Subsections

Writing Larger Programs

This Chapter deals with theoretical and practical aspects that need to be considered when writing larger programs.

When writing large programs we should divide programs up into modules. These would be separate source files. `main()` would be in one file, `main.c` say, the others will contain functions.

We can create our own library of functions by writing a ***suite*** of subroutines in one (or more) modules. In fact modules can be shared amongst many programs by simply including the modules at compilation as we will see shortly..

There are many advantages to this approach:

- the modules will naturally divide into common groups of functions.
- we can compile each module separately and link in compiled modules (more on this later).
- UNIX utilities such as **make** help us maintain large systems (see later).

Header files

If we adopt a modular approach then we will naturally want to keep variable definitions, function prototypes ***etc.*** with each module. However what if several modules need to share such definitions?

It is best to centralise the definitions in one file and share this file amongst the modules. Such a file is usually called a **header file**.

Convention states that these files have a `.h` suffix.

We have met standard library header files already ***e.g***:

```
#include <stdio.h>
```

We can define our own header files and include then our programs via:

```
#include "my_head.h"
```

NOTE: Header files usually ONLY contain definitions of data types, function prototypes and C preprocessor commands.

Consider the following simple example of a large program (Fig. 34.1) .

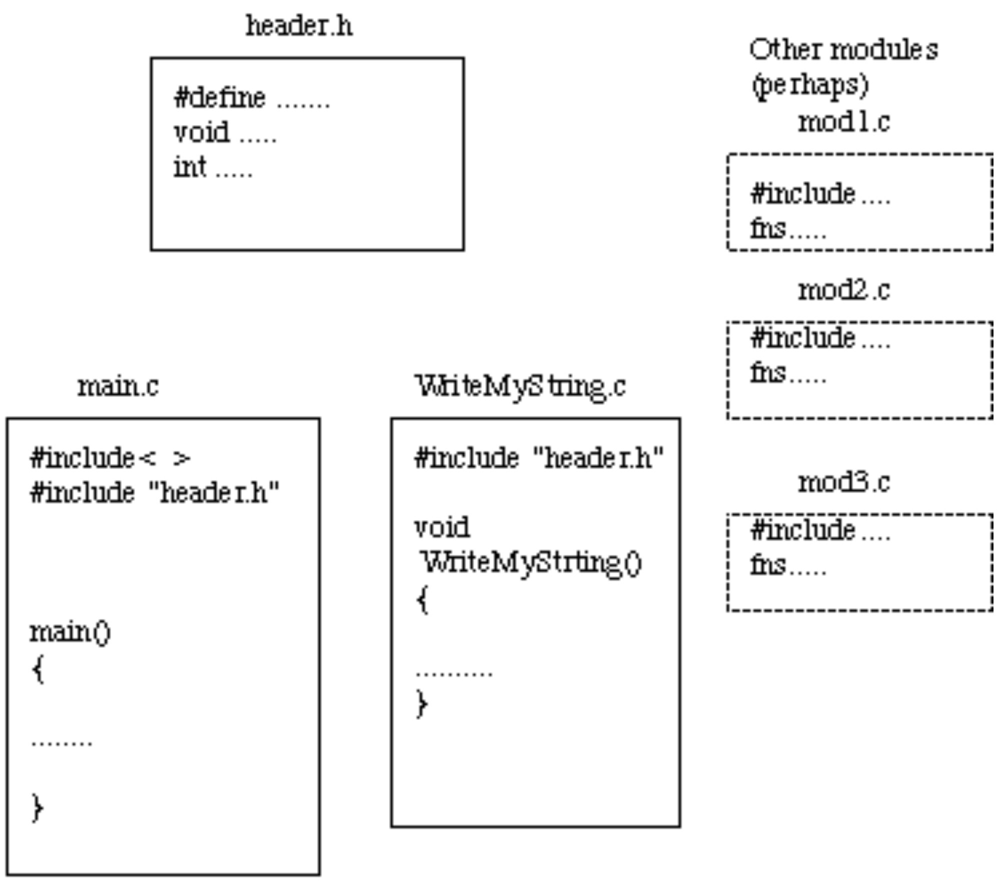

Fig. **Modular structure of a C program** The full listings `main.c, WriteMyString.c` and `header.h` as as follows:

main.c:

```
/*
 *      main.c
 */
#include "header.h"
#include <stdio.h>

char    *AnotherString = "Hello Everyone";

main()
{
        printf("Running...\n");

        /*
         *      Call WriteMyString() - defined in another file
         */
        WriteMyString(MY_STRING);

        printf("Finished.\n");
}
```

