**Chapter 13: Multifile Programs**

Table of Contents

[Reasons for Multifile Programs 4](#_Toc49599345)

[Class Libraries 4](#_Toc49599346)

[Interface 5](#_Toc49599347)

[Implementation 5](#_Toc49599348)

[Organization and Conceptualization 7](#_Toc49599349)

[Creating a Multifile Program 8](#_Toc49599350)

[Header Files 8](#_Toc49599351)

[Directory 8](#_Toc49599352)

[Projects 9](#_Toc49599353)

[Inter-File Communication 10](#_Toc49599354)

[Communication Among Source Files 10](#_Toc49599355)

[Inter-File Variables 10](#_Toc49599356)

[Inter-File Functions 14](#_Toc49599357)

[Inter-File Classes 15](#_Toc49599358)

[Header Files 17](#_Toc49599359)

[Common Information 17](#_Toc49599360)

[The Multiple-Includes Hazard 19](#_Toc49599361)

[Preventing Multiple Includes 20](#_Toc49599362)

[Namespaces 21](#_Toc49599363)

[Defining a Namespace 21](#_Toc49599364)

[Access Namespace Members 22](#_Toc49599365)

[Namespaces in Header Files 23](#_Toc49599366)

[Multiple Namespace Definitions 23](#_Toc49599367)

[Unnamed Namespaces 24](#_Toc49599368)

[Renaming Types with typedef 24](#_Toc49599369)

In this chapter, we will be looking at program organization involving multiple files. We will look at how the files communicate with each other, and how header files are involved.

## Reasons for Multifile Programs

There are many reasons to use multifile programs, including the use of class libraries, the organization of programmers working on a project and the conceptual design of a program.

### Class Libraries

In traditional procedure-oriented languages, it has been customary for a long time for software vendors to make libraries of functions. Other programmers then combine these libraries with their own custom-written routines to make applications for end users.

Libraries provide ready-made functions for a wide variety of fields. For example, a vendor may provide a library of functions to handle statistics calculations, or one for advanced memory management.

Since C++ is organized around classes instead of functions, the libraries for C++ programs also consist of classes. A class library is far superior to a function library, since classes contain both data and functions. They also more closely model objects in real life, which makes the interface between a class library and the application which uses it much cleaner. Thus, a class library can take a larger portion of the programming burden. If the right class library is available, only a minimal amount of programming may be required to create a final product. As more and more class libraries are created, it is also very likely that one that solves a specific programming problem is available.

A class library usually consists of two components: the interface and the implementation. We will look at these next.

#### Interface

Let the person who wrote the class library be called the class developer, and the person who uses the library be called the programmer. To use the class library, the programmer needs access to various declarations, including class declarations. These declarations can be thought of as the public part of the library and are usually furnished in source-code form as a header file, which is included with a #include statement.

The declarations in a header file need to be public for several reasons. First, it is convenient for the client to be able to see the actual class definitions instead of a description of them. More importantly, the programmer will need to declare objects based on these classes and call member functions from these objects. This is only possible by declaring the classes in the source file.

The declarations are called the interface, since that is what the programmer sees and interacts with. The programmer does not need to be concerned about the implementation.

#### Implementation

Class developers do not want to release source code, least it be illegally modified or pirated. Programmers do not need to know about the inner workings to be able to use the library. As such, member functions, except for short inline functions, are often distributed in object form, as .obj files, or as library (.lib) files.

#inclue

THEIRS.H

MINE.CPP

MINE.H

THEIRS.OBJ

MINE.OBJ

THEIRS.LIB

MINE.EXE

#inclue

Vendor-Supplied

Compiler

User-Written

Possible

Library File

Executable File

Linker

Header File

Source File

User-Written

Header File

Possible

Vendor-Supplied

Vendor-Supplied

Object File

### Organization and Conceptualization

Programs may need to be broken into multiple files for other reasons too. A common situation is when a project involves multiple programmers. Confining each programmer’s work to a separate file helps organize the project and cleanly define the interface among different parts of the program.

Often, a program is also separated into different files based on functionality. For example, one file might handle the code involved in a graphics display, while another handles mathematical analysis and a third handles disk I/O. In large programs, a single file might simply become too large to handle conveniently.

## Creating a Multifile Program

Suppose we have a pre-written class file that we purchased, theirs.obj, which we want to include in our own source file mine.cpp. .lib files would be handled in a similar manner.

### Header Files

The pre-written class file will most likely come with a header file, theirs.h. This can be easily incorporated into our own source file using a #include statement.

