2.7 — Forward declarations and definitions

ALEX[1] SEPTEMBER 13, 2022

Take a look at this seemingly innocent sample program:

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
6      return 0;
7  }
8
9  int add(int x, int y)
10 {
11     return x + y;
12 }
```

You would expect this program to produce the result:

```
The sum of 3 and 4 is: 7
```

But in fact, it doesn't compile at all! Visual Studio produces the following compile error:

```
add.cpp(5) : error C3861: 'add': identifier not found
```

The reason this program doesn't compile is because the compiler compiles the contents of code files sequentially. When the compiler reaches the function call to *add* on line 5 of *main*, it doesn't know what *add* is, because we haven't defined *add* until line 9! That produces the error, *identifier not found*.

Older versions of Visual Studio would produce an additional error:

```
add.cpp(9) : error C2365: 'add'; : redefinition; previous definition was 'for
```

This is somewhat misleading, given that *add* wasn't ever defined in the first place. Despite this, it's useful to generally note that it is fairly common for a single error to produce many redundant or related errors or warnings.

Best practice

When addressing compile errors in your programs, always resolve the first error produced first and then compile again.

To fix this problem, we need to address the fact that the compiler doesn't know what add is. There are two common ways to address the issue.

Option 1: Reorder the function definitions

One way to address the issue is to reorder the function definitions so *add* is defined before *main*:

```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
    return 0;
}
```

That way, by the time *main* calls *add*, the compiler will already know what *add* is. Because this is such a simple program, this change is relatively easy to do. However, in a larger program, it can be tedious trying to figure out which functions call which other functions (and in what order) so they can be declared sequentially.

Furthermore, this option is not always possible. Let's say we're writing a program that has two functions *A* and *B*. If function *A* calls function *B*, and function *B* calls function *A*, then there's no way to order the functions in a way that will make the compiler happy. If you define *A* first, the compiler will complain it doesn't know what *B* is. If you define *B* first, the compiler will complain that it doesn't know what *A* is.

Option 2: Use a forward declaration

We can also fix this by using a forward declaration.

A **forward declaration** allows us to tell the compiler about the existence of an identifier *before* actually defining the identifier.

In the case of functions, this allows us to tell the compiler about the existence of a function before we define the function's body. This way, when the compiler encounters a call to the function, it'll understand that we're making a function call, and can check to ensure we're calling the function correctly, even if it doesn't yet know how or where the function is defined.

To write a forward declaration for a function, we use a **function declaration** statement (also called a **function prototype**). The function declaration consists of the function header (the function's return type, name, and parameter types), terminated with a semicolon. The function body is not included in the declaration.

Here's a function declaration for the *add* function:

```
int add(int x, int y); // function declaration includes return type, name, parameters, and semicolon.  No function body!
```

Now, here's our original program that didn't compile, using a function declaration as a forward declaration for function *add*:

```
#include <iostream>

int add(int x, int y); // forward declaration of add() (using a function declaration)

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n'; // this works because we forward declared add() above
    return 0;
}

int add(int x, int y) // even though the body of add() isn't defined until here
{
    return x + y;
}
```

Now when the compiler reaches the call to *add* in main, it will know what *add* looks like (a function that takes two integer parameters and returns an integer), and it won't complain.

It is worth noting that function declarations do not need to specify the names of the parameters. In the above code, you can also forward declare your function like this:

```
int add(int, int); // valid function declaration
```

However, we prefer to name our parameters (using the same names as the actual function), because it allows you to understand what the function parameters are just by looking at the declaration. Otherwise, you'll have to locate the function definition.

Best practice

Keep the parameter names in your function declarations.

Tip

You can easily create function declarations by copy/pasting your function's header and adding a semicolon.

Forgetting the function body

New programmers often wonder what happens if they forward declare a function but do not define it.

The answer is: it depends. If a forward declaration is made, but the function is never called, the program will compile and run fine. However, if a forward declaration is made and the function is called, but the program never defines the function, the program will compile okay, but the linker will complain that it can't resolve the function call.

Consider the following program:

```cpp
#include <iostream>

int add(int x, int y); // forward declaration of add()

int main()
{
    std::cout << "The sum of 3 and 4 is: " << add(3, 4) << '\n';
    return 0;
}