WriteMyString.c:

```
/*
 *      WriteMyString.c
 */
extern char     *AnotherString;

void WriteMyString(ThisString)
char    *ThisString;
{
        printf("%s\n", ThisString);
        printf("Global Variable = %s\n", AnotherString);
}
```

header.h:

```
/*
 *      header.h
 */
#define MY_STRING "Hello World"

void WriteMyString();
```

We would usually compile each module separately (more later).

Some modules have a `#include "header.h"` that share common definitions.

Some, like ***main.c***, also include standard header files also.

`main` calls the function `WriteMyString()` which is in ***WriteMyString.c*** module.

The function prototype `void` for `WriteMyString` is defined in ***Header.h***

NOTE that in general we must resolve a tradeoff between having a desire for each `.c` module to have access to the information it needs solely for its job and the practical reality of maintaining lots of header files.

Up to some moderate program size it is probably best to one or two header files that share more than one modules definitions.

For larger programs get UNIX to help you (see later).

One problem left with module approach:

SHARING VARIABLES

If we have global variables declared and instantiated in one module how can pass knowledge of this to other modules.

We could pass values as parameters to functions, BUT:

- this can be laborious if we pass the same parameters to many functions and / or if there are long argument lists involved.

- very large arrays and structures are difficult to store locally -- memory problems with stack.

External variables and functions

"Internal" implies arguments and functions are defined inside functions -- **Local**

"External" variables are defined outside of
functions -- they are potentially available to the whole program (Global) but **NOT necessarily**.

External variables are always permanent.

NOTE: That in C, all function definitions are external. We CANNOT have embedded function declarations like in PASCAL.

Scope of externals

An external variable (or function) is not always totally global.

C applies the following rule:

The scope of an external variable (or function) begins at its point of declaration and lasts to the end of the file (module) it is declared in.

Consider the following:

```
main()
   { .... }

int what_scope;
float end_of_scope[10]

void what_global()
                 { .... }

char alone;

float fn()
                 { .... }
```

main cannot see what_scope or end_of_scope but the functions what_global and fn can. ONLY fn can see alone.

This is also the one of the reasons why we should ***prototype*** functions before the body of code ***etc.*** is given.

So here main will not know anything about the functions what_global and fn. what_global does not know about fn but fn knows about what_global since it is declared above.

```
NOTE: The other reason we prototype functions is that some checking can be
done the parameters passed to functions.

If we need to refer to an external variable before it is declared or if it
is defined in another module we must declare it as an extern variable. e.g.

   extern int what_global

So returning to the modular example. We have a global string AnotherString
declared in main.c and shared with WriteMyString.c where it is declared
extern.

BEWARE the extern prefix is a declaration NOT a definition. i.e NO STORAGE
is set aside in memory for an extern variable -- it is just an announcement
of the property of a variable.

The actual variable must only be defined once in the whole program -- you
can have as many extern declarations as needed.

Array sizes must obviously be given with
declarations but are not needed with extern declarations. e.g.:

   main.c:    int arr[100]:

   file.c:    extern int arr[];
```

Advantages of Using Several Files

The main advantages of spreading a program across several files are:

- Teams of programmers can work on programs. Each programmer works on a different file.
- An object oriented style can be used. Each file defines a particular type of object as a datatype and operations on that object as functions. The implementation of the object can be kept private from the rest of the program. This makes for well structured programs which are easy to maintain.
- Files can contain all functions from a related group. For Example all matrix operations. These can then be accessed like a function library.
- Well implemented objects or function definitions can be re-used in other programs, reducing development time.
- In very large programs each major function can occupy a file to itself. Any lower level functions used to implement them can be kept in the same file. Then programmers who call the major function need not be distracted by all the lower level work.
- When changes are made to a file, only that file need be re-compiled to rebuild the program. The UNIX make facility is very useful for rebuilding multi-file programs in this way.

