Family Name:______________________________

Given Names:______________________________

ID#:______________________________

CMPT 295 Lab 2 (0.5%)

Thank you to Dr. Dixon for portions this lab.

The purpose of this lab is to introduce you to the use of `gdb`, the GNU debugger program. Though one commonly used debugging technique is to put print statements at various points in your 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 having to alter and recompile your source code. You can also pause and perform step by step execution of your code.

Note: This tutorial introduces you to the bare essentials of `gdb`. More information can be found on the web, or by referring to Stallman and Pesch.

Part 1: Tracing a Segmentation Fault

- Login to the CSIL machines in the Linux environment, download and unpack the care package into `sfuhome`, and change directories into `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 mainline program using `gcc -O0 -g -c main.c`. Next, compile the sub-program with the same compile options. Then link the code into an executable called `runtest`.

  Note: It is important that the `-O0` and `-g` flags be included: the first ensures that no optimization occurs; the second that the program text be included for use in `gdb`.
- Run the executable 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 will come into play.

- Instead of running the executable file, this time launch the debugger using `gdb runtest`. A preamble should be displayed and then the prompt:

  ```
  (gdb)
  ```

  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/lab2/runtest
  The binary is:

  Program received signal SIGSEGV, Segmentation fault.
  0x000000000040064c in cvt2bitstr (x=4660, str=0x0) at subprog.c:5
  5                       str[i] = (char) ((x & 0x8000) >> 15) | 0x30;
  ```

  Notice the information that is provided:

  - the subprogram and file name where the error occurred is identified;
  - the parameters of the subprogram and their values are shown; and
  - the line of source code on which the program was terminated is displayed.

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

- The values of local variables within the subprogram can be displayed. For example, enter the following `gdb` command:

  ```
  (gdb) print i
  $1 = 0
  ```

  This will display the value of the index `i` at the line of source code where the program was terminated. Because the value was 0, we can infer that this was the first pass through the for loop.

- To see more of the subprogram, you can use the `list` command. This will display lines near the line in question, in this case, line 5. Indeed, the command was within the for loop.

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

- Segmentation faults are often the result of an undefined or faulty memory address. The only memory reference on line 5 is via the pointer `str`. According to `gdb`, it shows the value of the parameter is `str=0x0`. Generally, an address of 0x0 is an invalid address. To confirm that is the case, use:

  ```
  (gdb) print str[i]
  Cannot access memory at address 0x0
  ```

  Note: In C, the `NULL` pointer is defined to be an address of 0 and it will generate a segmentation fault whenever your program attempts to dereference one.

- The problem must be in the mainline routine, i.e., it has passed an invalid pointer to the function. Indeed the declaration

  ```
          char *bitstr;
  ```

  was the right type (a pointer), but no actual space for the string was declared. To fix this, `quit` the debugger and edit `main.c` to change the declaration of `bitstr` to be:

  ```
          char bitstr[16];
  ```

- Recompile and relink the code. Remember to use the compile time flags `-O0` and `-g`.

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

Part 2: Breakpoints

- The subroutine yields a curious side effect. Within `main.c`, duplicate the last 3 lines so that it reads:

  ```
          puts(msg);
          cvt2bitstr(x, bitstr);
          puts(bitstr);
          puts(msg);
          cvt2bitstr(x, bitstr);
          puts(bitstr);
  ```

  Recompile and run the code. What do you observe?

- Launch `gdb`, but do not type `run`. Instead, you will set up some *breakpoints* that will cause the program to pause. Start by setting a breakpoint for the function call:

  ```
  (gdb) break cvt2bitstr
  ```

  Note: You can use the `Tab` key to complete commands quickly. Try it.

- Next, type `run`. The program will pause when `cvt2bitstr` is called:

  ```
  (gdb) run
  Breakpoint 1, cvt2bitstr (x=4660, str=0x7fffffffe630 "‘\346\377\377\377\177")
      at subprog.c:4
  4               for (i = 0; i < 16; i++) {
  ```

  Do a `list` to see where you are.

- Set up a second breakpoint at the beginning of line 6 with `break 6`. To continue to the next breakpoint use the command `continue`.

- You are now at breakpoint 2, and it shows you the current values of `x` and `str`. You can also print the value of `i`. If you do another `continue`, it will stop on line 6 of the next loop. Thus, you can view the progress as the binary string is built, loop by loop.

- You could step through the loop, one loop at a time, by typing `continue` 15 times. One shortcut is to press the `Enter` key instead, which repeats the last `gdb` command. Another way is `continue n`, where `n` is the number of breakpoints until the next pause.

- One particular breakpoint is very telling, i.e., the value of `str` shows exactly why the bug occurs. Find that critical breakpoint and figure out how best to fix the bug.

- This is the last step of Lab 2. Call the TA to your terminal to view your `gdb` window at the critical breakpoint. Then demonstrate your fix to `main.c` that would correct the bug. After verifying your work, the TA will then collect this sheet.

Thinking Ahead

The term *side effect* is usually heard in relation to pharmaceuticals: a drug is meant to solve a problem within a targeted area of your system, but it may interact with other, usually untargeted, parts of your system. The computer term *side effect* follows a similar vein: a function call may have an observable interaction outside its local variable scope.

Though the connotation of the medicinal type of side effect is usually bad, not all computer science side effects are. The function call from today's lab was designed to modify a character string outside of the local variable space of the function itself, a very common design choice in C. Of course, because of the misuse of the function call, an undesirable side effect occurred: a special type of error called a *buffer overrun error* obliterated the mainline's string `msg[]`.

Side effects can play their role in assembly language too. To do a branch, you usually overtly set the `FLAGS` register by using a `cmp` instruction, then do your jump based on those `FLAGS`. But the `dec` instruction also modifies some of the `FLAGS`: `ZF`, `SF` and `OF`. Sometimes these side effects can save you a compare instruction and result in tighter code.