// note: No definition for function add
```

In this program, we forward declare *add*, and we call *add*, but we never define *add* anywhere. When we try and compile this program, Visual Studio produces the following message:

```
Compiling...
add.cpp
Linking...
add.obj : error LNK2001: unresolved external symbol "int __cdecl add(int,int)
add.exe : fatal error LNK1120: 1 unresolved externals
```

As you can see, the program compiled okay, but it failed at the link stage because *int add(int, int)* was never defined.

Other types of forward declarations

Forward declarations are most often used with functions. However, forward declarations can also be used with other identifiers in C++, such as variables and user-defined types. Variables and user-defined types have a different syntax for forward declaration, so we'll cover these in future lessons.

Declarations vs. definitions

In C++, you'll frequently hear the words "declaration" and "definition" used, and often interchangeably. What do they mean? You now have enough fundamental knowledge to understand the difference between the two.

A **definition** actually implements (for functions or types) or instantiates (for variables) the identifier. Here are some examples of definitions:

```cpp
int add(int x, int y) // implements function add()
{
    int z{ x + y }; // instantiates variable z

    return z;
}
```

A definition is needed to satisfy the *linker*. If you use an identifier without providing a definition, the linker will error.

The **one definition rule** (or ODR for short) is a well-known rule in C++. The ODR has three parts:

1. Within a given *file*, a function, variable, type, or template can only have one definition.
2. Within a given *program*, a variable or normal function can only have one definition. This distinction is made because programs can have more than one file (we'll cover this in the next lesson).
3. Types, templates, inline functions, and inline variables are allowed to have identical definitions in different files. We haven't covered what most of these things are yet, so don't worry about this for now -- we'll bring it back up when it's relevant.

Violating part 1 of the ODR will cause the compiler to issue a redefinition error. Violating ODR part 2 will likely cause the linker to issue a redefinition error. Violating ODR part 3 will cause undefined behavior.

Here's an example of a violation of part 1:

```
int add(int x, int y)
{
     return x + y;
}

int add(int x, int y) // violation of ODR, we've already defined function add
{
     return x + y;
}

int main()
{
    int x;
    int x; // violation of ODR, we've already defined x
}
```

Because the above program violates ODR part 1, this causes the Visual Studio compiler to issue the following compile errors:

```
project3.cpp(9): error C2084: function 'int add(int,int)' already has a body
project3.cpp(3): note: see previous definition of 'add'
project3.cpp(16): error C2086: 'int x': redefinition
project3.cpp(15): note: see declaration of 'x'
```

For advanced readers

Functions that share an identifier but have different parameters are considered to be distinct functions. We discuss this further in lesson 8.9 -- Introduction to function overloading (https://www.learncpp.com/cpp-tutorial/introduction-to-function-overloading/)[2]

A **declaration** is a statement that tells the *compiler* about the existence of an identifier and its type information. Here are some examples of declarations:

```
int add(int x, int y); // tells the compiler about a function named "add" that takes two int parameters and returns an int.  No body!
int x; // tells the compiler about an integer variable named x
```

A declaration is all that is needed to satisfy the compiler. This is why we can use a forward declaration to tell the compiler about an identifier that isn't actually defined until later.

In C++, all definitions also serve as declarations. This is why *int x* appears in our examples for both definitions and declarations. Since *int x* is a definition, it's a declaration too. In most cases, a definition serves our purposes, as it satisfies both the compiler and linker. We only need to provide an explicit declaration when we want to use an identifier before it has been defined.

While it is true that all definitions are declarations, the converse is not true: not all declarations are definitions. An example of this is the function declaration -- it satisfies the compiler, but not the linker. These declarations that aren't definitions are called **pure declarations**. Other types of pure declarations include forward declarations for variables and type declarations (you will encounter these in future lessons, no need to worry about them now).

The ODR doesn't apply to pure declarations (it's the *one definition rule*, not the *one declaration rule*), so you can have as many pure declarations for an identifier as you desire (although having more than one is redundant).

Author's note

In common language, the term "declaration" is typically used to mean "a pure declaration", and "definition" is used to mean "a definition that also serves as a declaration". Thus, we'd typically call *int x;* a definition, even though it is both a definition and a declaration.

Quiz time

Question #1

What is a function prototype?

Show Solution (javascript:void(0))[3]

Question #2

What is a forward declaration?

Show Solution (javascript:void(0))[3]

Question #3

How do we declare a forward declaration for functions?

Show Solution (javascript:void(0))[3]

Question #4

Write the function declaration for this function (use the preferred form with names):

