Worksheet 2: Environments

Updated: 20th February, 2020

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 Excercises

(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. 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?

3. 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

1. Static local variables

In a 'c' file called `powers.c` Write a 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, ot global variables.

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

> **Note:** Due to an integer only being 32 bits, if this function is called more than 31 times than an integer overflow will occur and the result will be invalid

2. Basic Macros

Create a new file called `macros.h`. In this newly made header file you need to do 2 things

(a) Set up header guards
(b) Create macros for TRUE and FALSE

3. Bounds Checking

In a new file called `bounds.c` write a function that will perform bounds checking on an integer. This function is to take in 3 integers:

low : The lower bound value
high : The upper bound value
value : The value to check if its within range

The purpose of this function is to be able to replicate the process of performing a check such as shown below. Note that the syntax below isnt possible in 'C'

```
low <= value <= high
```

The function is also to return an integer, the value being either the TRUE or FALSE macro from the previous question.

> **Note:** Do not re-write the macros from macros.h inside the 'c' file. Instead include the header file

After writing and testing the function, you are to write 2 more functions similar to this one. One for doing bounds checking on doubles and another for characters.

Now write a main to test your 3 functions.

4. Functions to Macro

After writing these 3 functions you should hopefully realise how similar they look so lets try simplify them. Go back to macros.h and create a new macro called BETWEEN. This macro is designed to replace all 3 of the between functions you just wrote.

> **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.

Modify your main inside `bounds.c` to use the macro instead of the 3 function. Once it works you can now remove the 3 between functions that you wrote as you no longer need them.

5. Multiple C files

Its now time to create a program that uses multiple 'C' files. So we are going to join our main inside `bounds.c` with the function we wrote in question 1 in `powers.c`.

> **Note:** As any program is only ever allowed 1 main function, delete your main function inside `powers.c`

Now back to your main inside `bounds.c`. Here you are to ask the user for an input and will proceed to run the power function that many times.

As you might've seen when testing question 1, the power function starts to return unexpected results after being called more than 31 times. So to ensure that this doesnt happen, check that

their input is within the bounds of 1 and 31 (Use your macro you just wrote). If it isnt then request a new input from them until it is.

When it comes to compiling multiple c files you have to do one at a time and then merge them together.

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

6. Conditional Compilation

There are times when coding that you are wanting to have the so called `debug prints`, however you find yourself having to constantly add and remove them. So why not create a macro that can do it for us.

Your goal is to have a printf statement inside your powers function that will only print when DEBUG is defined. The printf is to print out the value before it returns it.

Now to make this compilation work you have to compile powers.c with DEBUG defined shown as below.

```
[user@pc]$ gcc -Wall -pedantic -ansi -Werror -c bounds.c
[user@pc]$ gcc -Wall -pedantic -ansi -Werror -c powers.c -D DEBUG=1
[user@pc]$ gcc bounds.o powers.o -o bounds
```

Note: If you add the -D flag on the last command then it wont work as by that point the code has already been compiled

And if you were to recompile the code as you did from the previous question then the powers function should print nothing out.

7. Makefiles

As you have probably noticed by now, having to manually compile the code when you have more than 1 'c' file can get pretty tedious, and it only get worse the more 'c' files you have. This is where Makefiles come in.

Referring to the lecture notes, construct a Makefile to compile your code for you. You should be using good use of Make variables and have a `clean` rule.

Note: Get the makefile working without DEBUG first and then try to add it. An example can be found under blackboard->resources.

To test that your Makefile works run:

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

Then run "`make`" a second time; It should say that your program is "up to date". If it doesnt then something is missing in your makefile.

End of Worksheet