APPENDIX

G Multi-Source File Programs

Programming students normally begin by writing programs that are contained in a single file. Once the size of a program grows large enough, however, it becomes necessary to break it up into multiple files. This results in smaller files that compile more quickly and are easier to manage. In addition, dividing the program into several files facilitates the parceling out of different parts of the program to different programmers when the program is being developed by a team.

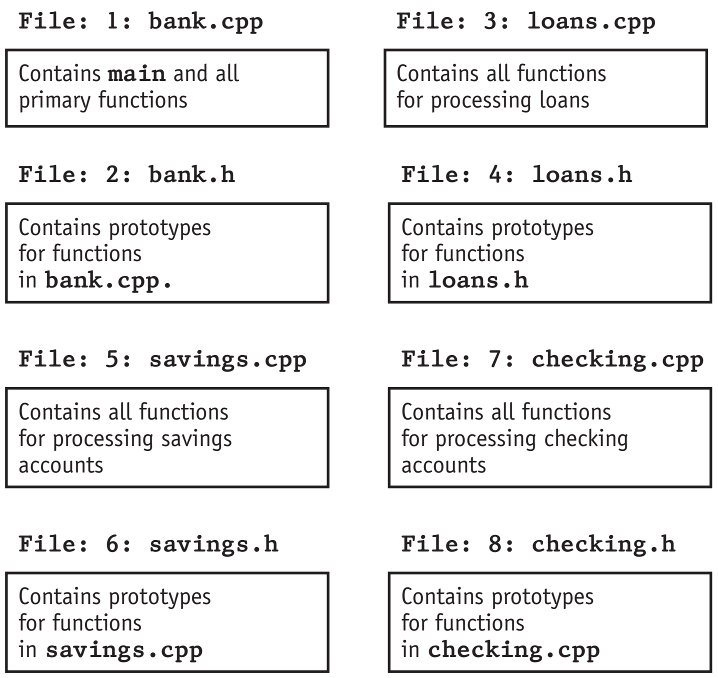
Generally, a multi-file program consists of two types of files: those that contain function definitions, and those that contain function prototypes and templates. Here is a common strategy for creating such a program:

- Group all specialized functions that perform similar tasks into the same file. For example, a file might be created for functions that perform mathematical operations. Another file might contain functions for user input and output.
- Group function `main` and all functions that play a primary role into one file.
- For each file that contains function definitions, create a separate header file to hold the prototypes for each function and any necessary templates.

As an example, consider a multi-file banking program that processes loans, savings accounts, and checking accounts. Figure G-1 illustrates the different files that might be used. Notice the file-naming conventions used and how the files are related.

- Each file that contains function definitions has a filename with a `.cpp` extension.
- Each `.cpp` file has a corresponding header file with the same name, but with a `.h` file extension. The header file contains function prototypes and templates for all functions that are part of the corresponding `.cpp` file.
- Each `.cpp` file has an `#include` directive for its own header file. If the `.cpp` file contains calls to functions in another `.cpp` file, it will also have an `#include` directive for the header file for that function.

Figure G-1


File: 1: bank.cpp
Contains main and all primary functions
File: 3: loans.cpp
Contains all functions for processing loans
File: 2: bank.h
Contains prototypes for functions in bank.cpp.
File: 4: loans.h
Contains prototypes for functions in loans.h
File: 5: savings.cpp
Contains all functions for processing savings accounts
File: 7: checking.cpp
Contains all functions for processing checking accounts
File: 6: savings.h
Contains prototypes for functions in savings.cpp
File: 8: checking.h
Contains prototypes for functions in checking.cpp

Compiling and Linking a Multi-File Program

In a program with multiple source code files, all of the `.cpp` files are compiled into separate *object* files. The object files are then linked into a single executable file. Integrated development environments such as Netbeans, Eclipse, and Microsoft Visual Studio facilitate this process by allowing you to organize a collection of source code files into a *project* and then use menu items to add files to the project. The individual source files can be compiled, and an executable program can be created, by invoking commands on menus.

If you are using a command line compiler such as `gcc`, you can compile all source files and create an executable by passing the names of the source files to the compiler as command line arguments.

```
g++ -o bankprog bank.cpp loans.cpp checking.cpp savings.cpp
```

This command will compile the four source code files and link them into an executable called `bankprog`. Notice that the file listed after the `-o` is the name of the file where the executable will be placed. The following `.cpp` files are the source code files, with the first one listed being the file that contains the `main` function. Notice that the `.h` header files do not need to be listed because the contents of each one will automatically be added to the program when it is `#included` by the corresponding `.cpp` file.

Global Variables in a Multi-File Program

Normally, global variables can only be accessed in the file in which they are defined. For this reason, they are said to have *file scope*. However, the scope of a global variable defined in one file can be extended to make it accessible to functions in another file by placing an `extern` declaration of the variable in the second file, as shown here.

```
extern int accountNum;
```

The `extern` declaration does not define another variable; it just permits access to a variable defined in some other file.

