Chapter 14
Multi-Module Programs
(plus pages 311-314 from Chapter 12)

14.1 Chapter Overview

Every program that you have seen so far has been contained in one file that included all the functions that the program used, except for a few pre-defined ones such as `printf()` and `scanf()`, which required the inclusion of standard header files, such as `<stdio.h>`.

With larger programs, editing becomes more complicated and recompiling takes longer. And, many times, large programs are written by a group of people. So, in C, large programs can be broken up into smaller "***modules***", which contain either a single function or a number of related functions that are logically grouped together.

This chapter explains how to edit and compile multiple source files from the command line producing one excecutable file. We will also discuss external variables, #include (from chapter 12) & header files, the *make* utility, and grep.

14.2 Compiling Multiple Source Files from the Command Line

Suppose you have divided your progam into 3 modules: `main.c`, `mod1.c`, and `mod2.c`. To tell the system that these 3 modules actually belong to the same program, you simple include the name of all 3 of these files with gcc to compile them.

```
gcc  mod1.c  mod2.c  main.c  –o  myProg
```

If there are no errors in any of these modules, then the above line will result in an executable file called `myProg`. If there was an error in `mod2.c`, the result of the above compile might look like the following:

```
mod2.c:10  mod2.c:  In function 'foo':
mod2.c:10  error:  'i' undeclared (first use in this function)
mod2.c:10  error:  (Each undeclared identifier is reported only once
mod2.c:10  error:  for each function it appears in.)
```

If this was the case, then after fixing the error(s), you would just need to recompile the modules.

When a program is compiled, there are a few intermediate steps that take place. The preprocessor is run, taking care of things like `#define` statements (the preprocessor is discussed in chapter 12 and elsewhere in the book). The assembler and linker are also run; if there are no errors at this point, then the executable file is created.

In the assembler step, the assembly language version is created, (the .s file). It is these assembly language statements that are then translated into binary code (object code) which then produces the .o file. (Typically, these intermediate files are automatically deleted by the system after the compilation process ends.)

After it reaches this point (compiles with no errors and produces the object code), the linker steps in and links together whatever else is needed, producing the final linked executable file, called `a.out`.

To capture these files you can do the following:

```
gcc  –E  prog.c  >  ppf    - - > this will create a file containing the results from the preprocessor
gcc  -S  prog.c   - - > this will create a .s file with the assembly code (which you can look at)
gcc  -c   prog.c   - - > this will create the .o file with the object code (which isn't anything you can look at)
```

or you can use this one line to create and save all 3 files (.i , .s, & .o files):

```
gcc  -save-temps  prog.c
```

Back to our example of compiling 3 modules: `main.c`, `mod1.c`, and `mod2.c` - - if you were to compile it with the above line (using –save-temps), and there was an error in `mod2.c`, then after fixing the error in `mod2.c`, you could do the following:

```
gcc  mod1.o  mod2.c  main.o  -o  myProg
```

which will only recompile the `mod2.c`, and then if that compiled with no errors, would then link up all the .o files and produce the executable called `myProg`.

With really large programs that consist of many modules, this mechanism of separate compilations can be very useful, especially if only one or two of the modules need to be recompiled – then you don't have to recompile everything.

14.3 External Variables

Functions contained in separate files can communicate through ***external*** variables. An external variable is one whose value can be accessed and changed by another module. Inside the module that wants to access the external variable, the variable is declared in the normal fashion with the keyword `extern` placed before the declaration. This signals to the system that a globally defined variable from another file is to be accessed.

The variable has to be defined in some place among your source files. One way to do this is to declare the variable as you would a global variable, outside any functions, such as:

```
int  moveNumber;
```

It can also be initialized here as well:

```
int  moveNumber  =  1;
```

Then, in the file where this external variable is to be accessed, at the top (or inside the function where it will be used), you would declare it like the following:

```
extern  int  moveNumber;
```

Another way to handle external variables is to define the variable outside of any function using the keyword extern and explicitly assigning an initial value to it, such as the following:

```
extern  int  moveNumber  =  1;
```

Then in any of the other files that use this variable, you would declare it using the `extern` keyword:

```
extern  int  moveNumber;
```

14.4 `Static` vs `Extern` Variables

So, variables declared outside of any function are therefore not only global variables but also external variables. You may come across an instance where you might want a global variable within a module, but not an external variable that may be accessed from other modules. In this case, you would use the keyword `static` when declaring the variable. The following statement:

```
static int  moveNumber  =  1;
```

made outside of any functions would make `moveNumber` a global variable, accessible to any function within the module, but not accessible to anything else outside of the module.

