CS240 Lab Assignment 6

GDB and C Pointers

You will be using the GNU debugger **gdb** when running C programs using pointers in this week's and upcoming labs and assignments. The purpose of a debugger is to allow you to see what is going on "inside" a program while it executes—or what it was doing at the moment it crashed. You can also display values of variables and examine contents of memory using **gdb**, which will be handy in understanding the effect of your programs on the hardware of the system.

The following is a link to a **gdb** manual, which you may use for reference:

http://www.gnu.org/software/gdb/documentation/

NOTE: Do not run your C programs directly on your own machine unless you are using *wx* (virtual machine). Either log in to one of the Linux machines in 173 or the micro-Focus, or log in remotely to the server (or one of the other Linux machine, use the appropriate name):

ssh –Y your_username@cs.wellesley.edu

1. For this exercise, answer any questions shaded in grey and hand in a hardcopy at the beginning of lab.

Create the following program using emacs and save as `prime.c`:

```
/* CS 240 program to check if a number is prime */

#include <stdio.h>

int test_prime(int num) {

  int i;
  int prime=1; //assume it is prime initially

  for (i=2; i<=num/2;++i) {

    if (num%i == 0) {
      prime = 0; // set to not prime
      break;
    }
  }
  if (prime)
    printf("%d is prime\n",num);
  else
    printf("%d is not  prime\n",num);
}

int main() {
  int test1 = 5;
  int test2 = 12;

  test_prime(test1);
  test_prime(test2);
}
```

Before running the program, examine the code and explain how it works:

3. In order to run programs under gdb, they should be compiled with debugging symbols turned on (**-g** option):

```
$ gcc -g -o gdb-example gdb-example.c
```

4. Run the program, and you should see the following output:

```
$ ./gdb-example
  5 is prime
 12 is not  prime
```

5. Now, run the program under gdb:

```
$ gdb gdb-example

      GNU gdb (GDB) Red Hat Enterprise Linux (7.2-75.el6)
      Copyright (C) 2010 Free Software Foundation, Inc.
      License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
      This is free software: you are free to change and redistribute it.
      There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
      and "show warranty" for details.
      This GDB was configured as "x86_64-redhat-linux-gnu".
      For bug reporting instructions, please see:
      <http://www.gnu.org/software/gdb/bugs/>...
      Reading symbols from /home/jherbst/gdb-example...done.
```

You will see the **(gdb)** prompt, and you can now enter gdb commands to perform various actions.   Observe and verify the output as indicated:

```
(gdb) run

  Starting program: /home/jherbst/gdb-example
  5 is prime
  12 is not  prime

  Program exited with code 021.

(gdb) quit
```

So, **run** is used to execute the program, and **quit** is used exit gdb.

6. The gdb debugger also allows you to walk through the program while it is running so that you can trace its steps carefully. Start another gdb session:

```
$ gdb gdb-example
```

The **break** command sets a breakpoint--a location in the program where gdb should stop when it gets to there. Breakpoints can be set at the beginning of a function or at specific lines in program file. There are many things that can be done with breakpoints, such as making them conditional or temporary. In this example, a common and simple usage case was shown that had gdb stop at the beginning of the main function.

```
(gdb) break main

  Breakpoint 1 at 0x40054b: file gdb-example.c, line 27.
```

Then, when you run the program, it pauses execution at the breakpoint:

```
(gdb) run

  Starting program: /home/jherbst/gdb-example

  Breakpoint 1, main () at gdb-example.c:24
  24    int test1 = 5;
```

The highlighted line above is the next statement to be executed when the program is resumed (the first statement in the *main()* function.

