Lab 6: Debugging

CSE/IT 113

1 Introduction

This lab introduces debugging with GNU's debugger `gdb`. The lab is split into three parts. Part I consists of a tutorial introduction to `gdb`. You will write a man page for gdb. Once you complete Part I, then you have the skills to complete Part II, which is applying debugging techniques to a variety of problems. Part III asks you to solve a few C puzzles.

All files are in the tarball `lab6.tar.gz`.

Part I

2 Debugging with `printf`

One way to debug is to use `printf` statements throughout your code to print variables. This works well for watching one or two variables, but it is tedious at best if you want to watch every variable in your program.

A much more efficient way is to use a debugger. A debugger is a program that is used to test and debug your program. Debugging is the process of finding bugs in your program. A debugger provides a whole lot more information than `printf` statements can provide.

3 Principle of Confirmation

Debugging is the process of confirming, one by one, that the many things you *believe* to be true about the code *are* actually true.

Debugging helps you find where in your code your assumptions are *false*. This is a clue to where the bug is located.

4 Debugging with `gdb`

For this lab we are going to use `gdb`, the debugger that comes with `gcc`

The following uses `swap.c` from the tarball.

To compile use the following command:

```
$ gcc -g -Wall swap.c -o swap
```

The -g option tells the compiler to produce debugging information. The -g option tells the compiler to save the *symbol table*, which is the list of memory addresses corresponding to your program's variables and lines of code. In gdb, the symbol table is used to allow you to refer to variable names and line numbers in a debugging session.

Your compiled code is still machine code. The symbol table makes it appear that your program in gdb is running source code rather than machine code. In fact, gdb is manipulating machine code, it only has the appearance of manipulating source code.

5 README

Your job is to write a man page for gdb, which you include in your README for the lab. The man page covers only those topics that are discussed in this lab. Make sure definitions are in your own words. Do not attempt to copy anything from this lab or from the actual gdb man page as New Mexico Tech's plagiarism policy will strictly apply.

For your man page, you should include these gdb commands as well as any shortcuts that can be used in their place:

- file
- backtrace
- run
- list
- help
- break
- continue
- step
- bt full
- print
- frame
- kill
- next
- info b
- disable/enable
- delete
- watch
- You should also explain what a conditional breakpoint is and how to use one.

6 Starting GDB

There are two ways to start gdb

One way is to type

```
gdb
```

at the command prompt. Doing this will get you the gdb command prompt `(gdb)`, but you haven't attached any program for gdb to debug. To load a program to debug:

(gdb) file swap

The other way is launch gdb with the program you want to debug

$ gdb swap

Try out both, and see which you like better.

7 The Call Stack and Stack Frames

Each time your program calls a function, the function is pushed onto a region of memory known as the call stack. Remember a stack is a LIFO (last in, first out) structure. Every time your program performs a function call, gdb keeps track of it and stores information about the it in a *stack frame*. The stack frame contains information that includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. So each stack frame represents a function call and as functions are called, the number of stack frames increases, and the stack grows. Conversely, as functions return to their caller, the number of stack frames decreases, and the stack shrinks.

Assuming you have loaded `swap` into gdb. At the gdb prompt type

(gdb) backtrace

You should see

```
(gdb) backtrace
No stack.
```

This makes sense as you haven't yet run the program and there is nothing in the call stack. To run the program type

`(gdb) run`

or

`(gdb) r`

Note: gdb uses shorthand for a number of commands, reducing them to a single letter: `p` for `print`, `n` for `next`, `s` for `step`, `i` for `info`, `b` for `break`, etc.

Your output should look something like this

```
(gdb) r
Starting program: /Users/scott/Labs/Lab5/swap
Reading symbols for shared libraries +....................... done
0 1 2 3 4 5 6
6 5 4 3 2 1 0

Program exited normally.
```

At the prompt, again type `backtrace`

(gdb) backtrace or (gdb) bt for short

The output again

```
(gdb) bt
No stack.
```

This makes sense as you ran the program, all the function calls where made and you exited the program. Hence, no call stack.

