RMIT UNIVERSITY

Computer Science and Information Technology

Advanced Programming Techniques

(a.k.a. Programming in ANSI / ISO C)

Module 06 – Managing C Programs

"C code. C code run. Run code, run!"

-- anon

Managing C Programs

- All but the most trivial of C programs consist of **multiple source** files and **header** files.
- C provides language facilities to assist with multi-file programs, these include: **header files**, **storages classes** (such as *extern* and *static*), various **pre-processor directives** including those for conditional compilation.
- In addition, there are several software development **tools** to help facilitate the management and compilation of multi-file programs, such as "<u>make</u>" utility.
- This topic will focus on <u>storage classes</u>, the <u>pre-processor</u>, and features of the <u>make</u> utility.

Advantages of Multi-file programs

- A **team** of programmers can work on one program, **each working on a different file**.

- Each file can define a particular type of **object** as a **data type** 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.
  - Well-implemented objects or function definitions can be **re-used** in other programs, reducing development time.

Advantages of Multi-file programs

- Files can also contain all the **functions** from a **related group**. E.g., **all the file handling operations**. These can then be accessed like a function library.

- When changes are made to a file, **only that file needs be re-compiled** to rebuild the program.

Organising Multi-file C programs

- In typical multi-file C programs, each file will contain one or more functions. One file will include main(), while the others will contain functions which are called by others.

- 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 be located in a single file.

Organising Multi-file C programs

- Where objects are implemented as data structures, it is typical to keep **all functions which access that object in the same file**. This has advantages:
  - The object can be easily **re-used** in other programs.
  - All **related functions** are stored together.
  - Later changes to the object require **only one file** to be modified.

The role of header files in Multi-file C programs

- As a general rule of thumb, a typical C program will have **a header file for each of the C files**. Note that this not need to necessarily be the case.

- The header file will have the **same name** as the C file, but ending in **.h**. The header file contains **declarations of <u>symbols</u>**, **<u>types</u>**, and **<u>functions</u>** used in **the C file**.

The role of header files in Multi-file C programs

- 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**.

- Header files often define the “interface” for modules. More on this in the next topic.

Organisation of Files

- Source files are typically organised in a fairly orderly fashion:
  - A preamble consisting of **#defined** constants, **#included** header files and **typedefs** of important data types (though, some or most of these may be in a header file).
  - Declaration of **global** and **external variables**. Global variables may also be initialised here. In general you should avoid using global variables wherever possible.
  - One or more **functions**

Organisation of Files

- In general the order of items is important, as **every object must be defined before it can be used**. Functions which return values must be declared before they are called.

- Function prototypes may appear among the global variables at the start of the source file. Alternatively, and this is a **better** way to do it, they may be declared in a header file which is imported using an *#include*.

Storage Classes

- Scope rules
- Storage class auto
- Storage class extern
- Storage class static
- Storage class register
- Multiple source files
- Storage class static – revisited
- The make utility, revisited

Scope Rules

- Storage classes in C define the scope rules of variables and functions
  – i.e. its visibility (i.e. whether it is accessible or not) to other functions

- For variables, the storage class also define its:
  – creation – when it comes into existence
  – duration – when it ceases to exist

Categories of Linkage

C/C++ support 3 categories of linkage:

1. **External:** functions and global variables have external linkage; that is, they are available to all files that comprise a program.

2. **Internal:** available only within the file in which they are declared (e.g. global objects declared as **static**).

3. **None:** available only within its own block (e.g. local variables).

Storage Class: extern (1)

- Principal use is to specify that an object is declared with **external linkage** elsewhere in the program.

- Declaration vs Definition
  - Declaration: name and type of an object
  - Definition: causes storage to be allocated for the object
  - Multiple declarations of one object is allowed
  - ONLY ONE definition for one object is allowed

- In most cases variable *declarations* are also *definitions*.
- The **extern** specifier can *declare* a variable without *defining* it. That is, when there is no initial value!

Storage Class: extern (2)

- When you need to refer to a global variable that is defined in another part of your program, you can declare that variable using **extern**

```
int main(void)
{
  extern int first, last; /* use global vars */

  printf("%d %d", first, last);

  return 0;
}

/* global definition of first and last */
int first = 10, last = 20;
```


Borrowed from CPP, The Complete Reference, 4th Edition

Storage Class: extern (3)

- The default storage class for variables declared <u>outside</u> a block - i.e. **a truly *global* variable**

```
int x;

int main(void)
{ … }
void func1()
{ … }
```

<u>Created</u>:	when program starts
<u>Destroyed</u>:	when program stops
<u>Visibility</u>:	entire program

- <u>extern int x;</u> used if variable declared **in another file**
- Also the **default** storage class for **functions**

  i.e. function is visible (i.e. can be called) by all other functions

Storage Class: extern (4)

- Example for *extern* in multiple-file programs

File One

```
int x, y;
char ch;
int main(void)
{
 /* ... */
}

void func1(void)
{
 x = 123;
}
```