Only true variables, not constant variables, should be declared to be `extern`.

```
const int maxCustomers = 35;      // Don't declare this as extern.
```

This is because some compilers compile certain types of constant variables right into the code and do not allocate space for them in memory, thereby making it impossible to access the constant variable from another file. So how can functions in one file be allowed to use the value of a variable defined in another file while ensuring that they do not alter its value? The solution is to use the `const` key word in conjunction with the `extern` declaration. Thus the variable is defined in the original file, as shown here:

```
string nameOfBank = "First Federal Savings Bank";
```

In the other file that will be allowed to access that variable, the `const` key word is placed on the `extern` declaration, as shown here:

```
extern const string nameOfBank;
```

If you want to protect a global variable from any use outside the file in which it is defined, you can declare the variable to be `static`. This limits its scope to the file in which it is defined and hides its name from other files:

```
static double balance;
```

Figure G-2 shows some global variable declarations in the example banking program. The variables `customer` and `accountNum` are defined in `bank.cpp`. Because they are not declared to be `static` variables, their scope is extended to `loans.cpp`, `savings.cpp`, and `checking.cpp`, the three files that contain an `extern` declaration of these variables. Even though the variables are defined in `bank.cpp`, they may be accessed by any function in the three other files.

Figure G-2

bank.cpp

```
#include "bank.h"
#include "loans.h"
#include "savings.h"
#include "checking.h"
...
     (other #include
     directives)

char customer[35];
int accountNum;

int main()
{
...
}
function1()
{
...
}
function2()
{
...
}
```

loans.cpp

```
#include "loans.h"
...
   (other #include
    directives)

extern char customer[];
extern int accountNum;
static double loanAmount;
static int months;
static double interest;
static double payment;

 function3()
 {
 ...
 }
 function4()
 {
 ...
 }
```

checking.cpp

```
#include "checking.h"
...
     (other #include
     directives)

extern char customer[];
extern int accountNum;
static double balance;
static double checkAmnt;
static double deposit;

function5()
{
...
}
function6()
{
...
}
```

savings.cpp

```
#include "savings.h"
...
   (other #include
    directives)

extern char customer[];
extern int accountNum;
static double balance;
static int interest;
static double deposit;
static double withdrawl;

function7()
{
...
}
function8()
{
...
}
```

Figure G-2 includes examples of static global variables. These variables may not be accessed outside the file they are defined in. The variable `interest`, for example, is defined as a static global in both `loans.cpp` and `savings.cpp`. This means each of these two files has its own variable named `interest`, which is not accessible outside the file it is defined in. The same is true of the variables `balance` and `deposit`, defined in `savings.cpp` and `checking.cpp`.

In our example, the variable `customer` is defined to be an array of characters. It could have been defined to be a `string` object, but it was made a C-string instead to illustrate how an `extern` declaration handles arrays. Notice that in `bank.cpp`, the array is defined with a size declarator

```
    char customer[35];
```

but in the `extern` declarations found in the other files, it is referenced as

```
    extern char customer[];
```

In a one-dimensional array, the size of the array is normally omitted from the `extern` declaration. In a multidimensional array, the size of the first dimension is usually omitted. For example, the two-dimensional array defined in one file as

```
    int myArray[20][30];
```

would be made accessible to other files by placing the following declaration in them.

```
    extern  int myArray[][30];
```

Object-Oriented Multi-File Programs

When creating object-oriented programs that define classes and create objects of those classes, it is common to store class declarations and member function definitions in separate files. Typically, program components are stored in the following fashion.

- **The class declaration—**The class declaration is stored in its own header file, which is called the *class specification file*. The name of the specification file is usually the same as the class, with a `.h` extension.
- **Member function definitions—**The member function definitions for the class are stored in a separate `.cpp` file, which is called the *class implementation file*. This file, which must `#include` the class specification file, usually has the same name as the class, with a `.cpp` extension.
- **Client functions that use the class—**Any files containing functions that create and use class objects should also `#include` the class specification file. They must then be linked with the compiled class implementation file.

These components are illustrated in the following example, which creates and uses an `Address` class. Notice how a single program is broken up into separate files. The `.cpp` files making up the program can be separately compiled and then linked to create the executable program.

```
//*********************************************************************
// Contents of address.h
// This is the specification file that contains the class declaration.
//*********************************************************************
#include <string>
using namespace std;

class Address
{private:
   string name;
   string street;
   string city;

 public:
   Address(string name, string street, string city);
   Address( );
   void print();
};

//**********************************************************************
// Contents of address.cpp
// This is the implementation file that contains the function definitions
// for the class member functions.
//**********************************************************************
#include "address.h"
#include <iostream>

Address::Address(string name_in, string street_in, string city_in)
{
   name = name_in;
   street = street_in;
   city = city_in;
}

Address::Address()
{
   name = street = city = "";
}

void Address::print()
{
   cout << name   << endl
        << street << endl
        << city   << endl;
}
```

```
//***********************************************************************
// Contents of userfile.cpp
// This file contains the client code that uses the Address class.
//***********************************************************************
#include "address.h"

int main( )
{
   // Create an address and print it.
   Address addr("John Doe", "123 Main Street", "Hometown, USA");
   addr.print();

   // Other code could go here.
   return 0;
}
```