2.7 — Forward declarations and definitions

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: '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 'formerly
unknown identifier'
```

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. It can sometimes be hard to tell whether any error or warning beyond the first is a consequence of the first issue, or whether it is an independent issue that needs to be resolved separately.

Best practice

When addressing compilation errors or warnings in your programs, resolve the first issue listed 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's return type, name, and parameter types, terminated with a semicolon. The names of the parameters can

be optionally included. 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 (as they are not considered to be part of the function declaration). 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). This allows you to understand what the function parameters are just by looking at the declaration. For example, if you were to see the declaration `void doSomething(int, int, int)`, you may think you remember what each of the parameters represent, but you may also get it wrong.

Also many automated documentation generation tools will generate documentation from the content of header files, which is where declarations are often placed. We discuss header files and declarations in lesson 2.11 -- Header files.

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.

Why forward declarations?

You may be wondering why we would use a forward declaration if we could just reorder the functions to make our programs work.

Most often, forward declarations are used to tell the compiler about the existence of some function that has been defined in a different code file. Reordering isn't possible in this scenario because the caller and the callee are in completely different files! We'll discuss this in more detail in the next lesson (2.8 -- Programs with multiple code files).

Forward declarations can also be used to define our functions in an order-agnostic manner. This allows us to define functions in whatever order maximizes organization (e.g. by clustering related functions together) or reader understanding.

Less often, there are times when we have two functions that call each other. Reordering isn't possible in this case either, as there is no way to reorder the functions such that each is before the other. Forward declarations give us a way to resolve such circular dependencies.

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()

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 types. Variables and 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 **declaration** tells the *compiler* about the *existence* of an identifier and its associated 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 **definition** is a declaration that actually implements (for functions and 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;
}

int x;                // instantiates variable x
```

In C++, all definitions are declarations. Therefore `int x;` is both a definition and a declaration.

Conversely, not all declarations are definitions. Declarations that aren't definitions are called **pure declarations**. Types of pure declarations include forward declarations for function, variables, and types.

Nomenclature

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.

When the compiler encounters an identifier, it will check to ensure use of that identifier is valid (e.g. that the identifier is in scope, that it is used in a syntactically valid manner, etc…).

In most cases, a declaration is sufficient to allow the compiler to ensure an identifier is being used properly. For example, when the compiler encounters function call `add(5, 6)`, if it has already seen the declaration for `add(int, int)`, then it can validate that `add` is actually a function that takes two `int` parameters. It does not need to have actually seen the definition for function `add` (which may exist in some other file).

However, there are a few cases where the compiler must be able to see a full definition in order to use an identifier (such as for template definitions and type definitions, both of which we will discuss in future lessons).

Here's a summary table:

Term	Technical Meaning	Examples

Declaration	Tells compiler about an identifier and its associated type information.	void foo(); // function forward declaration (no body)
		void goo() {}; // function definition (has body)
		int x; // variable definition

Definition	Implements a function or instantiates a variable. Definitions are also declarations.	void foo() { } // function definition (has body) int x; // variable definition
Pure declaration	A declaration that isn't a definition.	void foo(); // function forward declaration (no body)
Initialization	Provides an initial value for a defined object.	int x { 2 }; // 2 is the initializer

The term “declaration” is commonly used to mean “pure declaration”, and the term “definition” used for anything that is both a definition and a declaration. We use this common nomenclature in the example column comments.

The one definition rule (ODR)

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

1. Within a *file*, each function, variable, type, or template can only have one definition. Definitions occurring in different scopes (e.g. local variables defined inside different functions, or functions defined inside different namespaces) do not violate this rule.
2. Within a *program*, each function or variable can only have one definition. This rule exists because programs can have more than one file (we’ll cover this in the next lesson). Functions and variables not visible to the linker are excluded from this rule (discussed further in lesson 7.6 -- Internal linkage).
3. Types, templates, inline functions, and inline variables are allowed to have duplicate definitions in different files, so long as each definition is identical. 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.

Related content

We discuss ODR part 3 exemptions further in the following lessons:

- Types (13.1 -- Introduction to program-defined (user-defined) types).
- Function templates (11.6 -- Function templates and 11.7 -- Function template instantiation).
- Inline functions and variables (5.7 -- Inline functions and variables).

Violating part 1 of the ODR will cause the compiler to issue a redefinition error. Violating ODR part 2 will 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(int,
int)
{
     return x + y;
}

int main()
{
    int x{};
    int x{ 5 }; // violation of ODR, we've already defined x
}
```

In this example, function `add(int, int)` is defined twice (in the global scope), and local variable `int x` is defined twice (in the scope of `main()`). The Visual Studio compiler thus issues 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'
```

However, it is not a violation of ODR part 1 for `main()` to have a local variable defined as `int x` and `add()` to also have a function parameter defined as `int x`. These definitions occur in different scopes (in the scope of each respective function), so they are considered to be separate definitions for two distinct objects, not a definition and redefinition of the same object.

For advanced readers

Functions that share an identifier but have different sets of parameters are also considered to be distinct functions, so such definitions do not violate the ODR. We discuss this further in lesson 11.1 -- Introduction to function overloading.

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 declaration 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, 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)

```
#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)

```
#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

d)

```
#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 z, int y, int x) // names don't match the declaration
{
    return x + y + z;
}
```

Show Solution

e)

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

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