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 of variables 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`**.
- **Question**: Why are we wasting our time learning about this GNU debugger program?

  Answer: As stated above, using print statements to debug our C code is effective enough, however, when we debug a program written in assembly code, the *printing statement* technique may not be as easy to use as with C code. Therefore, the main objective of this lab is to give us a tool (this GNU debugger program) which can help us when we are writing assembly code.

  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
- http://www.unknownroad.com/rtfm/gdbtut/

You can also refer to Stallman and Pesch:

- http://web.mit.edu/gnu/doc/html/gdb_toc.html
- 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 **`Lab2.zip`** from our course web site, save and unpack it (using the command `unzip`) in your **`sfuhome/CMPT295`**, and change directory 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. Note: It is important that the **-g** flag be included so that the program can be executed by **gdb**.
- Then link the code into an executable called **runtest**.
- As you will be compiling these source files a few times in this lab, you may find the following **makefile** very useful (tip: use a tab to indent the lines under each target):

```
runtest: main.o subprog.o
    gcc -o runtest subprog.o main.o

main.o: main.c
    gcc -c -g main.c

subprog.o: subprog.c
    gcc -c -g subprog.c

clean:
    rm -f runtest *.o
```

- 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 by typing **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.
0x000000000040064c in mystery (x=4660, str=0x5555555 …) at
subprog.c:6
6       str[i] = (char) ((x & 0x8000) >> 15) | 0x30;
```

- Notice the information provided:
  - the function and the file name in which the error occurred and the line at which the error occurred are 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?
  Answer: **`gdb`** shows you the stack of all called functions, still currently executing, up to the point of failure. This tells you which function failed and how you got there (chain of calls).
- Now, examining the content of the variables as your program executes can be very useful too. The values of local variables within the function can be displayed by entering the **`gdb`** command:

  ```
  (gdb) print i
  ```

- Try it. The debugger displays:

  ```
  $1 = 0
  ```

  which is the value of the index **`i`** in the line of source code where the program failed and terminated. Because the value of **`i`** was 0, we can infer that the program failed and 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 address. The only memory reference on line 6 is via the pointer **`str`**. The problem may be in the **`main`** function, where an invalid pointer was passed to the function **`mystery`**.

  As we saw when we issued the command **`backtrace`**, the function **`mystery`** was called at line 13. So, let's have a look at **`main.c`** and see what happens prior to line 13.

  In the declaration section, we see:

  ```
  char *someStr;
  ```

  This declares **`someStr`** to be a pointer to a string, but the declaration does no allocate space i.e., memory for the string.
- To fix this, **`quit`** the debugger and edit **`main.c`** to change the declaration of **`someStr`** to be:

```
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. For example, the shorter version of **print** is **p**. See the links above for more information.

Suggestion: I encourage you to create a cheat sheet of **gdb** commands for yourself. You may find such list useful.

BTW, there is a small cheat sheet in our textbook, in Figure 3.39 on page 280, geared toward assembly language programs.

Part 2: Breakpoints

- The code yields a curious side effect. To produce and examine this side effect, edit **main.c** by duplicating its last three lines so that the **main** function now reads:

```
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
```

- Next, type **run**. The program will pause at the first statement of the function **mystery**, after it is called:

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

```
Breakpoint 1, mystery (x=4660, str=0x7fffffffe630 "") at
subprog.c:3
3       char *mystery(bit16 x, char* str) {
```

- Issue a **list** command to confirm that we are at the start of the function **mystery**.
- Set up a second breakpoint at the beginning of line 7 with **break 7**. This breakpoint will allow you to stop and examine the content of variables at each iteration of the loop. 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**. By the way, where does "**4660**" (the value of x) come from? You can also print the value of **i** by issuing the command **print i** (or **p 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, i.e., **i = 1**. By repeating this command, you can view the string **str** being built at each iteration of the loop. But make sure you stop issuing **continue** once you reach the last iteration, i.e., when **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 (as we do 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.

- What has happened to the value of **str**? When the loop is iterating for the last time, i.e., when **i = 15**, **str** contained:

```
"0001001000110100The result is:"
```

  Have a look at **main.c**. You can see that the character arrays **someStr** and **msg** are declared one after the other, which means that they are given two adjacent memory locations, each location capable of holding 16 characters (**hint**: each of these 16 characters are accessible using indices 0 to 15).

- Now, have a look at **subprog.c**. What happens when the loop terminates (**i = 16**) and the execution flows to line 9. To observe this, put a third breakpoint at line 9 then enter

continue. The execution flow will stop at line 9 and you will see the following on the screen:

```
(gdb) continue
Continuing.

Breakpoint 3, mystery (x=0, str=0x7fffffffd5c0
"0001001000110100The result is:") at subprog.c:9
9       str[16] = '\0';
1: i = 16
2: str = 0x7fffffffd5c0 "0001001000110100The result is:"
```

- **Step** to execute line 9 (**str[16] = '\0';**) of the **mystery** function. Remember, in C, we *null-terminate* our strings (array of characters).

  What has happened to **str**?

  This null terminating character at index 16 is actually written in the first character (index 0) of the character array **msg** indicating that there are no characters in **msg** when it is printed the second time in **main.c** i.e., when the second **puts(mgs)** in **main.c** is executed. This is why nothing is displayed on the computer monitor screen. Ah! Mystery solved!

- What would be the best way to fix this situation? How can we print **msg** twice, as expected, on the computer monitor screen? Go ahead and fix the code.

Part 3: Exploration

- Open the demo program **Lecture_4_Demo.c** found under Lecture 4 in your favourite code editor and as instructed in Lab 1, change the data type of the iterating variable **i** from an **int** to an **unsigned int** in the function **show_bytes(...)**.
- Compile (with **-g** flag) and execute the program using **gdb**. From our Lab 1, we already know what is going to happen. The idea now is to use **gdb** to explore the situation and answer the question **why it happens?**

Part 4: Challenge

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

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