



## About Compiling

What most people mean by the phrase "*compiling a program*" is actually two separate steps in the creation of that program. The first step is proper **compilation**. Compilation is translating high level programming instructions into machine language instructions. The input to compilation is a source code file in a high level language such as C or C++. Source code files have extensions such as ".c", ".cpp", or ".cc". The output of compilation is an **object file**, which is not quite an executable file. Object files usually have a ".o" or ".obj" extension.

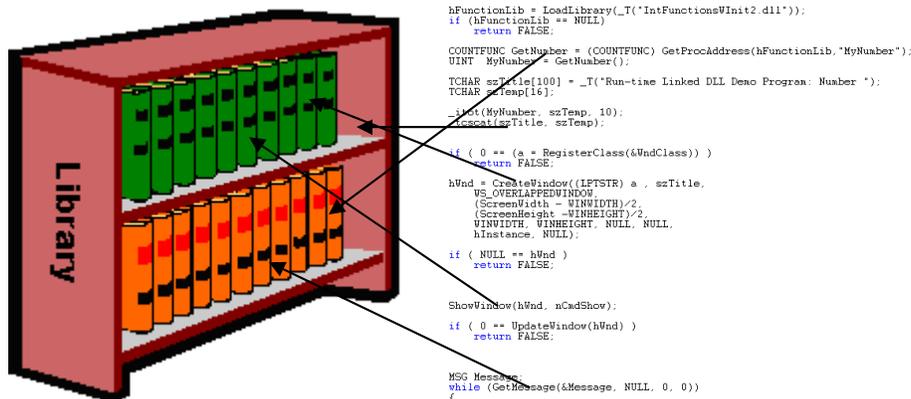
Consider the C++ code fragment

```
#include <iostream>
#include <math.h>
using namespace std;
double number;
cout << "Enter a positive number here:";
cin >> number;
cout << "The square root is " << sqrt(number) << endl;
```

The first two lines tell the compiler to copy the contents of the header files `iostream` and `<math.h>` into the program at those points in the file. The third line tells the compiler to use the `std` namespace for resolving symbols, since the `iostream` header file just copied into the program is declared within this `std` namespace. In other words, every declaration and definition in this header file is contained within the namespace known as `std`. A namespace is essentially just a scope, so all of `iostream` has scope `std`.

In particular, `cout` and `cin` are really known to the world outside of the `std` scope by their proper names, `std::cout` and `std::cin`. If you wrote `std::cout` instead of `cout`, `std::cin` instead of `cin`, and `std::endl` instead of `endl`, you could eliminate the third line. The "using namespace `std`" directive tells the compiler that whenever it finds a symbol in the program that is not defined in the program, it should search the `std` namespace in case it is defined there. Names like `cin` and `cout` are called **external symbols** in a program because their definitions are not contained in the program itself.

The inclusion of the header files `<iostream>` and `<math.h>` in the program allows the compiler to determine whether the names `cin`, `cout`, and `sqrt` are being used properly, thereby allowing it to compile the code, but it cannot create an executable module, because the objects associated with the names `cin` and `cout` are not defined in your program. Because the name `cin` is defined in a separate file, the compiler cannot create a jump instruction to jump to the code that does stream extraction, i.e., "`cin >>`", because there is no memory address associated with this code. Names like `cin` and `cout` that are defined outside of the program module are said to be **unresolved** at compile time.



The most that the compiler can do is to create an entry in a table in the code that allows the second stage to solve the problem. This table contains the location of every instruction in the program that refers to a name whose location is unresolved, or external, to the program. The second stage is **linking**, and it is performed by, not surprisingly, the **linker**. The linker's job is to find the unresolved names listed in the table in the executable module and to **link** them to the actual objects to which they refer. To link a name means to replace it with the address to which it refers. Of course a name cannot be associated with an address unless the object that it names actually has an address, which implies that before the name can be resolved, the associated object must be incorporated into the address space of the executable file. There is a special type of linking called **dynamic linking** that is an exception to this rule, but how that works is a subject for a different chapter. Static linking is the type of linking in which all code needed at runtime is actually copied into the program

## What is Separate Compilation?

Projects should be organized into collections of small files that can be compiled individually. Typically, large classes are given their own files and smaller classes may be grouped together into a single file. Sometimes collections of functions that are not members of any class are placed into a separate file. As long as each of the files is included in the project file, the compiler will usually compile each of them when it is given the instruction to compile the project.

