# 2.6 — Forward declarations and definitions

ALEX AUGUST 30, 2021

Take a look at this seemingly innocent sample program:

```
#include <iostream>

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

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

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: &#039;add&#039;: 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: &#039;add&#039; : redefinition; previous definition was &#039;formerly unknown identifier&#039;
```

**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 declaration statement called a *function prototype*. The function prototype consists of the function's return type, name, parameters, but no function body (the curly braces and everything in between them), terminated with a semicolon.**

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

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

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

```
#include <iostream>

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

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 prototypes 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 prototype
```

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 prototype. Otherwise, you'll have to locate the function definition.

**Best practice**

When defining function prototypes, keep the parameter names. You can easily create forward declarations by using copy/paste on your function declaration. Don't forget the semicolon on the end.

## 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:

```
#include <iostream>

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

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@@YAHHH@Z)
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 often hear the words "declaration" and "definition" used, often interchangeably. What do they mean? You now have enough of a framework 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:**

```
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, object, type, or template can only have one definition.**
2. **Within a given *program*, an object 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 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**

**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: all declarations are not definitions. An example of this is the function prototype -- 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

**Question #2**

**What is a forward declaration?**

Show Solution

**Question #3**

**How do we declare a forward declaration for functions?**

Show Solution

**Question #4**

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

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

Show Solution

**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)**

```
#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

**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**

**c)**

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

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

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

**Show Solution**

**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**

---

Next lesson
2.7 Programs with multiple code files

Back to table of contents

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

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

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

Notify me about replies: POST COMMENT

332 COMMENTS

Newest