Worksheet 2: Environments

Updated: 7th August, 2019

The objectives of this practical are:

- to understand the compilation process;
- to understand and practice using C preprocessor directives; and
- to understand and practice Makefile construction.

Pre-lab Exercises

(Attempt these questions before coming to the practical.)

1. Preprocessor Directives

Explain the effect of the following:

(a)
```
#include <marvellous.h>
```

(b)
```
#include "marvellous.h"
```

(c)
```
#define LENGTH 100
```

(d)
```
#define CUBE(x) ((x) * (x) * (x))
```

(e)
```
#define CALC(x,y,z) ((x) + CUBE(y) + CUBE(CUBE(z)))
```

(f)
```
#ifdef LENGTH
    printf("%d", LENGTH);
#endif
```

(g)
```
#ifndef THEFILE
#define THEFILE
void f(void)
{
    printf("Hello world\n");
}
#endif
```

2. Shell Commands

Construct bash commands to do the following:

(a) List all files in the current directory whose name contains "Ralph";

(b) List all files in the "codes" directory whose name consists of two digits, then a dash, then any three characters;

(c) List all files that end with a vowel in all subdirectories that *don't* end with a vowel;

(d) Create an alias for listing all .c files;

(e) Create an alias to turn the `-Wall -ansi -pedantic -Werror` switches on by default for gcc.

3. Global Variables and Static Functions

(a) Why are global variables considered bad programming practice?

(b) Why would you never declare a static function in a header file?

4. Compile Dependencies

Consider a program that consists of the following files:

```c
/* main.c */
#include "database.h"

int main(void) { ... }
```

```c
/* database.c */
#include "database.h"
...
```

```c
/* database.h */
#include "util.h"
...
```

```c
/* util.c */
#include "util.h"
...
```

```c
/* util.h */
...
```

```c
/* interface.c */
#include "interface.h"
#include "util.h"
...
```

```c
/* interface.h */
...
```

(a) Which file(s) does main.c include?

(b) Which file(s) include util.h?

(c) What .o files would be created during compilation?

(d) If database.h is modified, which .o file(s) would need to be recompiled?

(e) If util.h is modified, which .o file(s) would need to be recompiled?

(f) If util.c is modified, which .o file(s) would need to be recompiled?

Practical Exercises

1. Static local variables

Write a C function to calculate powers of 2. Your function should take *no* parameters. Each time it is called, the function should return the next power of two in sequence.

Called once, your function should return 2. Called a second time, your function should return 4, then 8, then 16, then 32, etc.

Note: Use *local* variables only, not global variables.

Once your function is finished, write a `main()` function to test it.

2. Macros

You have been asked to write the code for a progress bar for downloading a file. The progress bar will incorporate various statistics about download progress and display them.

You have access to the following information:

- the current time (measured in seconds elapsed since 1970);
- the download start time (also in seconds elapsed since 1970);
- the number of bytes downloaded; and
- the total file size in bytes.

Given this, write macro definitions for the following:

(a) `ELAPSED_TIME(time, startTime)`
— the download time so far (in seconds).

(b) `PERCENT_COMPLETE(bytes, totalBytes)`
— the percentage complete (0–100%).

(c) `DOWNLOAD_SPEED(time, startTime, bytes)`
— the current download speed (in bytes per second).

(d) `TOTAL_TIME(time, startTime, bytes, totalBytes)`
— the estimated total time (in seconds).

(e) `REMAINING_TIME(time, startTime, bytes, totalBytes)`
— the estimated remaining time (in seconds).

Save these macros in a header file called `download.h`.

3. Header Files and Multiple .c Files

Obtain the other 2 files; `download_stats.c` and `download_stats.h`. Have a look at how the c file is working, now compare the difference between the function version and the macro version.

Naming conventions: Macro names are usually in `ALL_CAPS`. Function names are usually either in `lower_case` or `camelCase`. Technically this is just a convention, not a C rule, but conventions are important for readability.

Compile your modified program as follows:

```
[user@pc]$ gcc -Wall -pedantic -ansi -Werror -c download.c
[user@pc]$ gcc -Wall -pedantic -ansi -Werror -c download_stats.c
[user@pc]$ gcc download.o download_stats.o -o download
```

4. Makefiles

Referring to the lecture notes, construct a Makefile for your code from the previous questions. You should make good use of Make variables and have a `clean` rule (being very careful with the `rm -f` command!)

To test it, manually remove any existing object/executable files and run:

```
[user@pc]$ make
[user@pc]$ ./download
```

Then, run "`make`" a second time; it should say that your program is "up to date". Test the other rules individually, including the `clean` rule.

5. Conditional Compilation

You have decided that some of the download statistics are a distraction. Using conditional compilation, make it possible to compile `download.c` *with or without* the first four statistics.

That is, the command "`gcc -c -Wall -ansi -pedantic -Werror download.c`" should compile a concise version that only outputs the estimated remaining time.

By contrast, the command "`gcc -c -Wall -ansi -pedantic -Werror download.c -DALL_STATS=1`" (which defines a constant called `ALL_STATS`) should compile another, verbose version that outputs all five download statistics.

Finally modify your makefile so that you are able to compile it with or without ALL_STATS. You should only need 1 makefile for this and only 1 executable is to be created at a time. Under resources is an example of how to achieve this.

6. Processes

Here we'll investigate how to manage running programs — "processes" — from the command-line.

(a) Run your `download` program from the previous exercise. *While* it runs, type Ctrl-C. What happens?

> **Note:** Ctrl-C forceably terminates (most) programs run from the command line. You can use it, for instance, if your program goes into an infinite loop.

(b) Run your program again, this time typing Ctrl-Z instead of Ctrl-C. What happens?

Now type `fg`. What happens?

> **Note:** Ctrl-Z "suspends" the current process, returning you to the prompt. However, the process is hasn't ended, and can be resumed in one of two ways: the `fg` and `bg` commands.

(c) Run your `download` program asynchronously (i.e. in the "background"):

```
[user@pc]$ ./download &
```

Try Ctrl-C this time. What happens? Wait for the output to stop.

Notice the extra prompts that appear, intermixed with the output. It may not look like it, but you could have entered *another* command in the same terminal while `download` was running. Experiment with this — try running `ls` (or anything else) at the same time as `download`. It will be difficult to see what you're typing, because the input and output will be intermixed, but the shell will understand.

> **Note:** The `bg` command mentioned above places an existing process into the background.

End of Worksheet