P2T: C Programming under Linux

Lab 5: Building in the Large

Introduction

The purpose of this laboratory is to help you become familiar with the compilation process and building larger projects that have several files. This lab is based on the Lecture 6 of the C component of P2T: C Programming under Linux course. This C Lab contains three Challenge Questions, which are the work which you will be marked on. All Challenge Questions, in all C Labs, are worth 10 points in their respective lab, unless explicitly noted. In general, you can assume that you will get at least 1, and up to 3, points for laying out well-presented, well-commented code; the rest of the marks will depend on the question (although partly-functional answers will get marks for the parts which work). (There is an additional, optional exercise in this Lab, but there are no marks for it, it is provided for interest.)

Preprocessing

The C preprocessor is a separate program called by the gcc before the compiler. Put simply, the C preprocessor is a fancy text substitution tool. Pre-processing includes:

- Replace defined TOKENS by predefined text.
- Remove blocks of text (based on test).
- Copy in (*include*) contents of another file.

All 'preprocessor directives' begin with the hash symbol (#). It must be the first non-blank character on the line. For readability, a preprocessor directives should not be indented if possible.

You can instruct gcc to stop at the preprocessor stage and print out the result to the terminal with the `-E` option (that is: `gcc -E mycode.c`).

The most commonly used preprocessor directives are covered in the following subsections.

#define directive

The syntax of the #define statement is the following:

```
#define identifier replacement
```

The define directive instructs the preprocessor to replace identifier with replacement in the rest of the file.

The identifier cannot contain spaces. It is a good practice to distinguish it from variable names by using uppercase letters, underscores, etc, so that you don't accidentally end up with the replacement text breaking things.

The replacement does not have such restrictions: it may include spaces, and consists of the rest of the line after the identifier. It can be an expression, a statement, a block or simply anything. However, the preprocessor will not replace identifier in string literals.

Keep in mind that the preprocessor directive extends only across a single line (you do not need to, and should not, add a semicolon (;) at the end of the directive). Although it is possible to extend a preprocessor directive through more than one line by using backslash ('\') character at the end of a line, in general this is not necessary. Below is an example demonstrating #define usage:

```c
#include <stdio.h>

#define MAX 100
#define MIN 5
#define STRING "Hello, World!\n"

int main(void) {
  char arr[MAX] = {STRING};
  printf(STRING);
  printf("%.MINs", STRING);
  return 0;
}
```

#define is used three times in this example. The identifiers are "MAX", "MIN" and "STRING". "MAX" and "MIN" are replaced with 100 and 5 respectively while "STRING" is replaced with the string "Hello, World!\n". You can use the gcc option -E to see how the code looks after preprocessing is done. In this case, it looks something like:

```c
int main(void) {
```

```
2   char arr[100] = {"Hello, World!\n"};
3   printf("Hello, World!\n");
4   printf("%.MINs", "Hello, World!\n");
5   return 0;
6 }
```

The preprocessor gave the expected results on the lines 2 and 3. “MAX” was replaced with 100 and the string “Hello, World!\n” replaced the identifier “STRING”. However, the line 4 is not what we might have expected. “MIN” did not get replaced with the integer 5. As it was mentioned before, the preprocessor does not replace identifiers within strings. The preprocessor does not warn about this: in this case, the resulting code would have generated a warning as `M` is not a valid conversion specifier, but this will not always be the case.

This example, as many other codes used before, begins with a statement:

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

The #include directive and header files

The `#include` is another preprocessor directive. When the preprocessor finds the `#include` directive it replaces it by the entire contents of the specified “header file”. There are two ways of using an `#include` statement:

```
#include <...>
```

```
#include "..."
```

The difference between these two declarations is where the preprocessor will look for a file/ header. `< >` quotes make the preprocessor look for a file in system defined locations containing generally useful header files, while `" "` quotes refer to a file in the same directory as the source file. As such, `< ... >` should be used for including header files for third-party code, while `" ... "` should be used for including header files managed as part of the same project.

Essentially, a “header file” is just another file containing C source code, by convention ending in ‘.h’. Usually, if we have a C file called “code.c” which we want other files to be able to use the contents of, we create a header file called “code.h”. “code.h” should contain function prototypes for all of the functions in “code.c” that we want other files to be able to use, and any other shared variables or definitions that the code relies on. Also, you should include comments sufficient enough for other programmers to understand what your functions do.