How to Divide a Program between Several Files

Where a function is spread over several files, each file will contain one or more functions. One file will include main while the others will contain functions which are called by others. These other files can be treated as a library of functions.

Programmers usually start designing a program by dividing the problem into easily managed sections. Each of these sections might be implemented as one or more functions. All functions from each section will usually live in a single file.

Where objects are implemented as data structures, it is usual to to keep all functions which access that object in the same file. The advantages of this are:

- The object can easily be re-used in other programs.
- All related functions are stored together.
- Later changes to the object require only one file to be modified.

Where the file contains the definition of an object, or functions which return values, there is a further restriction on calling these functions from another file. Unless functions in another file are told about the object or function definitions, they will be unable to compile them correctly.

The best solution to this problem is to write a header file for each of the C files. This will have the same name as the C file, but ending in .h. The header file contains definitions of all the functions used in the C file.

Whenever a function in another file calls a function from our C file, it can define the function by making a `#include` of the appropriate `.h` file.

Organisation of Data in each File

Any file must have its data organised in a certain order. This will typically be:

- A preamble consisting of `#defined` constants, `#included` header files and `typedefs` of important datatypes.
- Declaration of global and external variables. Global variables may also be initialised here.
- One or more functions.

The order of items is important, since every object must be defined before it can be used. Functions which return values must be defined before they are called. This definition might be one of the following:

- Where the function is defined and called in the same file, a full declaration of the function can be placed ahead of any call to the function.
- If the function is called from a file where it is not defined, a prototype should appear before the call to the function.

A function defined as

```
float find_max(float a, float b, float c)
{  /* etc ... ... */
```

would have a prototype of

```
  float find_max(float a, float b, float c);
```

The prototype may occur among the global variables at the start of the source file. Alternatively it may be declared in a header file which is read in using a `#include`.

It is important to remember that all C objects should be declared before use.

The Make Utility

The ***make*** utility is an intelligent program manager that maintains integrity of a collection of program modules, a collection of programs or a complete system -- does not have be programs in practice can be any system of files (***e.g.*** chapters of text in book being typeset).

Its main use has been in assisting the development of software systems.

Make was originally developed on UNIX but it is now available on most systems.

NOTE: Make is a programmers utility not part of C language or any language for that matter.

Consider the problem of maintaining a large collection of source files:

```
 main.c f1.c ......... fn.c
```

We would normally compile our system via:

```
 cc -o main main.c f1.c ......... fn.c
```

However, if we know that some files have been compiled previously and their sources have not changed since then we could try and save overall compilation time by linking in the object code from those files say:

```
 cc -o main main.c f1.c ... fi.o .. fj.o ... fn.c
```

We can use the C compiler option (Appendix) `-c` to create a `.o` for a given module. For example:

```
 cc -c main.c
```

will create a `main.o` file. We do not need to supply any library links here as these are resolved at the linking stage of compilation.

We have a problem in compiling the whole program in this ***long hand*** way however:

- It is time consuming to compile a .c module -- if the module has been compiled before and not been altered there is no need to recompiled it. We can just link the object files in. <u>However</u>, it will not be easy to remember which files are in fact up to date. If we link in an old object file our final executable program will be wrong.

- It is error prone and laborious to type a long compile sequence on the command line. There may be many of our own files to link as well as many system library files. It may be very hard to remember the correct sequence. Also if we make a slight change to our system editing command line can be error prone.

If we use the **make** utility all this control is taken care by make. In general only modules that have older object files than source files will be recompiled.

Make Programming

Make programming is fairly straightforward. Basically, we write a sequence of commands which describes how our program (or system of programs) can be constructed from source files.

The construction sequence is described in
makefiles which contain ***dependency rules*** and ***construction rules***.

A dependency rule has two parts - a left and right side separated by a :

```
left side : right side
```

The `left side` gives the names of a ***target(s)*** (the names of the program or system files) to be built, whilst the `right side` gives names of files on which the target depends (eg. source files, header files, data files)

If the ***target*** is **out of date** with respect to the constituent parts, ***construction rules*** following the dependency rules are obeyed.