```cpp
int doMath(int first, int second, int third, int fourth)
{
     return first + second * third / fourth;
}
```

Show Solution (javascript:void(0))[3]

Question #5

For each of the following programs, state whether they fail to compile, fail to link, fail both, or compile and link successfully. If you are not sure, try compiling them!

a)

```cpp
#include <iostream>
int add(int x, int y);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

Show Solution (javascript:void(0))[3]

b)

```cpp
#include <iostream>
int add(int x, int y);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Show Solution (javascript:void(0))[3]

c)

```cpp
#include <iostream>
int add(int x, int y);

int main()
{
    std::cout << "3 + 4 = " << add(3, 4) << '\n';
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Show Solution (javascript:void(0))[3]

d)

```cpp
#include <iostream>
int add(int x, int y, int z);

int main()
{
    std::cout << "3 + 4 + 5 = " << add(3, 4, 5) << '\n';
    return 0;
}

int add(int x, int y, int z)
{
    return x + y + z;
}
```

Show Solution (javascript:void(0))[3]

Next lesson
2.8 Programs with multiple code files

4

Back to table of contents

5

Previous lesson
2.6 Why functions are useful, and how to use them effectively

6

7

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

Leave a comment...

Name*

Email*

Find a mistake? Leave a comment!

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

Notify me about replies:

POST COMMENT

392 COMMENTS

Newest

hantao

October 30, 2022 6:19 am

for c), this compiles and links.

```cpp
#include <iostream>
int add(int x, int y);

int main()
{
    std::cout << "3 + 4 = " << add(3, 4) << '\n';
    return 0;
}



int add(int x, int y, int z)
{
    return x + y + z;

}int add(int x, int y)
{
    return x + y;
}
```

0 Reply

is it arguments or parameters

September 5, 2022 3:43 pm

Q5-b)the solution part:

>>because the add() function that was forward declared only takes **2 arguments.**
is it arguments or parameters?

0 Reply

Alex Author

Reply to is it arguments or parameters September 13, 2022 4:54 pm

In this context, I think either works. The forward declaration for add(int, int) only has 2 parameters, so it only accepts 2 arguments.

I focus on arguments here because the functions themselves are well formed (it's not an issue with the parameters). The issue is that we're calling add() with 3 arguments before the compiler has seen a declaration of add() that can handle 3 arguments.

0 Reply

Violating ODR part 3 will cause undefined behavior

September 5, 2022 3:34 pm

>>Violating ODR part 3 will cause undefined behavior.

What is it like violating rule #3? Is it like having distinct identifiers?

0 Reply

Alex Author

Reply to Violating ODR part 3 will cause undefined behavior September 13, 2022 4:51 pm

I'm not sure what you're asking.

0 Reply

Violating ODR part 3 will cause undefined behavior

Reply to Alex September 14, 2022 3:41 am

in ODR3 part3, it says "types, templates, inline functions, and inline variables are allowed to have IDENTICAL definitions in different files. ", so my question is violating this rule means not having IDENTICAL definitions and having distinct definition for types, templates, inline functions?

0 Reply

Alex Author

Reply to Violating ODR part 3 will cause undefined behavior

September 15, 2022 10:45 pm

Yes. You will get undefined behavior if you have non-identical definitions for a given type, template, inline function, or inline variable.

1 Reply

normal function

September 5, 2022 1:13 pm

>>Within a given program, a variable** or normal function** can only have one definition. This distinction is made because programs can have more than one file (we'll cover this in the next lesson).

What do you mean by normal function? Do we have abnormal functions excluded from this rule?

0 Reply

Alex Author

Reply to normal function September 13, 2022 4:48 pm

Yes, inline functions are excluded from this rule. We cover inline functions in lesson https://www.learncpp.com/cpp-tutorial/inline-functions/

1 Reply

asd

August 14, 2022 6:06 am

hi, is it important to differentiate whether compiler or linker will fail? i got 4/4 on quiz, but only if program would work or not, not if compiler or linker would fail.

