How to Write a Program in Many Files

A class in a C^{++} programs should always be defined in two files, which have the same name as the class.

1. The first file should be called **class.h**. It should define the members of the class, and declare the member functions.

   In addition, it is acceptable to implement some small member functions in the **class.h** file. The **class.h** file is called the header file of **class**.

2. The second file should be called **class.cpp** or **class.cc**. It should contain the definitions (implementations of most of the member functions.

Separation between header files and implementation files

There are two reasons for the distinction between **.h** and **.cpp**:

1. It enables separate compilation. A big program can be spread over many files. When something changes in the implementation of one of the classes, only its corresponding **class.cpp** file needs to be recompiled.
2. It allows a user of the class to read the specification of a class, without having to understand how it is implemented.

Role of the .h File

A large program consists of many **.cpp** files. The compiler compiles each of these **.cpp** files separately and creates a **.o** file for each of them.

Each of the files may use functions and classes that are defined in another file. The compiler cannot fill in such references, so they are left open.

In order to obtain an executable program, the cross references have to be filled in. The linker collects all the **.o** files, fills in the cross references, and constructs an executable file.

Technical Reason for the .h file

Suppose in file **bbbb.cpp**, we have a reference to a class **aaaa**.

```
aaaa a(44);
int i = a. getint( );
std::cout << a << "\n";
```

When compiling **bbbb.cpp**, the compiler needs to know that

1. Class aaaa exists, how much space it needs.
2. Class aaaa has a constructor that can construct from **int**.
3. Class aaaa has a method **getint( )** which returns an **int**.
4. Class aaaa can be printed with **<<**.

Exactly these things are defined in the **.h** file.

Usage of .h File

The **.h** file exists for technical reasons, but you should use it for separating interface from implementation.

If you put comments in your file (which you should, so this is not an 'If'), the comments must be separated accordingly. Comments about specification and usage of the class go into the **class.h** file.

Comments about implementation go into the **class.cpp**, to the method to which they belong.

The **.h** File

```
// This is the date.h header file. Normally, such obvious
// things should not be written in a comment, but this
// is the first time.

#ifndef DATE_INCLUDED
#define DATE_INCLUDED  1
   // This is called the include guard. Its purpose is
   // to avoid that the same file is included twice.
   //
   // As far as I can see, nothing bad happens when a
   // header file is included twice, but it is
   // inefficient.
```

```
    class date
    {
       unsigned int day;
       unsigned int month;
       unsigned int year;
    };

#endif
```

Include Guards

If you define a class **complex** for complex numbers, and a class **complexmatrix**, which defines matrices over complex numbers, then you have:

1. A main file, which includes `complex.h` and `complexmatrix.h`.
2. A file `complexmatrix.h`, which includes `complex.h`.
3. A file `complexmatrix.cpp`, which includes `complexmatrix.h`.
4. A file `complex.cpp`, which includes `complex.h`.

Nested include chains can be much longer. It is difficult to avoid that the same file gets included twice. Include guards prevent this.

The .cpp File

In file **class.cpp**, you define the methods that were defined in file **class.h**. The file must include **class.h**, and possibly other **.h** files.

Function definitions have form

```
int class::method( )
{


}

int class::method( ) const
{



}
```

If you define a method, that was not defined in `class{ }`, the compiler will complain.

1. A class (and a class method) can be declared as often as you want. It can be defined only once.
2. If you define a method through `aaaa::method( )`, there must have been a declaration of form
   `class aaaa { method( ) };` before it.

Include files are expanded by the preprocessor. The compiler does not see the include files.

Separate Compilation

Compile without Linking:

```
g++ -c -o file file.cpp
```

The `-c` option tells the compiler that it should not create an executable, but a `file.o`, which may possibly contain undefined references.

The command

```
g++ -o progname  file1.o file2.o file3.o
```

calls the linker, which puts everything together and creates an executable callsed **progname**.

Adapting Make to Separate Compilation

In order to define a project consisting of many files, adapt **Makefile** as follows:

```
prog : file1.o file2.o file3.o
     g++ -o prog file1.o file2.o file3.o

file1.o : file1.cpp  aa.h bb.h cc.h
   g++ -c -o file1.o file1.cpp

// Similar definitions for file2.o file3.o
```

Each line of form `a : b1 b2 b3` specifies that `a` is obtained from `b1 b2 b3`. Each next line specifies what should be done. Each action line has to start with a TAB!

Static Members

A static class field has the following features:

- It is shared by all objects of the class, it exists independent of the objects of the class.
- It can be accessed without class object. If the class is called `A`, and the field is called `f`, then it can be addressed by `A::f`.
- It can be viewed as a global variable, whose name happens to be preceded by `A::`.

