P2T: C Programming under Linux

Lab 5: Git and Make, multi-file C projects and libraries

Introduction

This lab introduces two tools which are useful when developing software: Git and Make. The information you need to complete the Linux exercises can be found in *Linux Lecture 6 – Makefiles and Git*.

Git is a revision control system which can be used to track and manage changes made to a set of files. This allows you to see how a file has changed over time and to tag particular points in its history, which is useful when looking for bugs or when preparing to make complex changes. This functionality is also very important when multiple software developers are working on a project together.

Make is a tool which lets you automate the software build process. Make lets you describe the sequence of commands needed to compile and link your code in the form of a Makefile. This can help you organise the compilation of larger projects containing many files.

Makefiles are especially useful when building projects that involve more than one source file, which is common for non-trivial projects. In addition, most projects will also make use of (potentially precompiled) resources from other sources, such as libraries. The way the C programming language handles multi-file projects is covered in *C Lecture 6*.

Getting Started

As with the previous labs, the files required for this lab can be obtained using **`git`**:

```
git clone https://bitbucket.org/glaphysp2t/linux-lab05.git
```

Revision Control and Git

Activity 1

This activity includes questions which should be answered in the file **`answers.txt`** which can be found in the **`linux-lab05`** directory created at the start of this lab. It also asks you to modify certain files, and your submission for this lab will include these files in addition to the answers file.

1. What command would you use to create a new Git repository in a particular directory? **(1)**
2. In a web browser, navigate to the following address: **(2)**

https://bitbucket.org/glaphysp2t/lab5-example

What is the full command you would use to create a copy of this repository in your home directory?

Create a copy of the **lab5-example** repository mentioned in question 2 in your home directory. Note that you will need to the use the URL beginning https:// for this; do not use the SSH URL.

The following questions ask you to make certain changes to the contents of your copy of the **lab5-example** repository. Change into this directory now

3a) What command would you use to display a list of commit messages, each on one line? **(1)**

b) What message is associated with the hash **2a65f62**? **(1)**

4. Build the code using the provided **Makefile**, then run it with the **make test** command.

After running the program, edit the **README.md** file and include a short description of the what the code does. One or two sentences will suffice. **(1)**

5. Having modified the **README.md** file in the previous step, what is the output of the **git status** command? **(1)**

6a) What command or commands would you use to commit your modified file to the local repository? **(2)**

b) Commit your modified file to the local repository, including a short, appropriate message. Copy the output of **git log** into the **answers.txt** file. **(1)**

7. What command would you use to create a new branch called **myfeature**?

Create this branch now. **(1)**

8a) What command would you use to change to the **myfeature** branch?

Change to that branch now. **(1)**

b) What is the output of the **git branch --list** command? **(1)**

9. Make any modification to the code. (It really doesn't matter what!) Commit your changes.

What is the output of **git log**? **(1)**

(14)

Make

Questions

10 . Briefly describe the meaning of the following automatic **make** variables:

a) **`$@`** **(1)**

b) **`$^`** **(1)**

c) **`$<`** **(1)**

d) **`$?`** **(1)**

(4)

Note: math.h, complex.h and libm

In the previous Lab, we discussed including **`math.h`** to allow the use of the special floating-point values **`INFINITY`** and **`NAN`**.

In fact, **`math.h`** is the C Standard Library header for *all* floating-point mathematical tools, including many useful mathematical functions, such as the trigonometric functions (**`sin`**, **`cos`**, **`tan`**, **`asin`** (arcsine), **`acos`** (arccos) and **`atan`** (arctan)), exponentials and logarithms (**`exp`** and **`log`**) and so on.

However, for historical reasons, the mathematical functions were split from the single file (**`libc`**) that contains all of the other C Standard Library implementation. Instead, they were placed in the (optional) library **`libm`**, which must be explicitly linked.

(We don't need to link **`libm`** if we just want **`INFINITY`** etc., as they're just constants, not functions.)

Nowadays, some C Standard Library implementations – for example, the one that comes with MacOS – include the math functions as part of the single **`libc`** library file. However, it is still safe to explicitly link **`libm`**, as all C compilers are aware of this history, and know what to do in their case.

Similarly, **`complex.h`** also declares equivalent functions for the complex floating-point domain, usually prefaced by a **`c`** in their name – so **`csin`** is the sin function defined for complex arguments. The implementations of these functions are also found in **`libm`**, so you need to link it if you use them as well.

Activity 2: Refactoring Your Own Code + Makefiles

In Lab 3, you wrote a simple program to print a PPM format image of a Newton-Raphson-Seki fractal, and in Lab 4, you modified it to split out the code into multiple functions.

