APPENDIX I

Multi-Source File Programs

Many of the programs you have written have been contained in one file. Once you start writing longer programs, however, you need to break them into multiple smaller files. Generally, a multi-file program consists of two types of files: ones that contain function definitions, and ones that contain function prototypes and templates. Here is a common strategy for creating such a program:

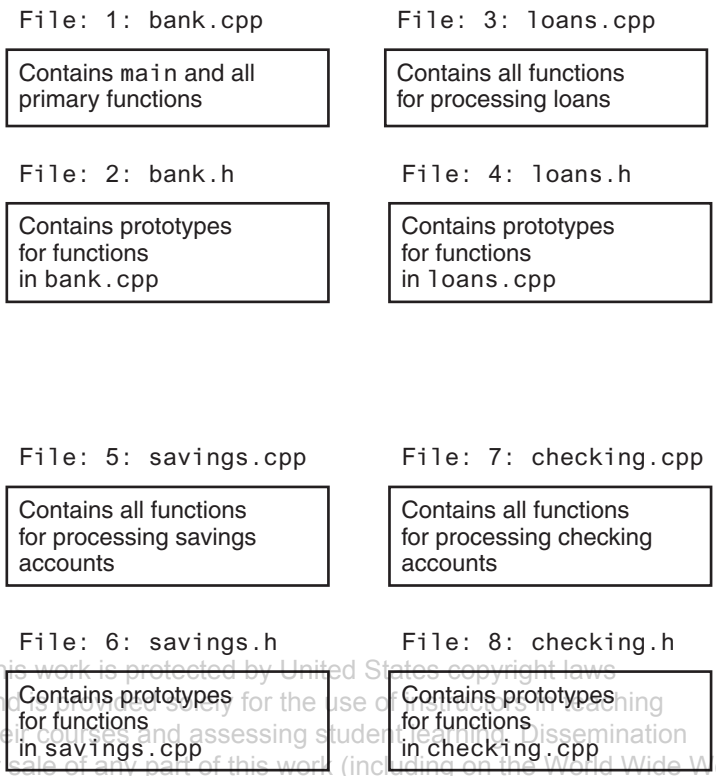
- Group all specialized functions that perform similar tasks into the same files. For example, a file might be created for functions that perform mathematical operations. Another file might contain functions for user input and output.
- Group function `main` and all functions that play a primary role into one file.
- Create a separate header file for each file that contains function definitions. The header files contain 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 I-1 illustrates the different files that might be used.

Each file whose name ends in `.cpp` contains function definitions. Each `.cpp` file has a corresponding header file whose name ends in `.h`. The header file contains function prototypes and templates for all functions that are part of the `.cpp` file. Each `.cpp` file has an `#include` directive that reads its own header file. If the `.cpp` file contains calls to functions in another `.cpp` file, it will have an `#include` directive to read the header file for that function as well.

All of the `.cpp` files are compiled into separate *object* files. The object files are then linked into a single executable file. This process is most easily performed with a *project* or *make* utility. These utilities allow you to create a list of files that make up a multi-file program. The compiler then automatically compiles and links all the necessary components into an executable file. (Check with your compiler documentation for instructions on using its specific utilities.)

Figure I-1



Global Variables in a Multi-File Program

Normally, global variables are only in scope in the file that defines them. In order for a global variable defined in file A to be accessible to functions in file B, file B must contain an `extern` declaration of the variable. This means the keyword `extern` must precede the data type name in the declaration. The `extern` declaration does not define another variable, but extends the scope of the existing variable.

If a global variable is defined as `static`, its scope cannot be extended beyond the file it is defined in. This can be done to ensure that a variable is private to one file, and its name is hidden outside the file it is defined in.

Figure I-2 shows some global variable declarations in the example banking program. The variables `customer` (a character array) and `accountNum` (an `int`) are defined in `bank.cpp`. The scope of these variables is extended to `loans.cpp`, `savings.cpp`, and `checking.cpp` because each file has an `extern` declaration of the variables. Even though the variables are defined in `bank.cpp`, they may be accessed by any function whose file contains an `extern` declaration of them.

Figure I-2

bank.cpp

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

string customer;
int accountNum;

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

loans.cpp

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

extern string 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 string customer;
extern int accountNum;
static double balance;
static double checkAmnt;
static double deposit;

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

savings.cpp

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

extern string customer;
extern int accountNum;
static double balance;
static int interest;
static double deposit;
static double withdrawl;

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

Each file in the example also contains static global variable definitions. 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 file 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`.

Class Declarations

It is common to store class declarations and member function definitions in their own separate files. Typically, program components are stored in the following fashion:

- A class declaration is stored in its own header file, which is called the specification file. The name of the specification file is usually the same as the class, with a `.h` extension.
- The member function definitions for the class are stored in a separate `.cpp` file, which is called the implementation file. The file usually has the same name as the class, with the `.cpp` extension.
- Any program that uses the class should `#include` the class's header file. The class's `.cpp` file (which contains the member function definitions) should be compiled and linked with the main program. This process can be automated with a project or make utility, or an integrated development environment such as Visual C++.