11.10 — Using function templates in multiple files

ALEX[1] OCTOBER 20, 2024

Consider the following program, which doesn't work correctly:

main.cpp:

```
#include <iostream>

template <typename T>
T addOne(T x); // function template forward declaration

int main()
{
    std::cout << addOne(1) << '\n';
    std::cout << addOne(2.3) << '\n';

    return 0;
}
```

add.cpp:

```
template <typename T>
T addOne(T x) // function template definition
{
    return x + 1;
}
```

If `addOne` were a non-template function, this program would work fine: In *main.cpp*, the compiler would be satisfied with the forward declaration of `addOne`, and the linker would connect the call to `addOne()` in *main.cpp* to the function definition in *add.cpp*.

But because `addOne` is a template, this program doesn't work, and we get a linker error:

```
1>Project6.obj : error LNK2019: unresolved external symbol "int __cdecl addOne<int
1>Project6.obj : error LNK2019: unresolved external symbol "double __cdecl addOne<
```

In *main.cpp*, we call `addOne<int>` and `addOne<double>`. However, since the compiler can't see the definition for function template `addOne`, it can't instantiate those functions inside *main.cpp*. It does see the forward declaration for `addOne` though, and will assume those functions exist elsewhere and will be linked in later.

When the compiler goes to compile *add.cpp*, it will see the definition for function template `addOne`. However, there are no uses of this template in *add.cpp*, so the compiler will not instantiate anything. The end result is that the linker is unable to connect the calls to `addOne<int>` and `addOne<double>` in *main.cpp* to the actual functions, because those functions were never instantiated.

As an aside...

If *add.cpp* had instantiated those functions, the program would have compiled and linked just fine. But such solutions are fragile and should be avoided: if the code in *add.cpp* was later changed so those functions are no longer instantiated, the program would again fail to link. Or if *main.cpp* called a different version of `addOne` (such as `addOne<float>`) that was not instantiated in *add.cpp*, we run into the same problem.

The most conventional way to address this issue is to put all your template code in a header (.h) file instead of a source (.cpp) file:

add.h:

```
#ifndef ADD_H
#define ADD_H

template <typename T>
T addOne(T x) // function template definition
{
    return x + 1;
}

#endif
```

main.cpp:

```
#include "add.h" // import the function template definition
#include <iostream>

int main()
{
    std::cout << addOne(1) << '\n';
    std::cout << addOne(2.3) << '\n';

    return 0;
}
```

That way, any files that need access to the template can #include the relevant header, and the template definition will be copied by the preprocessor into the source file. The compiler will then be able to instantiate any functions that are needed.

You may be wondering why this doesn't cause a violation of the one-definition rule (ODR). The ODR says that types, templates, inline functions, and inline variables are allowed to have identical definitions in different files. So there is no problem if the template definition is copied into multiple files (as long as each definition is identical).

Related content

We covered the ODR in lesson 2.7 -- Forward declarations and definitions (https://www.learncpp.com/cpp-tutorial/forward-declarations/#ODR)[2].

But what about the instantiated functions themselves? If a function is instantiated in multiple files, how does that not cause a violation of the ODR? The answer is that functions implicitly instantiated from

templates are implicitly inline. And as you know, inline functions can be defined in multiple files, so long as the definition is identical in each.

Key insight

Template definitions are exempt from the part of the one-definition rule that requires only one definition per program, so it is not a problem to have the same template definition #included into multiple source files. And functions implicitly instantiated from function templates are implicitly inline, so they can be defined in multiple files, so long as each definition is identical.

The templates themselves are not inline, as the concept of inline only applies to variables and functions.

Here's another example of a function template being placed in a header file, so it can be included into multiple source files:

max.h:

```
#ifndef MAX_H
#define MAX_H

template <typename T>
T max(T x, T y)
{
    return (x < y) ? y : x;
}

#endif
```

foo.cpp:

```
#include "max.h" // import template definition for max<T>(T, T)
#include <iostream>

void foo()
{
    std::cout << max(3, 2) << '\n';
}
```

main.cpp:

```
#include "max.h" // import template definition for max<T>(T, T)
#include <iostream>

void foo(); // forward declaration for function foo

int main()
{
    std::cout << max(3, 5) << '\n';
    foo();

    return 0;
}
```

In the above example, both main.cpp and foo.cpp `#include "max.h"` so the code in both files can make use of the `max<T>(T, T)` function template.

Best practice

Templates that are needed in multiple files should be defined in a header file, and then #included wherever needed. This allows the compiler to see the full template definition and instantiate the template when needed.

Next lesson
11.x Chapter 11 summary and quiz

3

Back to table of contents

4

Previous lesson
11.9 Non-type template parameters

5

6

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[9] are connected to your provided email address.

19 COMMENTS Newest

Tim

February 3, 2025 8:15 am PST

Can someone please help me understand how `main.cpp` has access to `foo.cpp` in the last example? I thought it must be included. The only way I see that is through the `max.h` file but I think that is a bit of a

stretch.

