P2T: C Programming under Linux

Lab 6: GDB, File I/O and Pseudo-random Number Generators (PRNGs)

Introduction

GDB is the GNU Debugger, which can be used to test and debug programs written in C as well as in other languages. This lab introduces GDB and offers some basic familiarisation with its use. The information you need to complete the exercises can be found in *Linux Lecture 7 – Debugging with GDB*.

Interacting with files, both to read data from them and to write out data to them, is an important aspect of many programs. In addition, there are a number of techniques in mathematics and physics (for example, “Monte-Carlo” methods for evaluating integrals) which require random, or pseudo-random, sequences as input. This lab provides some experience in implementing file I/O in C, as well as a practical account of pseudo-random number generators (PRNGs). The information you need to complete the C component can be found in C Lectures 7 and 8.

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-lab06.git
```

GDB: the GNU Debugger

Activity

This activity includes questions which should be answered in the file **`answers.txt`** which can be found in the **`linux-lab06`** directory created at the start of this lab. In this directory, you will also find the source code for a simple C program, along with its **`Makefile`**.

1. Examine the **`Makefile`**. What is missing from this file that is required in order for us to debug our program with GDB? (Hint: something needs to be passed to the compiler in order to tell it to generate debug information.)

   Modify the **`Makefile`** to include the missing item and build a program to debug with the command **`make debug`**. **(1)**

2. Run the program from the command line.

a) What happens when the program is run? **(1)**

b) What command would you use to run the program through the debugger? **(1)**

3a) Run the program through the debugger. How would you set a breakpoint at the beginning of the code?

Set the breakpoint and start the program running using the **`run`** command. **(1)**

b) After execution halts, which line of code and which function is now highlighted or printed in the debugger? **(2)**

c) What command would you use to advance the execution by one line so that you enter into the function you identified in part (b)? Do this now. **(1)**

4. Continue to run the code using **`c`** or **`continue`**. The program will run and then break.

a) What is the error message, and on which line did the error occur? **(2)**

b) How would you print out the contents of the **`dataptr`** variable? **(1)**

c) What are the contents of the **`dataptr`** variable? **(1)**

d) How would you print out the contents of the memory this pointer points to? **(1)**

e) What happens when you try this? From reading the error message, what do you think the problem with this section of the code is? **(3)**

5. Exit **`gdb`** using the **`quit`** command. Change line 6 of **`util.c`** to read:

```
int *dataptr = data;
```

Re-compile the code and run it from the command line. What happens now? (Remember you can use **CTRL + C** to terminate a misbehaving program.) **(1)**

6. Run you code through the debugger again. On which line might you set a breakpoint in order to continue debugging the code? Why? **(2)**

7. Set a breakpoint on line 12, run the program and print out the contents of the **`data`** array. What are the contents of this array? **(1)**

8. How would you set a watch point on the index variable **`i`**? Do this now. **(1)**

9a) Continue running the code, using the **`next`**, **`step`** or **`continue`** commands. What happens? **(2)**

b) Look at the behaviour observed in part (a) and the contents of **`main.c`**, particularly line 13. What is the problem with the execution of the code? **(2)**

c) How would you correct the code on line 13? **(1)**

Make the change, recompile and rerun the program. It should now work correctly!

(25)

File I/O

Activity

From Lab 2 to Lab 5, you have built a project which outputs a PPM format image representing a Newton-Raphson-Seki fractal. In this Lab, we will extend this code to:

- Allow it to read in the colour map to use from an external file.
- Output the image to a file directly.
- Add additional functionality to save the binary version of PPM.

In this lab, you will modify mostly the code in the **`printpgm.c`** file you created in Lab5.

10. Add code to the top of your main function to read in colours from an external text-format file, and set the components of your colour map array appropriately.

    You will need to:

    - Open the file (for reading)
    - Loop over the file, storing values appropriately in an array of structs
    - Close the file.

    You should detect and deal with the case where the file cannot be opened by using default values, and print a warning to the terminal. You may assume that the file will always have at least 4 colours in it, and you do not need to read more than 4 of them. **(4)**

    You may lay out your external file format however you wish.

11. Now, modify the function in **`printpgm.c`** to output to a file (in text mode) rather than to the screen. By convention, PPM files have names that end in **`.ppm`**.

a) You will need to add lines to open the file in the appropriate mode at the start of the function, **(1)**

b) And close it at the end. **(1)**

c) You will also need to replace your **`printf`** functions with an appropriate function to print to the *file* instead. **(1)**

Compile, link and test the resulting code.

While the “P3” version of the PPM image format uses text representations of numbers, there is also a “binary” version of PPM where the image data is represented with actual integer values (not the text representation of them).

The “header” line for the binary PPM is the same as the text PPM, except

that the first two characters are “P6” rather than “P3”.
(It is still *text* and can still be written using a text formatted io function.)
There are no spaces between the values in this version of the format.

12. Write a new function in **`printpgm.c`** to output an array to the *binary* PPM format (you can start by copying the function you wrote in the previous section if you want).

a) Open the new file appropriately, and close it when done. **(2)**

b) Image data should be written using the **`fwrite`** function. (You can still write the *header* as text.)

You should write out each pixel’s “colour” in a single **`fwrite`**. **(3)**

c) Add a single additional line to the main C source file to make it output a binary PPM file (*as well as* the text PPM it already does), remembering to also alter the **`printpgm.h`** file for the new prototype **(1)**

Compile, link, and test the resulting code. (Make sure the two image files look the same when viewed using **display**.)

13. Compare the two files that are produced: using **`ls -lh`** or **`stat`**, check their sizes. (Optionally, look at the files using **`hexdump`** or **`decdump`**.)

a) Add comments on the different advantages and disadvantages of the two file formats. **(2)**

14. Appropriately comment and layout your code. **(6)**

15. Your code should compile in clang without warnings, even with **`-Wall`**. **(1)**

<u>Your submission for this Exercise is the modified C source files (2 .c and 1 .h).</u>

(22)

PRNGs

Activity

You are provided with the collection of tools and code in **`~/examples/p2t/lab06/Exe2/`** .

This is a testing framework for Pseudo-Random Number Generators, and some provided PRNGs to test.

The example **`main.c`** is set up to test the PRNG called RANDU (implemented in the object code **`randu.o`**, with the header **`randu.h`**). This has two functions:

"**srandu**" for seeding the PRNG

and

"**randu**" for getting a value.

There are three tests: a “spectral test”, a “roulette” (or “run length”) test, and a “periodicity test”, each of which is implemented in a particular function provided to you. Each function takes, as its first argument, the name of the “prng function” to test.

The Tests

Periodicity Test

The **periodicity** test determines how long it takes for the output from the PRNG to repeat (all PRNGs repeat eventually, with all seeds, the question is how long it takes). There is a max number of repetitions to test specified – don’t change this, as it is set to find periods for all the PRNGs which have short enough periods to find, while not taking too long to run. You can read the header file to find out what it does if the period is so long it can’t find it.

Spectral Test

The **spectral** test determines if there is correlation between successive values in the sequence produced by a PRNG. Correlations show up as visible structure in the output points (we’re doing a 3d spectral test, so we can visualise our output as a 3d scatter plot –the x,y,z coords for each red dot are successive values from the PRNG).

The provided script **`plotspectral.sh`** can be used to interactively plot the 3d points produced by the spectral test. It is run as:

```
./plotspectral.sh NAMEOFDATAFILE
```

You will need to exit the gnuplot shell after interactively rotating and zooming the graph window (with the mouse) by typing **`quit`**.

Roulette Test

The **roulette** test is based on a test used by casinos to measure the fairness of their roulette wheels. (Casinos keep the money on the table if the ball in roulette lands on the 0, so they are particularly interested in making sure their wheels roll a 0 the correct number of times.)

The test uses the provided PRNG to generate a sequence of values from a given range [0...N]. Every time a 0 is generated, it records the number of values until another 0 turns up.

The plot generated is the probability of going X values until the next 0, scaled to the theoretically perfect distribution. Deviations from perfect distribution show up as large, repeatable, deviations from the green line (that is, differences from the prediction which are large, and in the same place over multiple sequences).

The plotted results are *rescaled* to fill the graph drawn – remember to check the scale on graphs if you’re comparing two different PRNGs.

The provided script **`plotroulette.sh`** will plot the deviations from the mean. It is run as:

```
./plotroulette.sh NAMEOFDATAFILE
```

-

16. Appropriately compile and run the test framework (using the provided makefile) for RANDU as it is currently set up.

Plot the two .dat files it outputs, using the appropriate script for each test.

a) Run the test framework a few times – what are the output periods of RANDU for the seeds used? Add comments to the source code on this. **(2)**

17. There are two other PRNGs available for you to test with the same framework:

The **Mersenne Twister PRNG** (provided in the object code **`MT.o`** with the header file **`MT.h`** to let you use its functions)

The PRNG provided by the C Standard library (**`rand()`**)

a) By appropriately copying the contents of the main function already provided to test the RANDU function, **add** lines to test both of the above PRNGs with all of the tests, seeding all PRNGs with the same value at the start. **(4)**

You do not need to modify any C source code other than that in the main file.

You *may* need to add **`#include`** statements, as well as adding lines to the makefile to link additional files.

18. Compare the results of the three tests on the three PRNGs. Add comments to the main source file discussing which PRNG you think is the best, and why; and which is the worst, and why. **(4)**

Your submission for this is the modified C source files and makefile, including comments showing your conclusions.

(10)

Note for students attempting Exercise 2 on a system other than Brutha

As this exercise involves two precompiled object files (**`MT.o`** and **`randu.o`**), students attempting to work on their own systems may not be able to link to the default versions.

In the "extra" subdirectory of the Exercise folder, we provide some versions of the code compiled for other systems:

For newer Linux systems, **`MT-intel-linux.o`** and **`randu-intel-linux.o`** should work.

For Intel Mac systems, **`MT-intel-macos.o`** and **`randu-intel-macos.o`** should work.

For Apple Silicon Mac systems, **`MT-armv8-macos.o`** and **`randu-armv8-macos.o`** should work.

These are semi-experimental, so if you have issues with them please contact the Lab Head.

Submitting Your Work

Your submission for this lab is an archive file containing:

- Your **`answers.txt`** for Activity 1
- A subdirectory containing your submitted files for Activity 2
- A subdirectory containing your submitted files for Activity 3

The complete archive should be submitted through Moodle.

Summary

Following this lab, you should know:

- How to compile code suitable for use with GDB, and how to run a program through the debugger.
- How to advance through a program, either line by line, or until a particular line is reached or the value of a certain variable changes.
- How to display the contents of a variable.
- How to open and close files in C.
- How to write out data to a file in text format in C.
- How to write out data to a file in binary format in C.
- Some properties of PRNGs.

For Interest 1—Extended Topics on C Programming

THIS SECTION IS OPTIONAL AND NOT FOR CREDIT

In **`~/examples/p2t/extras/`** there are some additional directories containing worked examples of more advanced programming techniques with the C programming language and the C Standard Library.

Interested students may wish to work through these—there's not the same "set workflow" as the Lab exercises, but there's a lot of commenting in the files.

It is suggested that you start with

FunctionPointers

which does have a workflow for exploring the topic it touches on, and then move on to

Mandel

which is mostly documented in the source code [you should read the README first].