Same goes with functions. When a function is defined, by default it is an `extern` function. You can explicitly use the keyword `extern` when defining a function, but do not need to. Therefore, any function that is defined is accessible from any other module outside of the one it is defined in. If you do not want it to be accessible from functions in other modules, use the keyword `static` when defining the funcion, such as:

```
static double squareRoot (double x)
{
     . . .
}
```

This makes the above function local to the file in which it is defined; it cannot be called from outside the file.

mod1.c

```
double x;
static double result;

static void doSquare(void) {

   double square(void); // prototype

   x = 2.0;
   result = square();
}

int main(void) {
   doSquare();
   printf("%g\n", result);

   return 0;
}
```

mod2.c

```
extern double x;

double square(void) {

   return x * x;
}
```

These files above from the book, page 338, show that `main()` calls `doSquare()`, which in turn calls `square()`. This last function, `square()` is defined in a separate file called `mod2.c`.

Because `doSquare()` is declared `static`, it can only be called from within `mod1.c` and by no other module.

Both `x` and `result` are global variables declared at the top of `mod1.c`. Because `result` is declared `static`, it can only be accessed from within `mod1.c` however `x` can be accessed by any module that is linked together with `mod1.c`.

To compile these two modules, you would type: `gcc -Wall mod1.c mod2.c`

14.5 Header Files & `#include`

In Chapter 12, "The Preprocessor", introduces the concept of the include (or header) file. This is a file where you can group all your commonly used definitions, and then simply include that file in any program that needs those definitions. This is especially useful with large programs that have been divided up into separate program modules.

This also helps with large programs where multiple programmers are working on the program by providing a means of standardization – each programmer is using the same definitions. Common structure definitions, external variable definitions, typedef definitions, and function prototypes can be placed in an include file.

Then each module which refers to something in that include file would need to have the following at the top:

```
#include  "fileName.h"
```

eliminating the need to include the function prototypes or extern declarations, etc. in the modules. The statement must appear before any of the defines contained in `fileName.h` are referenced; usually they are placed at the top right after any other `#include` statements. The preprocessor looks for the specified file and effectively copies the contents of the file into the program at that precise point where the `#include` statement appears. Notice the quotation marks used, instead of the <> - this tells the compiler that it can find the specified `.h` file in the same directory as the file(s) being compiled.

14.6 The `make` Utility

A Makefile is a file that contains a list of files and their dependencies. The `make` program automatically recompiles files only when necessary. The makefile would contain the information for your modules with which file is dependent on which, and a line for the compile command, so when you need to compile the program, you would just type `make` at the Unix command. For example, if you had the 3 files at the top of this document: `main.c`, `mod1.c`, and `mod2.c`, your makefile might look something like the following:

```
SRC  =  mod1.c  mod2.c  main.c
OBJ  =  mod1.o  mod2.o  main.o
PROG  =  myProg

$(PROG) :  $(OBJ)
      gcc $(OBJ) -o $(PROG)

$(OBJ) :  $(SRC)
```

So, when you type `make` at the command prompt, it will automatically compile your modules and produce an executable file called `myProg`. If there is an error in any of the modules, you'll get error messages like before. After you correct the errors, and type `make` again, the makefile will take care of re-compiling only the files that need to be re-compiled.

14.7 The `grep` Utility

The `grep` utility is really useful when you're looking for something in a file but cannot remember which file it's in, (especially when you have a lot of files). The format is:

```
grep  word  fileName
```

so, for example, if you're looking for the variable `todaysDate` in the files that are in your current working directory, you could type:

```
grep  todaysDate  *.c
```

using the wildcard * which says to search all the files that end in .c for the word `todaysDate` in it; if there are any files with that word in it, it will tell you which one. If you just leave the fileName as * it will search ***all*** files in that directory for that word.

If you add a –n, then it will also tell you which line number in the file it is on.

```
grep  -n  todaysDate  *.c
```