Lab 2

Objectives:

- Introducing the bare essentials of the GNU debugger program.
  - Though one commonly used debugging technique is to put print statements at various
    points in a program to detect logical errors, a more elegant method is to use a
    debugging program.
  - With such a program, you can trace values at various points in a program without
    altering and recompiling your source code. You can also pause and perform step by
    step execution of your code.
- The GNU debugger program is called **gdb**. Its GUI version is called **ddd**. This lab introduces
  **gdb**. Feel free to try **ddd**.

You do not need to hand-in anything in this lab session!

Resources:

More information on the GNU debugger program can be found on the web. Here are some tutorials:

- [https://www.tutorialspoint.com/gnu_debugger](https://www.tutorialspoint.com/gnu_debugger)

You can also refer to Stallman and Pesch:

- [http://mermaja.act.uji.es/docencia/is37/data/gdb.pdf](http://mermaja.act.uji.es/docencia/is37/data/gdb.pdf)

Part 1: Tracing a Segmentation Fault

- Login to the CSIL machines in the Linux environment, download and unpack **Lab2-files**
  into **sfuhome/CMPT295**, and change directories to **Lab2**. A directory listing should
  reveal two source files: **main.c** and **subprog.c**.
- Read the text of the two files before continuing, but do not modify the files yet. Yes, the
  program contains a run-time error: your plan is to diagnose it through **gdb**.
- Compile the main program using **gcc -g -c main.c**. Next, compile **subprog.c** with
  the same compile options. Then link the code into an executable called **runtest**. Note: It
  is important that the **-g** flag be included so that the program can be executed by **gdb**.
- As you will be compiling these source files a few times in this lab, you may find the
  following makefile very useful:
Run the executable `runtest` and observe the result. This error message should appear:

**Segmentation Fault**

This is one of the most common logical errors in both C and assembly language. It often occurs when an invalid address is used to retrieve either an instruction or some data during the program’s execution. Yes, this error message is very terse, i.e., not much information about the cause of the error is included. This is where the debugger comes into play.

Instead of running the executable file, this time launch the debugger using `gdb runtest`. A preamble followed by the prompt `(gdb)` should be displayed.

This indicates the debugger is now active and it is waiting for you to enter a command.

First try the command `run`. This will run your executable, and produce something like the following:

```
(gdb) run
Starting program: /home/userid/sfuhome/CMPT295/Lab2/runtest
The result is:

Program received signal SIGSEGV, Segmentation fault. 0x0000000000040064c in mystery (x=4660, str=0x0) at subprog.c:6
6     str[i] = (char) ((x & 0x8000) >> 15) | 0x30;
```

Notice the information that is provided:
- the function and the file name in which the error occurred is identified,
- the formal parameters of the function and their values are shown, and
- the line of source code on which the program was terminated is displayed.

Another useful command is `backtrace`. Try it. What does `gdb` show you?

Examining the content of the variables as your program executes can be very useful. The values of local variables within the function can be displayed by entering the `gdb` command:
(gdb) print i

- The debugger displays:

$$1 = 0$$

which is the value of the index i in the line of source code where the program was terminated. Because the value of i was 0, we can infer that the program was terminated during the first pass through the for loop.

- To see more of the function, you can use the list command. This will display lines near the line in question, in this case, line 6. Looking at the displayed lines, we can confirm that line 6 was indeed within the for loop.

  Note: The list command will display 10 lines or so. If these are not enough, you can issue the list command again.

- Segmentation faults are often the result of an undefined or faulty memory addresses. The only memory reference on line 6 is via the pointer str. According to the error message displayed by gdb after issuing the command run, the value of the parameter is str=0x0. Generally, an address of 0x0 is an invalid address. To confirm that it is the case, use:

  (gdb) print str[i]

- The debugger displays:

  Cannot access memory at address 0x0

  - Note that in C, a NULL pointer is defined to be an address of 0 and such pointer will generate a segmentation fault whenever your program attempts to dereference it.