So for a typical C program, when a make file is run the following tasks are performed:

1.
The makefile is read. Makefile says which object and library files need to be linked and which header files and sources have to be compiled to create each object file.

2.
Time and date of each object file are checked against source and header files it depends on. If any source, header file later than object file then files have been altered since last compilation **THEREFORE** recompile object file(s).

3.
Once all object files have been checked the time and date of all object files are checked against executable files. If any later object files will be recompiled.

NOTE: Make files can obey any commands we type from command line. Therefore we can use makefiles to do more than just compile a system source module. For example, we could make backups of files, run programs if data files have been changed or clean up directories.

Creating a makefile

This is fairly simple: just create a text file using any text editor. The ***makefile*** just contains a list of file dependencies and commands needed to satisfy them.

Lets look at an example makefile:

```
prog: prog.o f1.o f2.o
  c89 prog.o f1.o f2.o -lm etc.

prog.o: header.h prog.c
                 c89 -c prog.c

f1.o: header.h f1.c
                 c89 -c f1.c



f2.o: ---
                 ----
```

Make would interpret the file as follows:

1.
prog depends on 3 files: `prog.o, f1.o and f2.o`. If any of the object files have been changed since last compilation the files must be relinked.

2.
prog.o depends on 2 files. If these have been changed prog.o must be recompiled. Similarly for `f1.o` and `f2.o`.

The last 3 commands in the makefile are called ***explicit rules*** -- since the files in commands are listed by name.

We can use ***implicit rules*** in our makefile which let us generalise our rules and save typing.

We can take

```
f1.o: f1.c
  cc -c f1.c

f2.o: f2.c
                 cc -c f2.c
```

and generalise to this:

```
.c.o:   cc -c $<
```

We read this as .source_extension.target_extension: `command`

`$< is shorthand for file name with .c extension.`

`We can put comments in a makefile by using the # symbol. All characters following # on line are ignored.`

`Make has many built in commands similar to or actual UNIX commands. Here are a few:`

```
   break     date        mkdir

> type           chdir           mv (move or rename)
                 cd              rm (remove)             ls
                 cp (copy)               path
```

There are many more see manual pages for make (online and printed reference)

Make macros

We can define ***macros*** in make -- they are typically used to store source file names, object file names, compiler options and library links.

They are simple to define, ***e.g.***:

```
SOURCES                  = main.c f1.c f2.c
CFLAGS           = -g -C
LIBS             = -lm
PROGRAM                  = main
OBJECTS                  = (SOURCES: .c = .o)
```

where `(SOURCES: .c = .o)` makes .c extensions of SOURCES .o extensions.

To reference or invoke a macro in make do $(macro_name).***e.g.:***

```
$(PROGRAM) : $(OBJECTS)
$(LINK.C) -o $@ $(OBJECTS) $(LIBS)
```

NOTE:

- $(PROGRAM) : $(OBJECTS) - makes a list of dependencies and targets.
- The use of an internal macros ***i.e.*** $@.

There are many internal macros (see manual pages) here a few common ones:

$*
    -- file name part of current dependent (minus .suffix).

$@
    -- full target name of current target.

$<
    -- .c file of target.

An example makefile for the WriteMyString modular program discussed in the above is as follows:

```
#
# Makefile
#
SOURCES.c= main.c WriteMyString.c
INCLUDES=
CFLAGS=
SLIBS=
PROGRAM= main

OBJECTS= $(SOURCES.c:.c=.o)

.KEEP_STATE:

debug := CFLAGS= -g

all debug: $(PROGRAM)

$(PROGRAM): $(INCLUDES) $(OBJECTS)
        $(LINK.c) -o $@ $(OBJECTS) $(SLIBS)

clean:
        rm -f $(PROGRAM) $(OBJECTS)
```

Running Make

Simply type `make` from command line.

UNIX automatically looks for a file called `Makefile` (note: capital M rest lower case letters).

So if we have a file called `Makefile` and we type make from command line. The `Makefile` in our current directory will get executed.

We can override this search for a file by typing `make -f make_filename`

e.g. `make -f my_make`

There are a few more `-options` for makefiles -- see manual pages.

Dave Marshall
1/5/1999