It's important to note that, as part of the process of `include`ing a file, that file is also read and interpreted by the preprocessor - so any preprocessor directives in that file will also affect the file including them (this is particularly relevant for `#define` directives).

Conditionals (#ifdef...#endif)

You can construct conditional statements using C preprocessor directives. These directives allow to include or discard part of the code of a program if a certain condition is met. For example, `ifdef` directive forms a preprocessor equivalent of an if-else construct with else and endif. To put this in code:

```
1 #define __VAR__ "Hello, World!\n"
2
3 #ifdef __VAR__
4   printf("Identifier __VAR__ has been defined!\n");
5   printf(__VAR__);
6 #else
7   printf("Identifier __VAR__ has not been defined!\n");
8 #endif
```

In the above example two `printf` statements on the lines 4 and 5 will be left in the code since the identifier `__VAR__` was defined before the conditional statement. The `printf` statement on the line 7 will be deleted before the compilation stage so that the program will not execute this statement. On the other hand, if `__VAR__` was not defined at the beginning, the two `printf` statements would be deleted and only the `printf` statement on the 7th line would be kept.

The opposite statement to the `#ifdef` is `#ifndef`. This is particularly useful in large projects which may contain nested header files. Suppose that two files (one.h and two.h) each include the same third file (three.h). In this case, if a file includes both one.h and two.h, it would end up including the contents of three.h twice, once from one.h and once from two.h. Depending on the contents of three.h, this is likely to prevent the resulting file from compiling (for example, by presenting multiple prototypes for the same function). The way around this problem is to build a check in three.h to see if it was already included. For example,

```
#ifndef _THREE_H_INCLUDED_
#define _THREE_H_INCLUDED_
more statements
#endif //_THREE_H_INCLUDED_
```

Having included these lines of code, the preprocessor will not include the statements within the "_THREE_H_INCLUDED_" `#ifndef`s twice. In the above situation, the preprocessor processes the file "one.h" first, which includes file three.h. "_THREE_H_INCLUDED_" has not been defined yet, so the preprocessor keeps all the statements between `#ifndef` and `#endif`, including the statement defining "_THREE_H_INCLUDED_". When two.h is processed, it also includes three.h, but "_THREE_H_INCLUDED_" has already been defined. Thus, statements between `#ifndef` and `#endif` are not included for a second time.

Compilation

The preprocessed source code is passed on to the Compiler. The Compiler does work of actually turning human readable source code into machine code. Firstly, the Compiler breaks the source code into syntactic units ("tokens") and then works out the meaning of the code, tagging every item (file scope, variables, functions and function calls) with a "symbol" (a name for referencing it). The resulting machine code, annotated with symbols, and prepended with a list of all symbols in the file, is called "object code".

Similarly to the Preprocessor and option '-E', you can stop compiling your process after the compilation stage using the gcc option -c. This will produce an object file with an extension '.o'.

This file, being mostly machine code, is not easily readable by a human (the command objdump, with various options, can tell you technical things about it, if you're interested). However, you can use the -S option to gcc instead, which produces a human readable assembly code representation of the same code. It may give you a sense of how source code is translated towards the machine code, but you are not expected to be able to read or write assembly as part of this course.

(Other useful gcc options include -Wall and -g. -Wall turns on all warnings and -g turns on debugging.) Below is a short table of useful gcc options (there are many more, as the manpage for gcc demonstrates!).

Optimisation

It is possible to set the level of optimisation in the compilation process. To put it other way round, we can ask the Compiler to be "smarter" about the compilation. Turning on an optimisation flag makes the Compiler to attempt to improve the

Option	Description
-c	compile or assemble the source file, but do not link (stop after compilation)
-S	stop after the stage of compilation proper, outputting human readable assembly
-E	stop after the preprocessing stage
-ansi	compiles using ansi C (C89)
-std=c99	compiles using C99 standard
-Wall	enables all the warnings

performance and/or code size at the expense of compilation time and the ability to debug a program.

The Compiler performs optimisation based on knowledge it has about a program. Compiling multiple files at once to a single output file mode allows the Compiler to use information gained from all files when compiling each of them.

We specify optimisation with `-On` option to the Compiler, where 'n' is an integer between 0 and 3. The extent and scope of the optimisation increases with n. Higher levels of optimisation can cause the compiled code to be larger, take more time to complete compilation, and, if you are doing especially 'clever'* things (or things sensitive to propagation of floating point errors), may change the results of the code slightly; they will almost always also improve the performance of the code compared to lower levels. Usually `-O2` is a reasonable choice which guarantees efficiency of a code and relatively fast compilation.