The **print** command displays the value of variables or expressions within the scope of the current frame. So, since *test1* is declared in main, you can print its value at this point:

```
(gdb) print test1

  $1 = 0
```

The $1 represents the variable. The current value is 0 because the statement initializing the value to 5 has not yet been executed. Execute a single statement by doing a **step:**

```
(gdb) step

  25    int test2 = 12;
```

Now display *test1* again:

```
(gdb) print test1

  $1 = 5
```

Try displaying a variable outside the current frame (*num* is a local variable inside *test_prime()*, so it is not understood at this point:

```
(gdb) print num

No symbol "num" in current context.
```

Execute another statement:

```
(gdb) step

  27    test_prime(test1);
```

The **step** and **next** commands are both used to make gdb move forward in the program. For statements that do not involve functions, the next and step commands are identical and merely make gdb execute one statement. For statements that involve a function, however, the two commands are different. **next** tells gdb to execute the entire function, while **step** tells gdb to move inside the function.

So, the entering **next** at this point should execute the entire function *test_prime(test1):*

```
(gdb) next

  5 is prime
  28    test_prime(test2);
```

Next, start to **step** through the second invocation of *test_prime()*:

```
(gdb) step

  test_prime (num=12) at gdb-example.c:8
  8     int prime=1; //assume it is prime initially
```

Now that you are within the *test_prime()* function, you can also change the current context with the **up** or **down** commands (this doesn't change the point at which you are executing the program, but instead allows you to display values defined within a different context or frame):

```
(gdb) up

  #1  0x0000000000400571 in main () at gdb-example.c:28
  28    test_prime(test2);
```

Use the **info** command to display information about the current frame:

```
(gdb) info locals

  test1 = 5
  test2 = 12
```

Go back **down** to the *test_prime()* frame, and display information about the **args** (arguments) in the current frame:

```
(gdb) down

  #0  test_prime (num=12) at gdb-example.c:8
  8     int prime=1; //assume it is prime initially

(gdb) info args

  num = 12
```

You can also use the **frame** command (with a numeric argument) to choose which stack frame to switch to.

The **backtrace** command produces a list of the function calls, which is known as either a *backtrace* or a *stack trace*.

```
(gdb) backtrace

   #0  test_prime (num=12) at gdb-example.c:8
   #1  0x0000000000400571 in main () at gdb-example.c:28
```

Reading backtraces is fairly straightforward. The data associated with each function call in the list is known as a *frame*.

The outermost frame is the initial function that your program started in, and is printed at the bottom of the list. Each frame is given a number (0, 1, 2, etc.). Following the frame number is an associated memory address (where the instruction is actually stored in memory).

Then each frame contains the name of the function that was called, its arguments, the name of the file where the function appears, and line number.

How many frames are there? Why that many?

Mark and label the following for the highlighted frames shown above:

- Outermost frame
- For each frame:
  - frame number
  - function name,
  - function arguments (if any), and
  - line number.

From what you have learned in class so far, what do you think the **0x0000000000400571** refers to (if this value is not exactly the same as what you see, don't worry):

Another convenience provided by gdb is to **list** a small segment of the code around where the program is currently stopped so you can see which statements have been executed and which ones are about to be:

```
(gdb) list
```

```
#include <stdio.h>

int test_prime(int num) {

  int i;
  int prime=1; //assume it is prime initially

  for (i=2; i<=num/2;++i) {

    if (num%i == 0) {
```

Step through several more instruction, following execution of the program.

To finish the program, enter **cont** to continue execution to the end:

```
(gdb) cont

 Continuing.
  12 is not  prime

  Program exited with code 021.


(gdb) quit
```

2. Investigate the use of pointers in a C program, to give you some more practice with editing and compiling C programs, and understanding how pointers work.

- On Bitbucket, fork the pointers repository https://bitbucket.org/wellesleycs240/cs240-pointers/fork (just once, then you will also have all the files for this week's lab and assignment) and add *bpw* as admin.

  The following commands are entered at the command-line prompt on a Linux machine:

  - From your account on a Linux machine, clone the cs240-pointers:

    ```
    hg clone ssh://hg@bitbucket.org/yourbitbucketname/cs240-pointers
    ```

  - Compile tour.c:

    ```
    make tour
    ```

  - Open *tour.c* using emacs, perform the experiments it describes, and record your answers in that file as comments. Remember that to run the program after compiling:

    ```
    ./tour
    ```

Submit a hardcopy of *tour.c* with your added comments.