot be called directly from functions that have inline asm and call CPU-specific intrinsics, for example, even if those functions are compiled with `/clr`.

__clrcall function pointers are only meant to be used in the application domain in which they were created. Instead of passing **__clrcall** function pointers across application domains, use CrossAppDomainDelegate. For more information, see Application Domains and Visual C++.

Examples

Note that when a function is declared with **__clrcall**, code will be generated when needed; for example, when function is called.

C++

```cpp
// clrcall2.cpp
// compile with: /clr
using namespace System;
int __clrcall Func1() {
   Console::WriteLine("in Func1");
   return 0;
}
```

```
// Func1 hasn't been used at this point (code has not been generated),
// so runtime returns the adddress of a stub to the function
int (__clrcall *pf)() = &Func1;

// code calls the function, code generated at difference address
int i = pf();   // comment this line and comparison will pass

int main() {
   if (&Func1 == pf)
      Console::WriteLine("&Func1 == pf, comparison succeeds");
   else
      Console::WriteLine("&Func1 != pf, comparison fails");

   // even though comparison fails, stub and function call are correct
   pf();
   Func1();
}
```

Output

```
in Func1
&Func1 != pf, comparison fails
in Func1
in Func1
```

The following sample shows that you can define a function pointer, such that, you declare that the function pointer will only be invoked from managed code. This allows the compiler to directly call the managed function and avoid the native entry point (double thunk issue).

C++

```
// clrcall3.cpp
// compile with: /clr
void Test() {
   System::Console::WriteLine("in Test");
}

int main() {
   void (*pTest)() = &Test;
   (*pTest)();

   void (__clrcall *pTest2)() = &Test;
   (*pTest2)();
}
```

See also

Argument Passing and Naming Conventions
Keywords

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A