More discussion of Compiler Optimisation can be found in the file 'Compilers and Optimisation: the -O option' on Moodle.

*'clever' in this context usually means "making assumptions that things which aren't guaranteed by the C Standard will do what you expect".

Linking

Linking is the final step of the compilation process, and creates a single executable file from multiple object files. In other words, the Linker joins the object code from the Compiler, so that all the symbols are mapped to their representations.

Firstly, it takes all object code files it is given (one for each C source file) and turns them into one large file with "consolidated" symbol index at the top. Before the object code is linked, function calls are simply a note of the symbol that corresponds to the function they want to call. The Linker looks up the symbol in the

symbol index and replaces it with a call to the function code with the corresponding symbol. This also happens for variable names in the *file scope* in each file.
Errors about undefined, or conflicting definitions for, functions and variables are generated by the Linker (which is why they often appear a bit different to Compiler errors in their syntax).
At the stage of compilation the Compiler looks at one (source) file at a time. Therefore, if the Compiler cannot find a definition of a particular function it assumes that the definition is in another file. However, the Linker's business is to resolve all of the definitions needed for all of the symbols in all of the files it is combining - it will complain if it cannot find all of the definitions it needs.
(Strictly, the Linker here performs only "static linking", where all of the symbols are matched up within the code of the executable itself. "Dynamic linking"; where a symbol can be replaced with a reference to the object code that it will be found in, without that code being directly incorporated into the executable, is also fairly common, especially when using commonly available libraries which you can rely on being present on anyone's computer. The Linker also manages the configuration of dynamic links - although, of course, the link itself is resolved each time the executable runs.)

External libraries

The Linker allows us to link someone else's object code to ours as long as we know what symbols are in it. Useful code from other people is often distributed as *libraries*, essentially precompiled object code in a special form. If we use such external libraries we usually need to tell the linker about that. We do that by using the gcc flag -lname. 'l' stands for library and 'name' is the name of the library. Flags like this tell the linker to look for a (system) library called libname.a (or .so).

The most common example of an external library is the math library, which resides in a file called libm.so. If you use a function like `sqrt()` you need to include

```
#include <math.h>
```

at the beginning of your code. But also you need to add the -lm flag to gcc in your terminal, on a Linux system.

You should notice that we did not need to add any flags when we compiled programs that included header files like `stdio.h`, `stdlib.h` and `string.h`. These header files include function prototypes for code in the *C Standard Library*. The C Standard Library (libc.a or libc.so) is linked to any C project by default without the need for special directive. While the C Standard Library itself is large, the linker will only link those parts of it necessary for the code to compile (which is also the reason why there are so many different header files that all refer to it).

On a standard Linux install, the header files covering the C Standard Library are found in /usr/include (along with other headers which are considered important for the system), which is one of the ”system paths” that `< >` include directives search. The library file itself is usually found in /usr/lib/ or a subdirectory of this.

(The math library mentioned above is actually part of the C Standard Library, by definition, but for historical reasons is a separate library, as well as header, on Linux. If you're compling on OSX, the math library *is* in libc, so you wouldn't need the -lm.) Documentation on the version of the C Standard Library most commonly used by Linux, the Gnu C Standard Library, can be found here:
https://www.gnu.org/software/libc/manual/

By default, the linker will look in standard locations for the libraries you ask it to link. You can add an additional location to search (if a library is in a non-standard place), with the `-L` flag. (i.e. `-L/tmp/myplace` will make the linker also look in `/tmp/myplace` for libraries, in addition to the standard locations.)

Challenging Questions

Exercise 1: Building In The Large 1

In Lab 3, you wrote a simple program to print a PGM format image, and in Lab 4, you modified it to split out the code into multiple functions.

Starting with your code from Lab 4, split out the function to print out the PGM file into a separate C source file (called `printpgm.c`). Create an appropriate header file to allow your function to be used in other code. Modify the code in the original file to include the header file appropriately. Compile your two C files, and link appropriately, and test that the code still does what it is supposed to do.

Write an appropriate makefile so that anyone can compile the full code, and add appropriate compiler flags to enable debugging (`-g`) and the C99 standard (`-std=c99`). The makefile from Exercise 3 may help with this as a starting point.