#include "theirs.h"

C++

Notice that quotes are used instead of angle brackets. This tells the compiler to look for the file in the current directory instead of the default include directory.

### Directory

All the component files, include our own source code and the pre-written class file and its header file should be in the same directory. Separate projects should have their own directories to avoid confusion, though this is not, strictly speaking, mandatory.

Each compiler keeps its own library files (the non-custom ones we have used so far such as iostream) in a particular directory inside the compiler’s directory. The compiler already knows where these files are.

It is also possible to tell the compiler about include directories that we have created ourselves, to keep some header files to be available to multiple projects. It is possible to tell the compiler where such a directory is located.

### Projects

Most compilers manage multiple files as a ‘project’. A project contains all the files necessary for an application. It also contains the instructions for combining the files, often in a special project file, the extension of which varies by compiler. Modern compilers construct and maintain this file automatically, so we do not need to worry about it. In general, we need to tell the compiler about all the source files we will be using, so that they can be added to the project. .obj and .lib files can be added in a similar way.

Only a single command is needed to tell the compiler to compile all the source files and link the resulting object files into a final .exe file. This is called the build process. A project keeps track of the dates when each source file was compiled. Only the source files that were modified since the build are recompiled, which can save a lot of time on larger projects. Some compilers additionally have a ‘make’ command to achieve this, whereas the normal ‘build’ command compiles all files regardless of date.

## Inter-File Communication

In a multifile program, program elements in different files need to communicate with each other. Here, we will see how separately-compiled source files that are linked together are able to communicate, and where header files come into the picture.

### Communication Among Source Files

We will be looking at three kinds of programming elements when discussing how source files communicate: variables, functions and classes. Each has its own rules for inter-file use.

The idea of scope will be important. Remember that scope is the region of a program where a variable or other program element can be accessed. Elements declared inside functions have local scope, meaning they are only visible within the function body. Similarly, class members are only visible within the class, unless the scope resolution operator is used. Program elements declared outside of any function or class have a global scope. They can be used throughout an entire file, following the point where they are defined, and are also visible to other files.

### Inter-File Variables

Recall the distinction between a declaration and a definition. Something is declared simply by giving it a name and a type. This does not provide the physical location in memory for the variable, but simply tells the compiler that a variable with this name and type may exist somewhere. A variable is defined, on the other hand, when it is given a place in memory that can hold the variable’s value.

Most declarations are also definitions. In fact, the only way to declare a simple variable that is not a definition, is by explicitly using the extern keyword.

int someVar; *// declaration + definition*extern int someVar; *// declaration only*

C++

As you might expect, a global variable can only be declared in one place in an entire program.

*// file A*int globalVar; *// definition in file A*

C++

*// file B*int globalVar; *// same definition in file B; illegal*

C++

Of course, variables of other types can be declared as many times as we want, given that each declaration is in a different class or function.

A global variable defined in one file is not automatically visible in another file. As such, a global variable must be declared (but not defined) in every file in which it is to be used.

*// file A*int globalVar; *// defined*

C++

*// file B*extern int globalVar; *// declared*globalVar = 3;

C++

Essentially, the compiler is being told to not worry about globalVar being undefined here, since the compiler is not able to see the other file. The linker, which can see all the files, takes care of connecting a reference to a variable in one file to its definition in another.

A surprising restriction is that a declaration alone cannot simultaneously be used to initialize a variable. It will cause the compiler to think we meant to define the variable as well and will ignore the extern keyword. This can cause an error if the variable is defined elsewhere.

extern int globalVar = 27; *// will cause definition instead of declaration*

C++

If we want to restrict the scope of a global variable to the file in which it is defined, and thus allow other global variables in other files to have the same name, we need to use the static keyword.

static int gobalVar;

C++

Static variables are said to have ‘internal linkage’, while non-static global variables have ‘external linkage’.

In multifile programs, it is a good idea to make global variables static whenever they are not accessed in other files. This prevents errors where the same name is used by mistake in another file. It also makes it clearer that we do not need to worry about this variable appearing anywhere else.

Notice that static has different meanings depending on whether it is used with a local or a global keyword. We have seen earlier that using static with a local variable causes the lifetime of the variable to become the same as that of the program, while keeping its visibility restricted to the function. When used with a class member, static causes that member to have the same value for every object of the class.

A const variable is, by default, not visible in other files. If it does need to be visible, the extern keyword must be used with both the definition and the declaration in the other file.