0 Reply

Alex Author

Reply to Tim [10] February 5, 2025 6:29 pm PST

main.cpp doesn't have access to foo.cpp.

In main.cpp, the forward declaration `void foo(); // forward declaration for function foo` satisfies the compiler that there is some function `foo()` out there, so it won't generate a compilation error when we call `foo()`.

The linker eventually connects the function call to `foo()` (in main.cpp) to the definition of `foo()` (in foo.cpp).

4 Reply

PooperShooter

December 28, 2024 4:04 pm PST

god i love header files

5 Reply

Schart

Reply to PooperShooter [11] January 26, 2025 6:04 pm PST

Me too, PooperShooter.

4 Reply

Jacob

October 13, 2024 2:25 am PDT

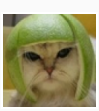
Hey,

I just wanted to thank you on behalf of this course!
I find it better than udemy or similar sites,

A question I'd like to ask though If CPP only has c++ courses!
I am quite eager to learn more and I am looking forward to your reply!

Best Regards,
Jacob.

0 Reply



Alex Author

Reply to Jacob [12] October 16, 2024 2:17 pm PDT

Yep, just C++.

0 Reply

Jacob

Reply to Alex [13] October 18, 2024 12:19 pm PDT

Thanks for replying Alex really appreciate your work!

0 Reply

Polo

September 14, 2024 12:05 pm PDT

Hi Alex! Could you explain why you added the forward declaration in main.cpp instead of in the header file? I've had a longer break from programming, and I am trying to continue learning by revisiting your lessons and some of the code I wrote. I feel like I might have missed this detail.

1 Reply

Alex Author

Reply to Polo [14] September 15, 2024 10:14 pm PDT

If you're talking about `template <typename T> T addOne(T x);` in the first example, it's because there is no header file in this example.

If you're talking about `void foo();`, it's because this is referencing a function in `foo.cpp`, and there is no `foo.h` header file. It wouldn't be appropriate to put the forward declaration for a function defined in `foo.cpp` in `max.h`.

0 Reply

Mexyn

August 2, 2024 7:58 am PDT

Let's say i want to share my dynamic library but want to keep the template function definition private?! Placing it in the header file would be readable by everyone, or did i miss something? Is that even possible?

0 Reply

Alex Author

Reply to Mexyn [15] August 4, 2024 6:08 pm PDT

Yes, generally the implementation of a template must be exposed.

If you know the exact set of types for which you want the template to be instantiated for, you can define it in a .cpp file and use explicit template instantiation. See https://isocpp.org/wiki/faq/templates#separate-template-fn-defn-from-decl

2 Reply

Satoru Gojo
July 30, 2024 3:50 pm PDT

What do you mean by "The templates themselves are not inline, as the concept of inline only applies to variables and functions."

0 Reply

Alex Author
Reply to Satoru Gojo [16] July 30, 2024 7:08 pm PDT

When you define a template (function or class), the entity that is the template is not inline. The functions or classes that are instantiated from the template are.

Templates are able to avoid ODR violations because the ODR explicitly exempts them, not because they are implicitly inline.

The practical effect is essentially the same though. We can define identical templates in multiple translation units.

1 Reply

Max
July 28, 2024 6:24 pm PDT

Are the explicitly defined template instances inline by default?

0 Reply

Alex Author
Reply to Max [17] July 30, 2024 2:42 pm PDT

Only fully specialized templates are not inline by default. Non-specialized and partially specialized template are inline by default.

0 Reply

pawel
July 1, 2024 4:30 pm PDT

Let's say I defined a function template in a header, which was then included in two .cpp files. In both, exactly the same instance of that function is called. Does it result in two copies of that function somewhere in my object files?
I guess that could be an issue if the function is big and used quite a lot, am I right?

1 Reply

Alex Author

Reply to pawel [18] July 4, 2024 1:46 pm PDT

Each translation unit will receive a copy of the instantiated function, which means the function will appear in each object file.

However, the linker should deduplicate these so you will only end up with 1 copy in your executable/library.

2 Reply

NordicPeace

Reply to Alex [19] December 11, 2024 7:07 pm PST

i guess it could be called as implicitly inline.

0 Reply

pawel

Reply to Alex [19] July 4, 2024 1:50 pm PDT

thanks!

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/forward-declarations/#ODR
3. https://www.learncpp.com/cpp-tutorial/chapter-11-summary-and-quiz/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/
6. https://www.learncpp.com/using-function-templates-in-multiple-files/
7. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/
8. https://www.learncpp.com/cpp-tutorial/the-as-if-rule-and-compile-time-optimization/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/#comment-607346
11. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/#comment-605831
12. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/#comment-603025
13. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/#comment-603206
14. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/#comment-601972
15. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/#comment-600419
16. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/#comment-600288
17. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/#comment-600224

18. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/#comment-599107
19. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/#comment-599194
20. https://g.ezoic.net/privacy/learncpp.com