1 Reply

Alex Author

Reply to asd August 15, 2022 11:34 am

In terms of a program working or not, not really. But if you understand the difference between the two and what is needed to satisfy each, then you'll be able to write better code and debug more quickly.

1 Reply

asd

Reply to Alex August 18, 2022 3:03 am

thanks for the reply.

0 Reply

smalcakmak

August 6, 2022 3:31 am

Hi Alex. Thanks for sharing this. I have a question. in exercise c, you say it will give an linker error. But isnt linker for linking different files. in exercise, whole program is in one file.

1 Reply

Alex Author

Reply to smalcakmak August 6, 2022 11:15 pm

Yes. In this case, the compiler is satisfied because we've provided a prototype for add(). As far as it knows, the definition of add() is in some other file -- it can't tell whether that's true or not, it just assumes it is.

Then the linker comes along and sees that we have a function call to add(). It goes to connect the function call to add() to the compiled definition for add(), and discovers that there isn't a definition for add() that it can find. It will flag this as an error.

Even in a one-file program, the linker is still typically linking in the precompiled standard library (and all of the definitions that exist as part of that).

1 Reply

ali
August 5, 2022 8:20 pm

In C++, all definitions also serve as declarations. This is why int x appears in our examples for both definitions and declarations. Since int x is a definition, it's a declaration too.

My question is, isn't `int x` merely a declaration, since it doesn't have any body? How's `int x` a definition hmm?

1 Reply

Alex Author

Reply to ali August 6, 2022 10:56 pm

Only functions have bodies. Variables do not (they may have initializers).

`int x` would be the definition of variable `x` of type int (without an initializer).

0 Reply

Krishna
July 20, 2022 11:47 am

```
std::cout << "3 + 4 + 5 = " << add(3, 4) << '\n';
```

In question 5c, do you want to just make it "3 + 4" for the string literal above?

0 Reply

Alex Author

Reply to Krishna July 21, 2022 11:25 am

Sure, I think that's probably better. Thanks!

0 Reply

Erwin de Geus

May 24, 2022 8:52 am

I get a slightly different error message from
the compiler. Instead of: add.cpp : c3861: 'add':
identifier not found, I get: E0020 identifier "add" is undefined. It's about the first example in this lesson.

0 Reply

cdx

April 22, 2022 3:35 am

Is "all declarations are not definitions" the right way to say it? It seems to contradict with "at least one declaration is a definition", which is true, e.g. "int x;". The more correct way to me seems to be "not all declarations are definitions" or "all declarations are not necessarily definitions". I'm non-native Enligsh speaker though and I haven't studied logic in English, so maybe I'm just reading it wrong.

4 Reply

Ronit

Reply to cdx May 27, 2022 12:39 am

You're not wrong at all here - the easiest way to understand this would definitely be "not all declarations are definitions". It is a thing in English, though, to say "all declarations are not definitions" - at least, my textbooks have used that form of phrasing very often, and it does get very confusing (I don't like that way of phrasing it at all).

0 Reply

Landen

Reply to cdx April 22, 2022 5:46 am

Logic in school was terrible for me, and I still get headaches thinking about logic. For my comprehension, I suppose **all declarations are not definitions** could translate to **every single declaration cannot be labeled as a definition** . Your suggestion of **not all declarations are definitions** makes the statement clear to me. However, this is going off of English semantics only and not literal logic (if there even is such a difference). I'm actually curious is the current statement is logically true...

0 Reply

Alex Author

Reply to Landen April 22, 2022 10:40 am

Yes, I think you're correct. Phrasing it as "not all declarations are definitions" is better.

2 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-function-overloading/
3. javascript:void(0)
4. https://www.learncpp.com/cpp-tutorial/programs-with-multiple-code-files/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/why-functions-are-useful-and-how-to-use-them-effectively/
7. https://www.learncpp.com/forward-declarations/
8. https://www.learncpp.com/wordpress/recent-news-box-for-tiga-102-theme-updated/
9. https://gravatar.com/
10. https://www.ezoic.com/what-is-ezoic/