*// file A*extern const int conVar = 99; *// definition*

C++

*// file B*extern const int conVar; *// declaration*

C++

The difference between a const declaration and a const definition is identified by the compiler using the point where the variable is initialized.

### Inter-File Functions

A function declaration specifies the name of the function, its return type and the type of any arguments. A function definition is a declaration that includes a function body.

When a compiler generates a call to a function, it does not need to know how the function works. It simply needs its name, return type and the types of its arguments. As such, a function can easily be defined in one file, and called from a second file without using any extra keywords like we had to do with global variables. All that is needed is that the function be declared in the second file before it is called.

*// file A*int add (int a, int b) *// function definition*{ return a + b; }

C++

*// file B*int add (int, int); *// function declaration*int answer = add (3, 2); *// function call*

C++

Incidentally, we can declare a function (without defining it) as many times as we want, as long as the declarations are all the same.

Like variables, functions can be made invisible to other files using the static keyword.

static int add (int a, int b)  
{ return a + b; }

C++

### Inter-File Classes

Classes are unlike simple variables, in that a class definition does not set aside any memory. It only informs the compiler what members constitute the class (and thus how much memory will be required). A class definition contains declarations and definitions for all its members.

class someClass *// class definition*{  
private:  
 int memVar; *// member data definition*public:  
 int memFunct(int, int); *// member function declaration*};

C++

Members must be declared but do not need to be defined inside a class definition. As we have seen before, member functions are regularly placed outside the class and identified using the scope resolution operator.

A class declaration is simply a statement that a certain name belongs to a class. It does not provide any information about the members of the class to the compiler.

class someClass; *// class declaration*

C++

Note that this is different from the definition of an object of the class. Unlike a class declaration or definition, an object definition does set aside space in memory for the object.

In inter-file communication, classes behave differently from variables or functions. To access a class across multiple files, it must be defined (not just declared) in each of the files. Simply declaring a class in one file that is defined in another file does not allow us to create objects of the class in the prior file. This is because the compiler needs to know the data type of everything it is compiling. A declaration is all that is needed to achieve this for simple variables, since the compiler already knows the data type, and for functions, since the declaration provides the required information. However, for a class, the entire definition is needed to specify the types of its member data and functions. The only exception to this is when pointers or references to objects are used.

Thus, a class can only be defined once in a single source file, but can be defined in multiple source files. A class definition can be supplied to many files using header files, which we will look at next.

## Header Files

The #include preprocessor causes the text from one file to be inserted into another. We have seen many examples of library files such as iostream being included in our source files. It is also possible to write our own header files and include them in our source files.

### Common Information

One reason to use a header file is to supply two or more source files with the same information. The header file holds variable or function declarations, and is included in the source files. As such, the variables and functions can be accessed from many files.

Of course, each element must also be defined somewhere. Here, we declare a variable and function in a header file, define them in a source file and use them in another source file.

*// fileH.h*extern int gloVar; *// variable declaration*int gloFunc (int); *// function declaration*

C++

*// fileA.cpp*int glover; *// variable definition*int gloFunc (int n) *// function definition*{ return n; }

C++

*// fileB.cpp*#include "fileH.h"  
gloVar = 5;  
int gloVarB = gloFunc (gloVar);

C++

Beware that we cannot put variable or function definitions in a header file that will be shared by multiple source files, unless they are static or const). If we do, the same definition will end up in two difference source files and the linker will show a ‘multiply defined’ error.

A common (and almost essential) technique is to put a class definition in a header file that is included in every source file that needs it. This does not cause a multiply defined error, since class definitions do not set aside any memory.

Why put class definitions in header files at all? Why not just copy the text of the class definition and paste it in every source file manually? Well, this would work, but any modifications to the class would mean the definition in each file needs to be changed separately. This would be time-consuming and error-prone.

The external member function definitions for a class can be put in any source file, and the linker will connect them as needed. Just like in a single file, the external definition must include the class name and scope resolution operator.

*// fileH.h*class someClass *// class definition*{  
private:  
 int memVar;  
public:  
 int memFunc (int, int); *// member function declaration*};

C++

*// fileA.cpp*#inlcude "fileH.h"  
int someClass::memFunc (int n1, int n2) *// member function definition*{ return n1 + n2; }

C++

*// fileB.cpp*#include "fileH.h"  
someClass anObj;  
int answer = anObj.memFunc (6, 7);

C++

### The Multiple-Includes Hazard

We have mentioned how you cannot define a function or variable in a header file that is shared by multiple source files, since it causes multiple-definition errors. A similar problem occurs when the same header file is included twice in a source file. This might seem like an impossible mistake to make, but suppose you include two header files, headerOne.h and headerTwo.h in a source file, but headerTwo.h was already included in headerOne.h. This would cause exactly this error to occur.