To see a call stack, what you need to do is stop or pause execution of the program so you can examine it. To do this you need to set a *breakpoint* which will stop/pause the program at a certain point. Among the many ways you can set breakpoints in gdb, two of the common ways is by using source code line numbers and function names.

Before you learn about breakpoints, it helps to understand how to display source code within gdb.

8 Viewing Source Code in GDB

There are a couple of ways to view source code in GDB.

To view source code in gdb, the command is list. This will show 10 lines of your source code. When you first execute list, it tries to displays your source code with `main` centered in its output.

```
(gdb) list
```

or

```
(gdb) l
```

```
(gdb) l
29    printf("%d ", d[i]);
30
31    printf("\n");
32
33    return;
34 }
35
36 int main()
37 {
38    int d[] = {0,1,2,3,4,5,6};
```

The numbers on the left hand side are your line numbers. If you keep entering the list command you will see the next ten lines.

```
(gdb) l
39    print_array(d,LEN);
40
41    swap_array(d,LEN);
42    print_array(d,LEN);
43
44    return 0;
45
46
47
48 }
```

And again

```
(gdb) l
Line number 49 out of range; swap.c has 48 lines.
```

How do you display other parts of the program with `list`?

Use the following commands:

starting with some line number `(gdb) list 5,`
ending with some line number `(gdb) list ,28`
between two line numbers: `(gdb) list 21,25`
by function name: `(gdb) list function`, where function is the name of the function you want to display.

Try these commands on your own. For example.

```
(gdb) l swap_array
10
11 return;
12 }
13
14 void swap_array(int d[], int len)
15 {
16 int i;
17
18 for(i = 0; i < len/2; i++)
19 swap(d,i,len-1-i);
```

9 Help on GDB

If you forget what the list options are you can always type `help list` or `h l` at the gdb prompt for specific help on the command `list`. Typing `help` at the gdb prompt will give you general help on gdb. **`Try it`**.

10 Breakpoints

Now that you can navigate around source code, you are ready to set breakpoints so you stop the execution of your program at various points. A breakpoint stops your program whenever a particular point in the program is reached.

Lets set a breakpoint somewhere in the main function.

To see the source code for main type

(gdb) l main

```
(gdb) l main
32
33 return;
34 }
35
36 int main()
37 {
38 int d[] = {0,1,2,3,4,5,6};
39 print_array(d,LEN);
40
41 swap_array(d,LEN);
```

Lets set a breakpoint at line 39.

(gdb) break 39

or

(gdb) b 39 for short.

```
(gdb) b 39
Breakpoint 1 at 0x100000e0e: file swap.c, line 39.
```

What does it mean to set a breakpoint at line 39? The program will pause execution between lines 38 and 39. That is everything up to line 38 will execute, but line 39 has not yet executed. The program is paused at that point.

Now type `run` or `r` at the gdb prompt. Run executes the program until reaches a breakpoint.

```
(gdb) r
Starting program: /Users/scott/Labs/Lab5/swap

Breakpoint 1, main () at swap.c:39
39 print_array(d,LEN);
```

At this point type `bt` at the gdb prompt to show the stack frame.

```
#0  main () at swap.c:38
d = {0, 1, 2, 3, 4, 5, 6}
```

If you just typed `bt` at the gdb command prompt, you will not see the same output as above. Use the `help` command and figure what option was given to `bt` to display the local variables in the stack frame. What is the correct command? (Make sure you include this command in your man page!)

Notice the stack frame #0 is currently the main function.