Static Initialization Order Fiasco

Static members are initialized before `main( )` is called. Inside a file, static members are initialized in the order in which they occur in the file.

Between different files, initialization order is not fixed.

This may cause problems when static members depend on each other.

The example that I always encounter consists of an identifier class, which has a static lookup table, (which maps all identifiers to integers), and some other classes (for example formulas) that have reserved identifiers (which are also static members).

It is not guaranteed that the lookup table is initialized before the reserved identifiers are created.

Be careful with static member initialization! (In my view, it is the biggest defect of C^{++})

Static Local Variables

In a function, it is possible to define static variables. The variable is initialized when control passes through the declaration for the first time. After that, the variable will kept until the program terminates.

```
void count_calls( )
{
   static unsigned int i = 1;
   std::cout << "called for the " << i << "-th time\n";
   ++ i;
}
```

Static Member Functions

A static member function of a class

- Can be called without class member, in the same way as a static field, for example `classname::membername( )`.
- Can use only static fields, and can call only static class members.

Technically, static member functions and static fields can be viewed as having nothing to do with their class. They just happen to be in the namespace of the class.

Static Local Variables

The Static Membership Initialization Fiasco can be solved by putting static fields in static member functions.

```
static std::string& myclass::getfield( )
{
   std::string s = "Initializer!\n";
   return s;
}

myclass::getfield( ) = "today is tuesday!\n";
```

Public/Private

Public fields and member functions can be used from everywhere. Private fields and member functions can be used only in member functions of the class.

Keyword `private:` makes all member functions and fields that come after it private.

Keyword `public:` makes all member functions and fields that come after it public.

A `struct` starts in public mode. A `class` starts in private mode.

Friends

Friends are functions or classes that can use the private fields and functions.

Friends must be declared in the class:

```
class A
{

   friend class B;
   friend A Atransformer( A );
   friend std::ostream& operator << ( std::ostream& ,
                                      const A& );
      // Happens often. Alternative is to define a
      // print( std::ostream& ) const
      // member function, and to have << call
      // this function.
```

Operators

All operators in C^{++} can be extended to user defined types.

Operators can be defined either as member, or as non-member function.

Defining Operator as Member

- No problems with accessing private members. Natural for assignment-like operators: `=`, `-=`, $=$, $+$.
- Often ugly due to asymmetry. (for example `<`,`>`,`<=`,`>=`,`!=`,`=`.)

  In such cases, one can define the operators as non-members, but have them call a static member function.

Defining Operators as Members: Declarations

In the declaration, one can write:

```
class X
{
   void operator = ( const X& x );
   X& operator = ( const X& x );
      // Binary assignment, 2 variants.

   X operator - ( ) const;
      // Unary - .

   X operator - ( const X& x ) const;
      // Binary - .

   X& operator -= ( const X& x );
```

```
    bool operator < ( const X& x );
        // Binary operator < .
};
```

The forms of the parameters (const/nonconst, ref/value) and their types can be freely chosen. Don't define crazy operators!

Defining Operators as Members: Definitions

```
void X :: operator = ( const X& x ) { ... }
X& operator = ( const X& x ) { ... }
X X :: operator - ( ) const { ... }
X X :: operator - ( const X& x ) const { ... }
X& X :: operator -= ( const X& x ) { ... }

bool X :: operator < ( const X& x ) { ... }
```

Defining Operators as Non-members

```
class X
{

...
};

X operator - ( const X& x );
   // Unary -.
X operator - ( const X& x1, const X& x2 );
   // Binary - .

bool operator < ( const X& x1, const X& x2 );
   // Comparison.
```

Inlining

Inline functions are substituted at compile time, instead of called.

- Class member functions that are defined inside the class are inlined by default.
- Class member functions that are defined outside the class must be preceded by word **inline**. This means that the **.h** contains both the declaration and the definition.

Inline only short functions.

If you define a function in a **.h** file, and it is not inline, the linker will complain about multiple definitions.

Anonymous Namespace

If you want to define a helper function in a **.cpp** file, then you do not want the name of the function passed to the linker. Do not use the **static** keyword for this! Instead, use anonymous namespace:

```
namespace
{
   int myfunction( int x );
      // Will not be passed to linker.
}

void otherfunction( ) // Will be passed to linker
{
   x = myfunction(4);
}
```

Anonymous Namespace

Advantage: You don't have to think long about names for local functions, because their names will not conflict with other names.

Disadvantage: Testing is difficult, because you cannot call the local functions from **main**.