Why am I getting LNK2019 unresolved external for my inline function?

devblogs.microsoft.com/oldnewthing/20130509-00

May 9, 2013

Raymond Chen

More than once, I've seen somebody confused by how inline functions work.

> I have implemented a few inline functions in one of my cpp files, and I want to use it from other cpp files, so I declare them as `extern`. But sometimes I will get linker error 2019 (unresolved external) for the inline functions.
>
> ```
> // a.cpp
> inline bool foo() { return false; }
>
>
> // b.cpp
> extern bool foo();
>
>
> bool bar() { return foo(); }
> ```

Yup, that's right. The C++ language says in section 3.2(3) [C++03, C++11], and repeats in section 7.1.2(4) [C++03, C++11],

> An inline function shall be defined in every translation unit in which it is used.

(A *translation unit* is the technical term for what we intuitively can think of as a single cpp file and all the files that it `#include`s.)

By putting the definition of `foo` in a cpp file, you make its definition visible only to that cpp file and no other cpp file. When you compile `b.cpp`, sees that you declared it as a normal external function, so it generates a call to it like a normal external function. On the other hand, when you compile `a.cpp`, the compiler sees that `foo` is an inline function, so it says, "I don't need to generate any code yet. Inline functions generate code at the point they are invoked, not at the point they are defined."

Result: `b.cpp` asks for a definition of `foo`, but nobody provides it, because the two declarations were inconsistent. This is a violation of 7.1.2(4) [C++03, C++11] which says "If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required." The magic phrase *no diagnostic is required* means that the compiler is not even required to report the error. (You're lucky that it did!)

This rule makes sense when you think about the classical model of compiling: The compiler logically takes the source code and sends it through the preprocessor. The result (the *translation unit*) then goes into the compiler proper, which learns about structures and classes and functions, and it generates code based on what it sees in that translation unit. The compiler does not have access to other translation units, so when compiling `a.cpp` it can't peek into `b.cpp` and say, "Hm, it looks like somebody is going to be calling `foo` as a non-inline function, so let me also generate a non-inline version of it." And similarly, when the compiler is generating code for the `bar` function, it doesn't peek into `a.cpp` and say, "Hm, it looks like `foo` is actually an inline function. Let me go steal its definition from that other file."

The solution is to move the definition of the inline function into the header file.

Now you can solve this problem:

> I'm getting error LNK2019 for my `GetValue` method. Can somebody explain why?
>
> ```
> // Widget.h
> class Widget
> {
> public:
>  Widget(int initialValue) : value_(initialValue) { }
>  void SetValue(int value);
>  inline int GetValue();
> private:
>  int value_;
> };
>
>
> // Widget.cpp
> #include <widget.h>
>
>
> inline int Widget::GetValue()
> {
>  return value_;
> }
>
>
> // Other.cpp
>
>
> void something()
> {
>  Widget widget(42);
>  printf("%d", widget.GetValue());
> }
> ```

Raymond Chen

Follow