1.7 — Forward declarations and definitions

BY ALEX ON JUNE 2ND, 2007 | LAST MODIFIED BY ALEX ON APRIL 13TH, 2018

Take a look at this seemingly innocent sample program called add.cpp:

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "The sum of 3 and 4 is: " << add(3, 4) << std::endl;
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 2005 Express produces the following compile errors:

```
add.cpp(5) : error C3861: 'add': identifier not found
add.cpp(9) : error C2365: 'add' : redefinition; previous definition was 'formerly unknown identifi
```

The reason this program doesn't compile is because the compiler reads 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 first error ("identifier not found").

When Visual Studio 2005 gets to the actual declaration of add() on line 9, it also complains about add being redefined. This is somewhat misleading, given that it wasn't ever defined in the first place. Later versions of Visual Studio correctly omit this additional error message.

Despite the redundancy of the second error, it's useful to note that it is fairly common for a single error to produce (often redundant) multiple compiler errors or warnings.

Rule: When addressing compile errors in your programs, always resolve the first error produced first.

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 calls so add() is defined before main():

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

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 they will both be 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.

Function prototypes and forward declaration of functions

Option 2: Use 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 part between the curly braces). And because the function prototype is a statement, it ends 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) << std::endl; // 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 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);
```

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.

Tip: You can easily create function prototypes by using copy/paste on your function declaration. Don't forget the semicolon on the end.

Forgetting the function body

One question many new programmers have is: what happens if we 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, 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) << std::endl;
    return 0;
}
```

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 2005 Express 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. Other types of identifiers (e.g. user-defined types) have a different syntax for forward declaration.

We'll talk more about how to forward declare other types of identifiers in future lessons.

Declarations vs. definitions

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

A **definition** actually implements or instantiates (causes memory to be allocated for) the identifier. Here are some examples of definitions:

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

int x; // instantiates (causes memory to be allocated for) an integer variable named x
```

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 two parts:
1) Within a given file, an identifier 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). Note that some other types of identifiers (such as types, template functions, and inline functions) are exempt from this rule (we haven't covered what these are yet, so don't worry about this for now -- we'll bring it back up when it's relevant).

Violating the ODR will generally cause the compiler or linker to issue a redefinition error, even if the definitions are identical.

A **declaration** is a statement that tells the compiler about the existence of an identifier (variable or function name) and its type. 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 using a forward declaration is enough to keep the compiler happy. If you use an identifier without providing a declaration, the compiler will error.

You'll note that "int x" appears in both categories. In C++, all definitions also serve as declarations. Since "int x" is a definition, it's by default a declaration too. Therefore, in many cases, we only need a definition. However, if you need to use an identifier before it's defined, then you'll need to provide an explicit declaration. This is why our use of a forward declaration in the above example is necessary.

There is a small subset of declarations that are not definitions, such as function prototypes. These are called **pure declarations**. Other types of pure declarations include forward declarations for variables, class declarations, and type declarations (you will encounter these in future lessons, but don't need to worry about them now). You can have as many pure declarations for an identifier as you desire (although having more than one is redundant).

Quiz

1) What's the difference between a function prototype and a forward declaration?

2) Write the function prototype for this function:

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

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

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

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

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

4)

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

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

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

5)

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

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

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

6)

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

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

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

Quiz Answers

1) **Hide Solution**

A function prototype is a declaration statement that includes a function's name, return type, and parameters. It does not include the function body.

A forward declaration tells the compiler that an identifier exists before it is actually defined.

For functions, a function prototype serves as a forward declaration.

Other types of identifiers (e.g. variables and user-defined types) have a different syntax for forward declaration.

2) **Hide Solution**

```
// Either of these is correct.
// Do not forget the semicolon on the end, since these are statements.
int doMath(int first, int second, int third, int fourth); // better solution
int doMath(int, int, int, int);
```

3) **Hide Solution**

Doesn't compile. The compiler will complain that the add() called in main() does not have the same number of parameters as the one that was forward declared.

4) **Hide Solution**

Doesn't compile. The compiler will complain that the add() called in main() does not have the same number of parameters as the one that was forward declared.

5) **Hide Solution**

Doesn't link. The compiler will match the forward declared prototype of add to the function call to add() in main(). However, no add() function that takes two parameters was ever implemented (we only implemented one that took 3 parameters), so the linker will complain.

6) **Hide Solution**

Compiles and links. The function call to add() matches the prototype that was forward declared, the implemented function also matches.

1.8 -- Programs with multiple files