File Two

```
extern int x, y;
extern char ch;
void func22(void)
{
 x = y / 10;
}

void func23(void)
{
 y = 10;
}
```

Borrowed from CPP, The Complete Reference, 4th Edition

Storage Class: static (1)

- **static** variables are permanent variables within their own function or file.
- They are not known outside their function or file.
- They maintain their values between calls.
- **static** has different effects on local variables and global variables

Storage Class: static (2)

Scenario 1) When declaring local variables:

```
void func()
{
  static int count = 0;
  count++;
    …
}
```

Created:	when program starts
Destroyed:	when program stops
Visibility:	only within its block

- **Value** in variable count is **preserved**
- **Initialisation** occurs **only once**, not each time function is called
- Potentially dangerous but may be handy
- Rarely used/seen

Storage Class: static (3)

Scenario 2) When declaring global variables and functions:

Created:	when program starts
Destroyed:	when program stops
Visibility:	only to functions declared **after** the variable/function and **in the same source file**

- Can be thought of as *local-global*
  - local to its source file – i.e. invisible to other files
  - global to any functions (after it) within the source file
- Used wisely can simulate OO concepts in C

Storage Class: static (4)

File: source1.c

```
void func1()
{ … }
static void func2()
{ … }
static int count;
void func3()
{ … }
```

File: source2.c

```
static void func4()
{ … }
void func5()
{ … }
```

- func1() is visible (i.e. can be called by) **all** other functions
- func2() is visible to **only** func3()
- func3() is visible to **all** functions
- func4() is visible to **only** func5()
- func5() is visible to **all** functions
- count is visible (i.e. can be accessed) **only** by func3()

Storage Class: register

- A <u>request</u> to the compiler to use a (fast but scarce) register to store a variable (as opposed to storing it on RAM)
- Access to the object will be as fast as possible, depending on the implementation of the compiler
- It is a request, and might not be granted
- Operator <u>&</u> (address-of) cannot be used because the variable might not have a memory address
- Works best for *integers* and *characters*

```
int main(void)
{
    register int x;
    …
}
```

RMIT UNIVERSITY Computer Science and Information Technology

Storage Class: auto

- The default storage class for variables declared within a block – i.e. **a truly *local* variable**

```
int main()
{
    auto int x;
    …
}
```

Created:	when block is entered
Destroyed:	when block is exited
Visibility:	only within its block

- You virtually never see auto used explicitly

RMIT UNIVERSITY Computer Science and Information Technology

Multiple Source Files

- As we've seen, it is common in C for the source code to be split across more than one source file

driver.c

```
int main(void)
{
  get(…);
  calcTotal(...);
  calcTotal(...);
  put(…)
}
```

io.c

```
get(…)
{
    ...
}
put(…)
{
    ...
}
```

calc.c

```
calcTotal(...)
{
    ...
}
calcGrade(...)
{
    ...
}
```

Multiple Source Files (cont'd)

- To recap the benefits of multiple source files:
  - another level of **modularity**
  - **multiple programmers** working on one program
  - source code **management**
  - only need to (fully) recompile files that have **changed** when enhancements or bug fixes are made
  - easier for using version control systems
  - etc etc etc … improved *software engineering*

Multiple Source Files (cont'd)

- Each file can then be compiled separately

  gcc -c driver.c produces unlinked object file driver.o

  gcc -c io.c " " " " io.o

  gcc -c calc.c " " " " calc.o

- Which can then be linked into executable program:

  gcc -o myprog driver.o io.o calc.o

- Alternatively can be done all in one command

  gcc -o myprog driver.c io.c calc.c

Note: –ansi –pedantic –Wall parameters are not mentioned for simplicity.

The make utility

- make is a useful tool to keep track of all the source files used for a given task and to compile the source codes in a convenient manner.
- make will refer to a text file containing instructions on how the source codes are to be compiled.
- By default make will refer to a file named “Makefile” in the current directory.

The make utility

- If there is a text file named “Makefile” in the current directory, you can compile the source codes by,

  `make` – no argument to refer to the default “Makefile” file

NB – make is a language independent utility and can be used to compile other programming languages not only C.

The make utility (cont'd)

- Consider the `hello.c` program we discussed in Module 1. We can write a "Makefile" with the following to compile using make.

```
all:
        gcc hello.c -o hello
clean:
        rm hello
```

tab indentation – very important

- To compile

  `make` – by default this calls all the commands defined in label "all"

- To clean the executable

  `make clean` – call commands defined in a label called "clean"

The make utility

- Here is a simple Makefile for compiling the driver.c, io.c and calc.c files into the program myprog.

targets and dependencies

```
myprog: driver.o  io.o  calc.o
        gcc -o myprog driver.o io.o calc.o
driver.o:  driver.c driver.h
        gcc -c driver.c
io.o: io.c io.h
        gcc -c io.c
calc.o: calc.o calc.h
        gcc -c calc.c
```

actions to create targets

The make utility (cont'd)