#### Preventing Multiple Includes

To avoid the error highlighted above, we precede the definitions in the header file with the following pre-processor directive:

#if !defined (someHeaderFile)  
#define someHeaderFile  
*// contents of header file*#endif

C++

Essentially, this says that if the header file is not already defined in the source file, it should first be defined and then the contents of the header file be pasted into the source file, right up to the closing #endif. If it has already been defined (by a previous #include statement), this will be ignored.

An older directive, #ifndef, was used the same way as #if!defined(), and can be spotted in some header files, but its use is now discouraged.

The things shown above are called header guards, and should be used whenever there is even a possibility of a header file being included in a source file multiple times. Note that this does not prevent a global variable declared in the header file from being included in different source files, which will still cause the linker to show an error.

## Namespaces

We have seen how to restrict the visibility of program elements by declaring them within a file or class or by making global elements static or const. However, a more versatile approach may be required. For example, when writing a class library, a programmer may way to use short, common names for non-member functions and classes, like add() or book. However, these names will most likely also be used by the creators of other libraries or the applications that use the library. This leads to name clashes and multiple definition errors. Before namespaces, programmers were forced to use long names to avoid this problem. These were difficult to read and write and took up excessive space in a listing. Namespaces solve this problem.

Note that member functions do not cause name clashes, since their scope is limited to the class.

### Defining a Namespace

A namespace is a section of a file that is given a name and has some declarations inside it.

namespace someNamespace  
{  
 const double PI = 3.14159;  
 double circumf (double radius)  
 { return 2 \* PI \* radius; }  
} *// end namespace someNamespace*

C++

Variables and other program elements declared within the braces are called namespace members. Notice that there is no semicolon following the closing brace as there is with classes.

### Access Namespace Members

Code outside a namespace cannot normally access the elements within the namespace. If we want to use those elements, we need to invoke the namespace name when referring to them. There are two ways to do this. The first way is to precede each element’s name with the namespace name and the scope resolution operator:

double c = someNamespace::circumf(10);

C++

The second way is to use the using directive.

using namespace someNamespace;  
double c = circumf(10);

C++

The using directive causes the namespace to be visible from that point onwards. However, the region in which it is in effect can be restricted to a particular block, such as a function.

void someCalcs()  
{  
 using namespace someNamespace;  
 double c = circumf(10);  
}  
double c = circumf(10); *// will not work here*

C++

### Namespaces in Header Files

Namespaces are most commonly used in header files containing library classes or functions. Each such library can have its own namespace. We have been regularly using the std namespace, whose members constitute the Standard C++ Library.

### Multiple Namespace Definitions

The same namespace can be defined several times. This will work like a continuation of the same definition, and not a redefinition. This allows the same namespace to be used in several header files, which can then all be included in a source file. There are dozens of header files that use the namespace std.

It is also possible to place declarations outside a namespace that behave as if they were inside it. To do this, the namespace name and the scope resolution operator must be used.

namespace beta  
{  
 int uno;  
}  
int beta::dos;

C++

### Unnamed Namespaces

We can create a namespace without a name. This will cause the namespace to be automatically visible throughout the file in which it is defined, but not from any other files. The compiler gives it an internal name unique to the file.

namespace  
{ int gloVar = 111; }

C++

This provides an alternative to the use of static for restricting the scope of global variables to their own files. This approach is preferable.

## Renaming Types with typedef

The typedef keyword allows the creation of a new name for a data type.

typedef unsigned long unlong;

C++

Variables can then be declared using the new name.

unlong var1, var2;

C++

This may save some space or make listings more readable. More usefully, new type names can be created that reveal the purpose of any variable declared with that type.

typedef int FLAG; *// integer variables that hold flag values*typedef int KILOGRAMS; *// integer variables that hold values in kilograms*

C++

Since classes are types in C++, typedef can also be used to create alternate names for them.

Typedef GeorgeSmith\_Display\_utility GSDu;  
GSDu anObj;

C++

Type renaming is typically handled in header files, so that multiple source files can use the new names. Many software development organizations make extensive use of typedef, which makes the result almost like a different language.