You should have appropriate comments in all of the files, including the makefile, and present your code in a consistent and readable manner

Exercise 2: External Libraries

At one point in time, programming was taught to children at school using a language called “Logo”, which allowed you to direct a “turtle” around a screen, drawing shapes as it travelled.

We provide you with a C library, libturtle.a , and a header file turtle.h, to allow you too to experience the magic of Logo, via C... (Both of these are located in /usr/share/p2t/lab05/)

By reading the header file, develop an understanding of the use of the libturtle library. (Essentially, you have functions which turn drawing on and off, let you set the colour you will draw with, and let you direct the “turtle” in various directions and distances.)

Hence, write a single-file program with just a `main` function in it, which creates a 256 by 256 pixel image, and draws a regular polyon with N sides, saving it to a file called “mylogo.png”. N should be provided by a command line argument (**not an interactive prompt!**). The code should exit with a helpful line of text if called with no command line argument.

(Hint: the internal angles of a regular N-gon are $180 \times \frac{N-2}{N}$ degrees. The exterior angles are $\frac{360}{N}$ degrees. The Mathworld page for Regular Polygon may be of use, depending on how complicated you want to be: http://mathworld.wolfram.com/RegularPolygon.html - if you need more mathematical functions, you will have to use libm, of course.)

Additionally, write code so that if the program is executed with the name “square”, it does the same thing as if it is called with the command line option 4. (Hint: what does argv[0] contain?)

Compile and appropriately link the program. Create a simple makefile to automate this, building a copy of the program called “polygon” and a copy called “square”.

Exercise 3: Building In The Large 2

The code uses the `sqrt` (square root) and `cbrt` (cube root) functions which are declared in the standard library header called math.h, which is implemented in the library libm.a (or libm.so). You are also provided with a makefile in /usr/share/p2t/lab05/ .

Based on the expectations in the makefile, split the code up into 3 .c source files, and an appropriate number of header files, adding lines where necessary (also add appropriate comments). You will need to add one piece of additional information to the makefile as well. Hence compile the project using the provided makefile. Add a README file to the project explaining how to build it. You should use appropriate comments in all file, and present your code in a consistent and readable manner.

```c


//Calculate the real offset of the complex cube roots of a
    real, given the real root
double re_complexroots(double);

//Calculate the imaginary parts of the complex cube roots
    of a real, given the real root
double im_complexroots(double);

//try to parse input appropriately
double validate_input(int argc, char * argv[]);

int main(int argc, char * argv[]) {
  double x, cub_x, re_cub_x, im_cub_x;
  x = validate_input(argc, argv);
  cub_x = cbrt(x);
  re_cub_x = re_complexroots(cub_x);
  im_cub_x = im_complexroots(cub_x);
  printf("The cube roots of %f are %f, %f + %fi and %f - %
    fi.\n", x, cub_x, re_cub_x, im_cub_x, re_cub_x,
    im_cub_x );
  return 0;
}

double re_complexroots(double z) {
  return -1.0*z/2.0 ;
}

double im_complexroots(double z) {
  return sqrt(3.0)*(z/2.0);
}

double validate_input(int argc, char * argv[]) {
  if (2!=argc) {
```

```
  fputs("Too many inputs - just enter one double!", stderr
   );
  exit(1);
    }
  double val;
  int parsed = sscanf(argv[1], "%lf", &val );
  if (0==parsed) {
  fputs("Could not parse input as a double!", stderr);
  exit(1);
  }
  return val;
}
```

Exercise for interest

This exercise is not assessed as part of the course, and is optional, but may be of interest to see how all of the various stages of compilation occur.

- In /usr/share/p2t/lab05/optional there are some C source files. Read them to see what they do.
- Using the -E option to gcc, see what happens when the file main.c is preprocessed.
- Using the -S option to gcc, see what main.c and function.c look like when compiled to assembly. You don't have to understand assembly, but see if you can spot any of the values assigned in the C code.
- Using the -S option and the -On option, with n from 1 to 3, to turn on optimisation, check that the generated assembly changes when you compile the C source, for example, using the diff command. (Can you see how it has optimised main.c ?)
- Can you see what the difference between -O2 and -O3 is when compiling function.c?
- Using the -c option to gcc, make two object code files from the two C source files.
- Using the -o option to pick an executable name, link the two object files into one executable, and then execute it. (Does the result match what you expected?)