- A comprehensive Makefile for myprog.

```
CC       = gcc
DEBUG    = -g
CFLAGS   = -Wall -ansi –pedantic $(DEBUG)
PROG     = myprog
OBJS     = driver.o io.o calc.o

all: $(OBJS)
	@echo “linking ...”
	$(CC) $(CFLAGS) –o $(PROG) $(OBJS)

%.o: %.c %.h                      # compile corresponding .c and .h files to .o files
	@echo “compiling ...”
	$(CC) $(CFLAGS) –c  $^

clean:
	@echo “cleaning ...”
	rm –f *.o $(PROG)
```

The C Pre-Processor

- Pre-processor directives
- #include
- Pseudo-constants: simple #define
- Macros: #define with parameters
- Undefining symbols: #undef
- Conditional compilation

Pre-processor directives

- The **C pre-processor** is the first of the 3 steps in compiling a C program
  - followed by the (true) **compiler** phase of syntax analysis and code generation
  - then the "**linking**" phase which resolves and links any external references and produces executable file
- Pre-processor directives start with a **#** symbol:
  #include
  #define and #undef
  #if (also #else, #elif, #endif, #ifdef, #ifndef)

- Use `gcc -E` to see the pre-processor output

Pseudo-constants – simple #define

- Simple textual substitution to define a pseudo-constant

```
#define MAX_ITEMS  100
int  arr[MAX_ITEMS];
```

the pre-processor would convert the array declaration into:

```
int  arr[100];
```

Macros – #define with parameters (1)

- #defines can also have parameters

  #define MY_PRINTF(x)   printf(“%d\n”, x)

- so that a statement such as:

  MY_PRINTF(amount * 100);

- would be converted by the pre-processor to:

  printf(“%d\n”, amount * 100);

Macros – #define with parameters (2)

- Some standard C library routines are implemented in this way. For example:
  - `int islower(char);`
  - `int getc(FILE *);`

- This inline substitution avoids the overhead of a function call
  - at the cost of larger executable file
    - if macro is invoked many times, and is large

Macros – #define with parameters (3)

Some **dangers** with parameterised macros – consider:

```
define square(x) x * x

   a = square(b);
=> a = b * b;
   /* which is fine but ... */

   a = square(b - c);
=> a = b - c * b - c;
   /* yikes!   */
```

Macros – #define with parameters (cont'd)

So let's try:

```
#define square(x)    (x) * (x)

 a = square(b - c);
=> a = (b - c) * (b - c);
 /*  fixes previous problem but … */


 a = square(b) / square(c);
=> a = (b) * (b) / (c) * (c);
 /*  yikes!  */
```

Macros – #define with parameters (cont'd)

So we need to include all necessary brackets:

```
    #define square(x)    ((x) * (x))

    a = square(b) / square(c);
 => a = ((b) * (b)) / ((c) * (c));
     /* OK */
```

Undefining symbols – #undef

#define symbols may be undefined:

```
#define max(x,y)   (x>y?x:y)
…
a = max(a, b);      /* uses the above macr   */
…                   /* i.e. a = (a > b ? a : b); */

#undef max

int max(int x, int y) /* 'max' redefined as func. */
{ … }
…
q = max(r, s);            /*  uses function max()     */
```

RMIT UNIVERSITY Computer Science and Information Technology

Conditional compilation (1)

- Pre-processor directives to control which segments of code do or do not get compiled:

#if	
#else	
#endif	
#elif	else-if
#ifdef	test whether a macro is defined
#ifndef	test whether a macro is not defined

Example: Inclusion Guards

Conditional compilation (2)

Some simple examples … version control ...

```
#define V1
...
#ifdef  V1
      printf("This is version 1");
      do_V1_stuff();
#else
      printf("This is standard version");
      do_std_stuff();
#endif
```

Conditional compilation (3)

To conditionally compile debugging code ...

```
#define DEBUG
...
#ifdef  DEBUG
     printf("Debug: p1 = %d\n", p1);
#endif
```

- once a very common debugging technique
- not required so often now with good symbolic debuggers
- but still handy ...

Conditional compilation (4)

Fancier debugging code ...

```
#define DEBUG_LEVEL  2
...
#if  DEBUG_LEVEL == 1
     printf("Debug: p1 = %d\n", p1);
#else
#if  DEBUG_LEVEL == 2
  printf("Debug: p1 = %d  p2 = %d\n", p1, p2);
#endif
#endif
```

Conditional compilation (5)

- It is (usually) possible to turn #defines on or off and/or set their values at compile time - e.g.

```
gcc -DDEBUG  -UV1  -DDEBUG_LEVEL=2  myprog.c
```

is equivalent to having the following lines in your source code:

```
#define DEBUG
#undef V1
#define DEBUG_LEVEL  2
```

- We will continue looking at more multi-file C programs when we talk about “Modular Abstraction”