  - The problem must be in the main routine, where an invalid pointer was passed to the function mystery. Having a look at main.c, we can see the declaration

    ```c
    char *someStr;
    ```

    This declares someStr to be a pointer to a string, but the declaration does no allocate space for the string. Remember that a string in C is a null ("\0") terminating array of char. To fix this, quit the debugger and edit main.c to change the declaration of someStr to be:

    ```c
    char someStr[16];
    ```

    - Recompile the program using your makefile.

    - When you re-run the code, the code completes as expected. But there is still a bug.

  Note: gdb commands do come in shorter version. See the links above for more information.
Part 2: Breakpoints

- The subroutine yields a curious side effect. To produce and examine this side effect, duplicate the last three lines of `main.c`, so that it reads:

```c
puts(msg);
mystery(x, someStr);
puts(someStr);

puts(msg);
mystery(x, someStr);
puts(someStr);
```

- Recompile and run the code. What do you observe? Anything missing in the displayed output?

- Launch `gdb` (`gdb runtest`), but do not type `run` yet. Instead, you will set up some breakpoints that will cause the program to pause. This way, you will be able to troubleshoot more easily, stepping through your program executing a few lines of code at a time and examining the content of the variables. Start by setting a breakpoint for the function call:

  ```
  (gdb) break mystery
  
  (gdb) run
  Starting program: /home/userid/sfuhome/CMPT295/Lab2/runtest
  The result is:

  Breakpoint 1, mystery (x=4660, str=0x7fffffffe630 "") at subprog.c:5
  5 for (i = 0; i < 16; i++) {
  ```

- Issue a `list` command to see where you are.

- Set up a second breakpoint at the beginning of line 7 with `break 7`. To execute the first iteration of the loop, use the command `continue`.

- You are now at breakpoint 2 (at line 7), and `gdb` is showing you the current values of `x` and `str`. You can also print the value of `i`. Another way to keep an eye on the value of variables as a program executes (aside from the `print` command) is to use the command `display x`, which displays the content of the variable `x` automatically after every step. So, issue the following commands in order to keep an eye on the value of variables `i` and `str`:
(gdb) display i
(gdb) display str

• If you do another continue, gdb will stop on line 7 of the next (2nd) iteration of the loop. By repeating this command, you can view the progress as the string str is being built, by each iteration of the loop. But make sure you stop issuing continue once the loop has completely executed (i = 15).

Note: To avoid having to type a command (e.g. continue) many times, you can recycle the commands you have already typed by pressing the up arrow key (like at the Linux command line). This brings back to the gdb command line the last gdb commands you have entered, one at a time, in reverse order.

• Once the loop has completely executed, use the next command to execute the last few lines of the mystery function.

• What is happening to the value of str? When the loop finished iterating, str contained:

"0001001000110100The result is:"

If you have a look at main.c, you can see that the character arrays somestr and msg are both declared one after the other. They are then given two adjacent memory locations, each of size 16 (each capable of holding 16 characters – index range: [0 .. 15]).

Now, have a look at subprog.c. What happens when the execution flows to line 9

str[16] = \0;

after the loop has executed? This null terminating character is actually written in the character array msg indicating that there are no characters in msg. Can you now see why nothing is displayed the second time msg is printed in main.c?

What would be the best way to fix this situation such that msg is actually printed twice, as expected on the computer monitor screen? Go ahead and fix the code.

---

Part 3: Challenge

In this final part, the challenge is for you to figure what the function mystery does.

Hint: Repeat Part 2 and display x along with i and str:

(gdb) display x.

Look at how the value of x changes at every iteration of the loop. Can you see what is happening to x? Remember that x has 16 bits (w = 16).
Can you also figure out what is being stored in $\mathbf{x}$, i.e., what each bit value (0 or 1) actually signifies?

Thank you to Dr. Dixon for having inspired this lab.