C++

Information
Tutorials
Reference
Articles
Forum

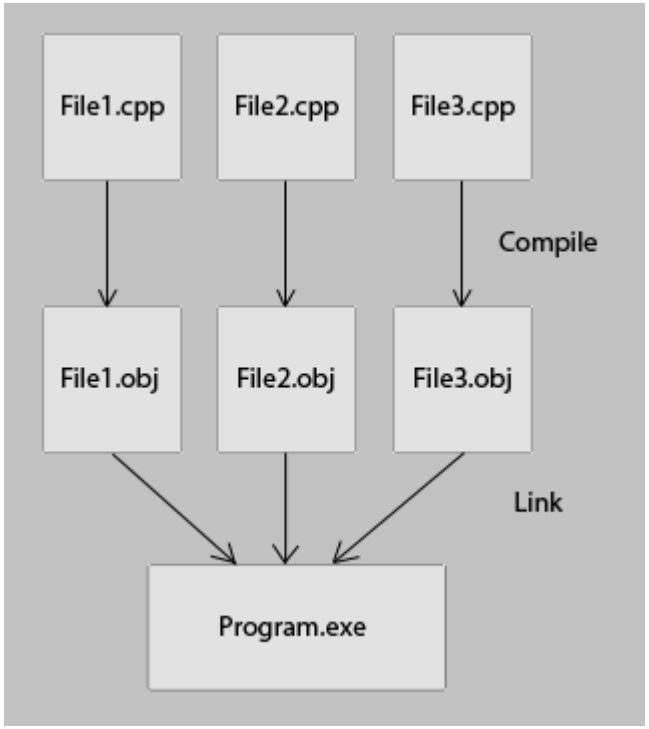
Articles

Algorithms
C++ 11
Graphics and multimedia
How-To
Language Features
Unix/Linux programming
Source Code
Standard Library
Tips and Tricks
Tools and Libraries
Visual C++
Windows API

Published by **Esslercuffi**
Oct 26, 2014 (last update: Oct 28, 2014)

Declarations, Prototypes, Definitions, and Implementations

Many people, when learning C++, seem to have some confusion about the distinction between and purpose of declarations and definitions. Especially concerning functions or classes. To understand why all this seemingly pedantic and repetitive stuff is necessary, let's review the build process.



First, each .cpp file is compiled independently. During this process any #include directives will first insert the included file into the .cpp. Compilation then processes the .cpp from top to bottom, generating machine language code and outputting this into an .obj file. The thing to keep in mind for purposes of this discussion is that it is a single pass process from top to bottom, and that it is completely unaware of anything in other .cpp files. Nor is the compiler even aware of anything in the current .cpp file at any point beyond the point at which it is processing. The code generated by the compiler does not fully resolve memory addressing and function calls. That's the job of the linker. When the compiler encounters a function call in a .cpp file, it usually has no idea what the function does, or how it does it. All the

compiler need to know is what parameters are passed and what type the return value is. The .obj code for the function call can be generated, and the linker will figure out exactly where to make the jump later.

Now keeping all this in mind we can discuss the real topic.

DECLARATIONS: A declaration introduces a name into a scope. Generally speaking, a scope is either an entire .cpp file or anything in code delimited by {}, be it a function, a loop within a function, or even an arbitrarily placed block of {} within a function. A name introduced, is visible within the scope from the point at which it is declared to the end of that scope. A declarations merely tells the compiler how to use something, it does not actually create anything.

```
1 extern int y; // declares y, but does not define it.  y is defined elsewhere,
2               // but the program can now use it since it knows what it is (an integer)
```

PROTOTYPES: A prototype is just another name for a **declaration** of a function.

```
double someFunction( double, int );
```

DEFINITIONS: A definition fully specifies an entity. Definitions are where the actual creation of the entity in memory takes place. All definitions are also declarations, but not all declarations are definitions.

```
Int x;  // declares and defines x.  it is a definition because it creates the variable allocating memory
```

IMPLEMENTATIONS: An implementation is another name for a **definition** of a function. That is, the implementation is the actual code of the function itself.

```
1 double someFunction( double x, int y )
2 {
3         return x * y;
4 }
```

Any entity can be declared multiple times, but can only be defined once.

Why all this matters to you

```
int main()
{
        double a = 3.5;
        int b = 2;
        double c;

        c = someFunction( a, b );
}

double someFunction( double x, int y )
{
        return x * y;
}
```

Recall that compilation is a single pass top-down process. Because of this, the above code can not work because the compiler gets to the "c = someFunction(a, b);" line and doesn't know how to handle someFunction(). The solution is to have someFunction() declared before this.

```
double someFunction( double, int );

int main()
{
        double a = 3.5;
        int b = 2;
        double c;

        c = someFunction( a, b );
}

double someFunction( double x, int y )
{
        return x * y;
}
```

```
double someFunction( double x, int y )
{
        return x * y;
}

int main()
{
        double a = 3.5;
        int b = 2;
        double c;

        c = someFunction( a, b );
}
```

Since a definition is also a declaration, either of the above approaches works. The first uses a prototype, while the second just moved the definition of someFunction() ahead of the call to it in main(). As a general rule, using prototypes is the preferred method as it allows code to be organized better (you don't have to start at the bottom and work up as you read it) and prevents errors being introduced if code is reorganized. Also, consider a set of recursive functions in which each calls the other.

```
1  funcA()
2  {
3          if( condition )
4                  funcB();
5          return;
6  }
7
8  funcB()
9  {
10         if( condition )
11                 funcC();
12         return;
13 }
14
15 funcC()
16 {
17         if( condition )
18                 funcA();
19         return;
20 }
```

This may sound bizarre to new students, but it is rather common in certain types of algorithms like sorting and grammar parsing. The thing to note is that there is no way to organize these functions without having at least one prototype somewhere. So do yourself a favor and get into the habit of using prototypes.

A note about .h and .cpp organization

All of the above relates directly to how you should organize your code into header files and source files. Most students learning C++ won't be writing programs large enough to require multiple source files for the first month or so of their study. But when they do, I've seen a lot of them get confused about what should be put in which file. The rule of thumb is this: Header files should contain **declarations**, sourch files should contain **definitions**. The reason for this should be fairly clear now. An entity can be declared multiple times, but only defined once. If a header file contains definitions, you can end up with the same entity being defined more than once. This will likely cause problems in the linking process when the linker tries to resolve external addressing and finds multiple entities with the same name.

For a more thorough discussion about header file organization, see this article by Disch.

(hover mouse and click)