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.

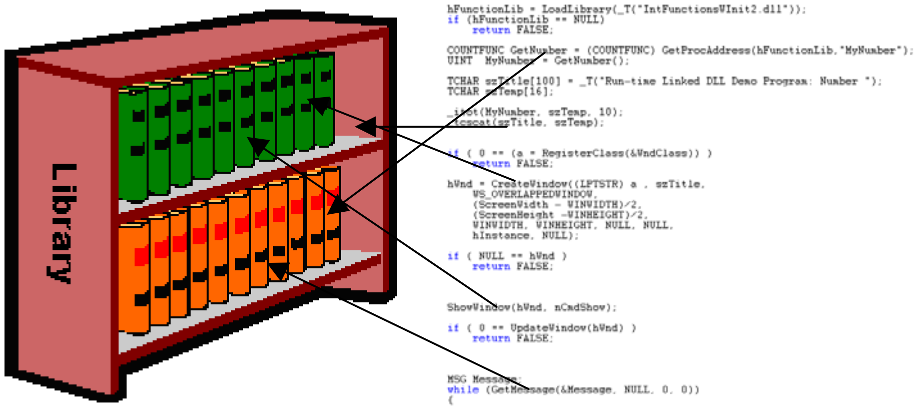
Consider the code fragment

```
#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.



```
hFunctionLib = LoadLibrary(_T("IntFunctionsVInit2.dll"));
if (hFunctionLib == NULL)
    return FALSE;

COUNTFUNC GetNumber = (COUNTFUNC) GetProcAddress(hFunctionLib,"MyNumber");
UINT  MyNumber = GetNumber();

TCHAR szTitle[100] = _T("Run-time Linked DLL Demo Program: Number ");
TCHAR szTemp[16];

_itot(MyNumber, szTemp, 10);
_tcscat(szTitle, szTemp);

if ( 0 == (a = RegisterClass(&WndClass)) )
    return FALSE;

hWnd = CreateWindow((LPTSTR) a , szTitle,
    WS_OVERLAPPEDWINDOW,
    (ScreenWidth - WINWIDTH)/2,
    (ScreenHeight -WINHEIGHT)/2,
    WINWIDTH, WINHEIGHT, NULL, NULL,
    hInstance, NULL);

if ( NULL == hWnd )
    return FALSE;

ShowWindow(hWnd, nCmdShow);

if ( 0 == UpdateWindow(hWnd) )
    return FALSE;

MSG Message;
while (GetMessage(&Message, NULL, 0, 0))
{
```

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.

A large class named `MyNode` would be placed into two files named `mynode.h` and `mynode.cpp`. The `mynode.h` file contains the class interface only. 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 names defined in the header file. The `mynode.cpp` file contains the implementation of the class. The implementation file contains the actual function definitions. The files will usually have the following form.

mynode.h:

```
#ifndef MYNODE_H
#define MYNODE_H

class MyNode
{

// class interface here

};

#endif
```

mynode.cpp:

```
#include "mynode.h"

MyNode::MyNode(...)

// and all implementation code for MyNode member functions here
```

Usually the main program also needs to include the header file for the class, so that it can make reference to class member functions and other parts of the public interface. Therefore, the main program will contain a line of the form

```
#include "mynode.h"
```

among the other header files included by it. 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 also created some other class, say `MyList`, that depends upon `MyNode`, perhaps because your `MyList` object uses `MyNode` objects. Then the `MyList` header file, `mylist.h`, probably has the line

```
#include "mynode.h"
```

also, and the main program has the two lines

```
#include "mynode.h"
#include "mylist.h"
```

When the compiler runs, it copies the `mynode.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 `mynode.h` again because of the #include directive in that file. Now I can explain why the following three funny lines must appear in your header files.

```
#ifndef MYNODE_H
#define MYNODE_H

#endif
```

The first line translates to “if the macro symbol MYNODE_H is not defined then continue reading until the matching occurrence of endif. If it does exist, then skip reading code until immediately after that matching endif”. If there is no symbol already defined, then the next line defines it, and the code is read. By defining the symbol here, the user prevents the macro preprocessor from reading the code reference in the `myheader.h` file twice.

You can use whatever symbol you want, 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.