Can you dllexport/dllimport an inline function?

devblogs.microsoft.com/oldnewthing/20140109-00

January 9, 2014

Raymond Chen

The MSDN documentation on the subject of Defining Inline C++ Functions with dllexport and dllimport was written with compiler-colored glasses. The statements are perfectly true, but they use terminology that only compiler-writers understand.

The short version is that all modules which share an inline function are considered to be part of the same program, so all of the C++ rules regarding inline functions in programs need to be followed.

Let's look at the paragraphs one at a time and translate them into English.

> You can define as inline a function with the **dllexport** attribute. In this case, the function is always instantiated and exported, whether or not any module in the program references the function. The function is presumed to be imported by another program.

Okay, first of all, what is *instantiation*?

In this context, the term *instantiation* when applied to an inline function means "The code is generated (*instantiated*) for the function as if it had not been marked inline."

For the purpose of discussion, let's say that you have a function written as

```
__declspec(dllexport)
inline int times3(int i) { return i * 3; }
```

Suppose that you compile this into a DLL, and that DLL also calls the inline function.

```
int times9(int i) { return times3(times3(i)); }
```

What code gets generated?

The `times9` function sees that the `times3` function is inline, so it inlines the function body and there is no trace of a `times3` function at all. The compiler generates the code as if it had been written

```
int times9(int i) { return (i * 3) * 3; }
```

That would normally be the end of it, except that the `times3` function was marked `dllexport`. This means that the compiler also generates and exports a plain old function called `times3` even though *nobody in the DLL actually calls it as such*. The code is generated and exported because you told the compiler to export the function, so it needs to generate a function in order to export it.

This is not anything special about the `dllexport` keyword. This is just a side-effect of the rule that "If you generate a pointer to an inline function, the compiler must generate a non-inline version of the function and use a pointer to that non-inline version." In this case, the `dllexport` causes a pointer to the function to be placed in the export table.

Okay, next paragraph:

> You can also define as inline a function declared with the **dllimport** attribute. In this case, the function can be expanded (subject to /Ob specifications), but never instantiated. In particular, if the address of an inline imported function is taken, the address of the function residing in the DLL is returned. This behavior is the same as taking the address of a non-inline imported function.

What this is trying to say is that if you declare an inline function as **dllimport**, the compiler treats it just like a plain old inline function: it inlines the function based on the usual rules for inlining. But if the compiler chooses to generate code for the function as if it were not inline (because the compiler decided to ignore the inline qualifier, or because somebody took the address of the inline function), it defers to the generated code from the original DLL, because you said, "Hey, the non-inline version of this function is also available from that DLL over there," and the compiler says, "Awesome, you saved me the trouble of having to generate the non-inline version the function. I can just use that one!"

The "I can just use that one!" is not just an optimization. It is necessary in order to comply with the language standard, which says [dcl.fct.spec] that "An inline function with external linkage shall have the same address in all translation units." This is the compiler-speak way of saying that the address of an inline function must be the same regardless of who asks. You can't have a different copy of the inline function in each DLL, because that would result in them having different addresses. (The "with external linkage" means that the rule doesn't apply to static inline functions, which is behavior consistent with static non-inline functions.)

Okay, let's try paragraph three:

> These rules apply to inline functions whose definitions appear within a class definition. In addition, static local data and strings in inline functions maintain the same identities between the DLL and client as they would in a single program (that is, an executable file without a DLL interface).

The first part of the paragraph is just saying that an inline function defined as part of a class definition counts as an inline function for the purpose of this section. No big deal; we were expecting that.

Update: On the other hand, it is a big deal, because it results in inline functions being exported when you may not realize it. Consider:

```
class __declspec(dllexport) SimpleValue
{
public:
 SimpleValue() : m_value(0) { }
 void setValue(int value);
 int getValue() { return m_value; }
private:
 int m_value;
};
```

The `SimpleValue` constructor and the `SimpleValue::getValue` method are exported inline functions! Consequently, any change to the constructor or to `getValue` requires a recompilation of all code that constructs a `SimpleValue` or calls the `getValue` method. **End update**.

The second part says that if the inline function uses a static local variable or a string literal, it is the same static local variable or string literal everywhere. This is required by the standard [dcl.fct.spec] and is what you would naturally expect:

```
int inline count()
{
 static int c = 0;
 return ++c;
}
```

You expect there to be only one counter.

And the final paragraph:

> Exercise care when providing imported inline functions. For example, if you update the DLL, don’t assume that the client will use the changed version of the DLL. To ensure that you are loading the proper version of the DLL, rebuild the DLL’s client as well.

This is just working through the consequences of the language requirement [dcl.fct.spec] that an inline function “shall have exactly the same definition” everywhere. If you change the definition in the exporting DLL and don’t recompile the importing DLL with the new definition, you have violated a language constraint and the behavior is undefined.

So there you have it. The rules of inline exported functions translated into English.

Raymond Chen

Follow