The usu

A set of functions that all provide various forms of randomization, for example, would be placed into two files named `myrandstuff.h` and `myrandstuff.cpp`. The `myrandstuff.h` file contains the function prototypes. Files that end in a ".h" are called **header files**. They are usually not compiled by the compiler. Instead they are included into the implementation files at compile time so that the compiler will have access to the symbols defined in the header file. The `myrandstuff.cpp` file contains the definitions of the functions defined in the header file. The files will usually have the following form.



myrandstuff.h:

```
#ifndef MYRANDSTUFF_H
#define MYRANDSTUFF_H

// whatever header files need to be included go here
// any typedefs or other type or class definitions go here

// description of func1
void func1( ... );

// description of func2
void func2( ... );

// and so on

#endif // MYRANDSTUFF_H
```

myrandstuff.cpp:

```
#include "myrandstuff.h"

void func1( ... )
{
    // implementation for func1
}

void func2( ... )
{
    // implementation for func2
}

// and all implementation code for remaining functions here
```

The main program only includes the header files, not the .cpp files, so that it can make reference to functions declared there and used in .. Therefore, the main program will contain a line of the form

```
#include "myrandstuff.h"
```

among the other header files included by it. Notice that the header file name is enclosed in double quotes, not angle braces. When the file to be included is in the same directory as the program, its name should be in double quotes.

What, you may be wondering, is the purpose of those lines at the beginning and end of the file that look like preprocessor directives?

When the macro preprocessor sees the #include directive, it finds the file that is to be included and copies it into the program at the point at which the #include directive was found. Every included file is copied into the main program.



Suppose that you have a second implementation file, say `mylist.cpp`, that uses the functions declared or the types defined in `myrandstuff.h`. It is possible that the header file `mylist.h` needs to include `myrandstuff.h`. Suppose also that the main program uses the functions declared in `mylist.h`. Then `mylist.h` has the line

```
#include "myrandstuff.h"
```

and the main program has the two lines

```
#include "myrandstuff.h"  
#include "mylist.h"
```

When the compiler runs, it copies the `myrandstuff.h` file twice, one after the other, first because of the first `#include` directive, then again because when it includes `mylist.h`, it will copy `myrandstuff.h` again because of the `#include` directive in that file. Now we can explain the reason for the lines

```
#ifndef MYRANDSTUFF_H  
#define MYRANDSTUFF_H  
  
#endif // MYRANDSTUFF_H
```

The first line translates to “if the macro symbol `MYRANDSTUFF_H` is not defined then continue reading and copying lines into the file until the matching occurrence of `endif`. On the other hand, if it is defined, then skip reading code until immediately after that matching `endif`”. The second line is therefore only read if the symbol was not defined. In this case that line defines it, and the code is read and copied. When the preprocessor opens the next file, `mylist.h`, the symbol `MYRANDSTUFF_H` is defined already and the code from `myrandstuff.h` is not copied twice into the `main()` function. Thus, these three lines prevent the macro preprocessor from putting multiple copies of code into other compilation units. These lines are often called a *header guard*.

You can use whatever symbol you want in this directive, but it must be unique in your project. It is best to follow a convention that ensures this uniqueness. The most common is to use the symbol consisting of the file name. in caps, with an underscore between the root and the extension. Some people use a leading underscore also.

## Compiling and Linking the Program

When a program consists of a collection of files, some of which are header files and their corresponding implementation files, and of course a single file containing the `main()` function, it is compiled and linked in a specific way.

Assume the project contains the files `f1.h`, `f2.h`, `f3.h`, `f1.cpp`, `f2.cpp`, `f3.cpp`, and `main.cpp` and that `main.cpp` includes all header files, but that the remaining `.cpp` files include only their own header files.

The set of steps that must be taken if this is to be done manually, using `g++`, for example, is

```
g++ -c f1.cpp
```



```
g++ -c f2.cpp
g++ -c f3.cpp
g++ -c main.cpp
```

This creates the object files `f1.o`, `f2.o`, `f3.o`, and `main.o`. Then the main program would be created by linking these together, using

```
g++ -o progname main.o f1.o f2.o f3.o
```

The compiler, `g++`, will not compile in this last step; it will just link the files and create the program named `progname`.

If you edit the file `f2.cpp`, you only need to do two steps to rebuild the program:

```
g++ -c f2.cpp
g++ -o progname main.o f1.o f2.o f3.o
```

because only the object file `f2.o` must be changed, and the main program needs to be relinked to it. If you edit a header file, such as `f3.h`, then you would also do two steps:

```
g++ -c f3.cpp
g++ -o progname main.o f1.o f2.o f3.o
```

because presumably `f3.cpp` depends on `f3.h` in such a way that the object file `f3.o` needs to be rebuilt, and then the program relinked.

This can be simplified by using a *makefile*. A makefile is a file that is read by the make program. It contains a set of instructions for carrying out various tasks, usually for building programs. This tutorial on separate compilation does not cover how to create makefiles, but I include one here that could be used to keep the program `progname` up to date whenever any of the files change, with minimal recompilation and linking:

```
CXX      := /usr/bin/g++
CXXFLAGS += -Wall -g
OBJECTS  := main.o f1.o f2.o f3.o
```

```
all: progname
```

```
progname: $(OBJECTS)
<TAB>$(CXX) $(CXX_FLAGS) -o progname $(OBJECTS)
```

```
main.o: f1.h f2.h f3.h
```

```
clean:
<TAB>$(RM) $(OBJECTS)
```

The `<TAB>` means that there should be a tab character in this position. Anything else and `make` will not work. To build the program or update it, one just types "make" on the command line in the directory containing this makefile and the program files. The makefile should be named either `makefile` or `Makefile`.