Why is my C++/CX ref class forbidden from having public methods that are templates or mention, say, std::vector?

devblogs.microsoft.com/oldnewthing/20211119-00

November 19, 2021

Raymond Chen

A customer had a class implement in C++/CX[1] and they tried to add a public method:

```
ref class MyClass
{
public:
  // 1
  std::vector<int> GetValues();

  // 2
  void SetValues(std::vector<int> values);

  // 3
  template<typename T> T GetValue();
};
```

But this generated errors:

```
// 1
error C3986: 'SetValue': signature of public member contains native type
'std::vector<int,std::allocator<_Ty>>'

// 2
error C3986: 'GetValues': signature of public member contains native type
'std::vector<int,std::allocator<_Ty>>'

// 3
error C2900: 'T MyClass::GetValue(void)': member function templates in WinRT classes
must be 'private', 'internal' or 'protected private'
```

That last error message is the big clue.

The C++/CX nonstandard language extension introduces the `ref` keyword which is used to mark a class as participating in the Windows Runtime. These classes are automatically reference-counted, and you access instances of them as if they were pointers, but using `^` instead of `*`.

One of the things that happens when you use this extension is that there are new member access control keywords, and existing keywords change their meanings. C++/CX uses these member access control keywords to control member access both for C++ code inside the same module, as well for the Windows Runtime metadata that allows the classes to be used from other languages that support the Windows Runtime, like C# and JavaScript.

Keyword	In metadata	In C++
`public`	public	public
`public protected` `protected public`	protected	public
`protected`	protected	protected
`private public` `public private` `internal`	(none)	public
`private protected` `protected private`	(none)	protected
`private`	(none)	private

The access control keyword `public private` is deprecated, and since it is part of the already-deprecated C++/CX extension, that makes it double-deprecated.

If you use an access control keyword that puts the member into Windows Runtime metadata, then that member must conform to Windows Runtime rules. One of those rules is that the types used in the method signature must be expressible in the Windows Runtime. This means that you can use `public enum class`, `value struct`, or `ref class`, as well as a handful of primitive types like integers, floating point values, and `Platform::String^`.

The `std::vector` is not one of the allowed types in the Windows Runtime. After all, how would C# or JavaScript access a `std::vector`? That thing is C++-only and has no cross-language ABI. Indeed, it doesn't even have a consistent ABI within the C++ language: different compilers, different versions of the same compiler, or even different compiler options within the same version of the same compiler, can have different ABI interfaces to `std::vector`.

If you intend your Windows Runtime member to be accessible outside your module, then you need to express the member in terms that can be represented in the Windows Runtime.

On the other hand, if you just want the member to be accessible to other part of your module, you can switch to one of the member access keywords that excludes the member from Windows Runtime metadata. That frees you from the restriction of having to conform to

Windows Runtime rules.

[1] The nonstandard C++/CX extension is no longer the recommended mechanism for using the Windows Runtime from C++. For one thing, the C++/CX extension is supported only in C++14 and C++17 modes. You won't be able to use it with C++20 or later, so your C++/CX code won't be able to take advantage of any new language features like concepts. (You can get coroutine support by adding the `/await` switch.) I encourage you to stop using C++/CX and switch to C++/WinRT.

Raymond Chen

Follow