11. Starting with your code from Lab 4, split out the function to print out the PPM image into a *new*, *separate*, C source file called **`printppm.c`** **(2)**

12. Create an appropriate header file (**`printppm.h`**) to allow your function to be used in other code, with a suitable prototype. **(2)**

13. Modify the code in the *original* file to include the header file appropriately. **(1)**

    Compile your *two* **`.c`** files, link appropriately, and test that the code still does what it is supposed to do.

14. Write an appropriate Makefile so that anyone can compile the full code:

    a) Write rules for

    - building a ".o" from a ".c" file
    - building the final executable from its component ".o" files
    - cleaning up the directory of ".o" files [this is a PHONY rule] **(3)**

    b) and add compiler flags to enable debugging (**`-g`**) and enable all warnings (**`-Wall`**) **(1)**

15. You should have appropriate comments in all the files, including the Makefile, and present your code in a consistent and readable manner. **(4)**

16. Your code should compile and link without warnings [with **`-Wall`** specified to clang] **(1)**

    You will not be penalised for any issues with Lab 3 and 4 code. If you need an example for Lab 4 to base your work on here, ask a demonstrator.

    Three C source code files (2 .c, 1 .h), and one Makefile, are your submission for this exercise.

(14)

Libraries and Linking

Activity 3: External Libraries + Makefiles

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 **`~/examples/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.)

17. Hence, write a single-file program with just a `main` function in it, which:

a) Creates a 256 × 256 pixel image **(1)**

b) Draws a regular polygon with N sides

(Hint: the exterior angles of such a polygon are 360/N degrees)

Your program should work for N between 3 and at least 10 (it may work for larger values of N) **(2)**

c) Saves the image to a file called "mylogo.png" **(1)**

where:

d) N is provided by a command-line argument and the code exits with a helpful line of text if no argument is given. **(2)**

Compile and link your program (you may need to link to the math library as well as **`libturtle`**, as **`libturtle`** relies on trig functions to draw lines at angles).

If your copies of **`libturtle`** and **`turtle.h`** are not in "standard locations", you may need to use **`-I`** and **`-L`** options to point the compiler and linker to the correct directories.

18. Create a simple makefile to automate the above process for building the project. You should include a "test" rule which builds the program (if needed) and then runs it with the argument '5'. **(3)**

19. You should have appropriate comments in all the files, including the makefile, and present your code in a consistent and readable manner. **(4)**

20. Your code should compile and link without warnings [with **`--Wall`** specified to clang] **(1)**

<u>One C source code file, and one makefile, are your submission for this exercise.</u>

(14)

<u>Note for students attempting this exercise on machines other than Brutha</u>

Because **`libturtle`** contains compiled code, the default version will not necessarily work with other systems. A number of alternative "libturtle" files are contained in the `extras` directory within the

~/examples/p2t/lab05/Exe3/ directory.
libturtle-intel-linux.a should work on newer Linux systems,
libturtle-intel-mac.a should work on older (Intel) Macs, and
libturtle-armv8-mac.a should work with newer ("Apple Silicon") Macs.

Note that this is semi-experimental, and that if you have issues using these alternative libraries, you should contact the Lab Head for support.

Submitting Your Work

You should create a **tar** or **zip** archive containing:

- Your **answers.txt** file.
- The **README.md** file from Activity 1.
- A subdirectory containing your submission files for Activity 2.
- A subdirectory containing your submission files for Activity 3.

You should submit this archive through Moodle.

Summary

Following this lab, you should know:

- How to create a new Git repository and to make a local copy of an existing repository.
- How to commit changes to a Git repository.
- How to read this history that Git maintains of changes to a repository.
- How to create a branch within a Git repository, and how to move from one branch to another.
- How to write a simple Makefile to build projects containing more than one source file.
- How to provide alternative targets in a Makefile to build a program with different compiler options.
- How to write C code distributed across multiple source files.
- The use of header files in C to allow the use of functions from one file in another.
- How to link external libraries to your code.

Optimisation (Optional)

For Interest 1: Optimisation and Compilation Flow in Detail

THIS SECTION IS OPTIONAL AND NOT FOR CREDIT

In **~/examples/p2t/lab05/optional** there are some C source files. Read them to see what they do.

Using the **-E** option to clang, see what happens when the file **main.c** is preprocessed.

Using the **-S** option to clang, 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 0 to 2, 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 **-O1** and **-O2** is when compiling **function.c**?

Using the **-c** option to clang, make two object code files from the two C source files.

The program **objdump** can process an object file and tell you various things about its contents.

Using:

```
objdump –d function.o
```

Verify that the contents of the object file's code looks like that in the assembly file (**function.s**)

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? If not, why not?