At the gdb prompt type `continue` or `c`. This will continue the program until it reaches the next breakpoint. Since there are no more breakpoints set, the program will exit normally. Type `r` again, gdb will stop again at line 39.

```
(gdb) continue
Continuing.
0 1 2 3 4 5 6
6 5 4 3 2 1 0

Program exited normally.
(gdb) r
Starting program: /Users/scott/courses/cse_113/Labs/lab6/swap

Breakpoint 1, main () at swap.c:39
39 print_array(d,LEN);
```

Now that we have the program stopped, type `step` or (you guessed it) `s`, which will just execute a single line of code. This is very useful as it allows you to *walk* through your code line by line to see what is happening.

```
(gdb) step
print_array (d=0x7fff5fbffb7c, len=7) at swap.c:27
27 for(i = 0; i < len; i++)
```

Since the line you "stepped" into was a function, gdb called the function. Lets look at the call stack by typing `bt full` at the command prompt.

```
(gdb) bt full
#0  print_array (d=0x7fff5fbffb7c, len=7) at swap.c:27
i = 1606420256
#1  0x0000000100000e24 in main () at swap.c:38
d = {0, 1, 2, 3, 4, 5, 6}
```

Notice, the call you are in is labeled #0. GDB has a strange way of numbering. #0 is always the top or current stack frame. The one that called #0 is the function labeled #1, in this case the main function called it.

Why does the local variable `i` in `print_array = 1606420256` in this output? Because it is uninitialized. Your value of `i` will probably vary from what is shown here.

To go through the `for` loop type `step`. You will keep stepping through the loop.

```
(gdb) step
28 printf("%d ", d[i]);
(gdb) step
27 for(i = 0; i < len; i++)
```

Another useful step command is `step n` where n is the number times you want to step before stopping.

In gdb just pressing ENTER (or RETURN) key will execute the last command. So if your last command was `step`, you can keep stepping by just hitting the ENTER key.

11 Displaying Variables

In gdb, you can easily view the values of your variables. Besides the basic types (`char, short, int, long, float, double`), you can also display arrays, structures, and unions. To look at what the values of `i` and `d[i]` are you use the `print` command.

(gdb) print variable_name

or

(gdb) p variable_name

for short

```
(gdb) p i
$1 = 0
(gdb) p d[i]
$2 = 0
```

The dollar sign notation ($n), where is n a number ($1, $2, etc) , is a way to have variable history. Every time you print a variable, n is incremented. The $ by itself refers to the last variable printed and $n refers to the n-th variable you printed.

So

```
(gdb) p $
$3 = 0
(gdb) p $1
$4 = 0
```

To find out what type the variable is use the command:

(gdb) ptype variable_name

or

(gdb) pt variable_name for short

```
(gdb) pt i
type = int
```

Why does `p d` print out a pointer, a memory address?

```
(gdb) p d
$5 = (int *) 0x7fff5fbffb7c
```

Remember, you passed in the array as an argument of the function. That means, you only got the address of the array not the array itself. If you wanted to display the whole array you would have to switch frames.

The command `frame` by itself just prints the current frame.

```
(gdb) frame
#0  print_array (d=0x7fff5fbffb7c, len=7) at swap.c:27
27 for(i = 0; i < len; i++)
```

To switch to a different frame, use `backtrace` to get the list of stack frames and then use `frame number`, where number corresponds to the frame you want to move to. The number of the stack frame is the leftmost column #n.

```
(gdb) bt
#0  print_array (d=0x7fff5fbffb7c, len=7) at swap.c:27
#1  0x0000000100000e24 in main () at swap.c:38
(gdb) frame 1
#1  0x0000000100000e24 in main () at swap.c:38
38 print_array(d,LEN);
(gdb) print d
$9 = {0, 1, 2, 3, 4, 5, 6}
(gdb) pt d
type = int [7]
```

Switch back to `frame 0`

```
(gdb) frame 0
#0  print_array (d=0x7fff5fbffb7c, len=7) at swap.c:27
27 for(i = 0; i < len; i++)
```

You can print out the address of a pointer.

```
(gdb) p &d
$6 = (int **) 0x7fff5fbffb48
```

and what the pointer points to by using the following:

```
(gdb) p *d
$7 = 0
```

The command `print/x` will print the value in hexadecimal

```
(gdb) p/x i
$14 = 0x1
```

Another very useful command is `info locals` to print all the local variables. Type `help info` to display a list of available info options. **Try it**.

