C++ Separate Header and Implementation Files

C++ classes (and often function prototypes) are normally split up into two files. The header file has the extension of .h and contains class definitions and functions. The implementation of the class goes into the .cpp file. By doing this, if your class implementation doesn't change then it won't need to be recompiled. Most IDE's will do this for you – they will only recompile the classes that have changed. This is possible when they are split up this way, but it isn't possible if everything is in one file (or if the implementation is all part of the header file).

Simple example:

File: Num.h

```
class Num
{
 private:
  int num;
 public:
  Num(int n);
  int getNum();
};
```

File: Num.cpp

```
#include "Num.h"

Num::Num() : num(0) { }
Num::Num(int n): num(n) {}
int Num::getNum()
{
 return num;
}
```

File: main.cpp

```
#include <iostream>
#include "Num.h"

using namespace std;

int main()
{
  Num n(35);
  cout << n.getNum() << endl;
  return 0;
}
```

To compile this from the command line we would use:

```
      g++ main.cpp Num.cpp
```

Note the include statements; we must include the “Num.h” from anywhere we use it.

Using #ifndef

Sometimes we can end up including a header file multiple times. C++ doesn’t like this if it ends up redefining something again. In our previous example this didn’t happen because main.cpp and Num.cpp are compiled separately so the inclusion of Num.h causes no problem. But consider if we had another class in main that uses Num:

File Foo.h (kept about as simple as possible, but not good practice to make Num public)

```
#include "Num.h"
class Foo
{
 public:
  Num n;
};
```

We have no Foo.cpp since there is nothing to implement…

Now in main:

```
#include <iostream>
#include "Num.h"
#include "Foo.h"

using namespace std;

int main()
{
  Num n(35);
  cout << n.getNum() << endl;

  Foo f;
  cout << f.n.getNum() << endl;

  return 0;
}
```

If we try to compile this we now get the error:

```
In file included from Foo.h:1:0,
                 from main.cpp:3:
Num.h:1:7: error: redefinition of ‘class Num’
In file included from main.cpp:2:0:
Num.h:1:7: error: previous definition of ‘class Num’
main.cpp: In function ‘int main()’:
main.cpp:13:13: error: ‘class Foo’ has no member named ‘num’
```

To fix this we can use #ifndef. This is called a **directive** as it is a message to the compiler, but not really a feature of the language. It tells the compiler to ignore what follows if it has already seen this stuff before. The format looks like this:

```
#ifndef NUM_H
#define NUM_H
<define class or whatever else>
#endif
```

This says if "NUM_H" is not defined, then define it. So subsequent attempts to read this class result in skipping the definition since NUM_H is already defined. If we add this to our Num.h class then it will now compile and run.

You can use any name but the recommendation is to use a name related to the class.

The `#pragma once` directive

The same functionality as #ifndef can be accomplished by adding #pragma once to the top of your file. This is the default used with Visual Studio.

Separate Compilation

With what we've done so far we split the header from the implementation. We're actually still compiling both all the time though when we run the g++ command. To really get separate compilation we need to:

1. Compile each .cpp file into an object file, which contains machine code for that file
2. Link each object file into an executable

To compile into an object file, you can use the command flag of –c:

```
g++ -c main.cpp Num.cpp
```

This produces a file main.o and Num.o. We can then link them. We can use g++ again to do this:

```
g++ main.o Num.o
```

We now have our familiar a.out program which we can run. An efficiency step is possible here because if Num never changes but main does change, then we can compile just main via:

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

Then we can link them again but we didn't have the overhead of compiling Num.cpp:

```
g++ main.o Num.o
```

With our little programs this saves virtually no time, but for really large programs this can be a significant savings in compile time. All of this stuff is done for you automatically by an IDE (which is one reason why they are great).

The make utility

It can also be a pain to compile lots of files when you have a big project. Once again, this functionality is provided for you as part of an IDE. But you have to do it yourself if you are compiling from the command line. Well, there is a solution in the **make** utility. If you run the command "make" then this program will look for a file named "Makefile" that contains information about program dependencies and what needs to be compiled.

Here is an example for our particular setup:

```
CFLAGS = -O
CC = g++

NumTest: main.o Num.o
        $(CC) $(CFLAGS) -o NumTest main.o Num.o

main.o: main.cpp
        $(CC) $(CFLAGS) -c main.cpp

Num.o: Num.cpp
        $(CC) $(CFLAGS) -c Num.cpp

clean:
        rm -f core *.o
```

This says that NumTest depends on main.o and Num.o and the second line says how to compile it. In turn, main.o is created by compiling main.cpp with the –c flag. We can compile by typing:

```
make
```

or

```
make NumTest
```

We can use the target of "clean" to delete object files and core dumps.

Only files that have changed are recompiled. Here is a similar version using some more advanced features of Makefiles:

```
CFLAGS = -O
CC = g++
SRC = main.cpp Num.cpp
OBJ = $(SRC:.cpp = .o)

NumTest: $(OBJ)
        $(CC) $(CFLAGS) -o NumTest $(OBJ)

clean:
        rm -f core *.o
```