You can also set variables to a specific value. `(gdb) p i = 6`, sets i to be 6. This is useful if you want to test a variable for a specific value.

Figure out the difference between the following cases.
Note: the command `kill` terminates the debugging session and returns you to a new debugging session, however breakpoints are preserved.

Case 1:

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) run
Starting program: /Users/scott/courses/cse_113/Labs/Lab5/swap

Breakpoint 1, main () at swap.c:38
38 print_array(d,LEN);
(gdb) step
print_array (d=0x7fff5fbffb7c, len=7) at swap.c:27
27 for(i = 0; i < len; i++)
(gdb) p i = 6
$23 = 6
(gdb) step
28 printf("%d ", d[i]);
(gdb) p i
$24 = 0
```

Case 2:

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) run
Starting program: /Users/scott/courses/cse_113/Labs/Lab5/swap

Breakpoint 1, main () at swap.c:38
38 print_array(d,LEN);
(gdb) step
print_array (d=0x7fff5fbffb7c, len=7) at swap.c:27
27 for(i = 0; i < len; i++)
(gdb) step
28 printf("%d ", d[i]);
(gdb) p i = 6
$25 = 6
(gdb) step
27 for(i = 0; i < len; i++)
(gdb) step
30 printf("\n");
```

12 Next

In gdb, `next` is similar to `step` but it skips over subroutines.

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) run
Starting program: /Users/scott/courses/cse_113/Labs/Lab5/swap

Breakpoint 1, main () at swap.c:38
38 print_array(d,LEN);
(gdb) next
0 1 2 3 4 5 6
39 swap_array(d,LEN);
```

13 Breakpoints Revisited

Lets `kill` the debugging session and some more breakpoints.

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) l main
32
33 return;
34 }
35
36 int main()
37 {
38 int d[] = {0,1,2,3,4,5,6};
39 print_array(d,LEN);
40
41 swap_array(d,LEN);
```

Lets set another breakpoint at line 41.

```
(gdb) b 41
Breakpoint 2 at 0x100000e33: file swap.c, line 41.
```

This sets a second breakpoint. To see the complete list of breakpoints type the command `info breakpoints`.

```
(gdb) info b
Num Type           Disp Enb Address            What
1   breakpoint     keep y   0x0000000100000e0e in main at swap.c:39
breakpoint already hit 1 time
2   breakpoint     keep y   0x0000000100000e33 in main at swap.c:41
```

The `Enb` field is telling you whether the breakpoint is enabled or not. To disable a breakpoint you use the `disable n` command where `n` is the number of the breakpoint. So to disable breakpoint one you would do this:

```
(gdb) disable 1
(gdb) info b
Num Type           Disp Enb Address            What
1   breakpoint     keep n   0x0000000100000e0e in main at swap.c:39
breakpoint already hit 1 time
2   breakpoint     keep y   0x0000000100000e33 in main at swap.c:41
```

Running the program again will just skip over the disabled breakpoints.

```
(gdb) r
Starting program: /Users/scott/courses/cse_113/Labs/Lab5/swap
0 1 2 3 4 5 6

Breakpoint 2, main () at swap.c:41
39 swap_array(d,LEN);
```

To enable a disabled breakpoint use the `enable n` command.

```
(gdb) enable 1
(gdb) info b
Num Type           Disp Enb Address            What
1   breakpoint     keep y   0x0000000100000e0e in main at swap.c:39
2   breakpoint     keep y   0x0000000100000e33 in main at swap.c:41
breakpoint already hit 1 time
```

If you want to delete a breakpoint, use the `delete n` command

```
(gdb) delete 1
(gdb) info b
Num Type           Disp Enb Address            What
2   breakpoint     keep y   0x0000000100000e33 in main at swap.c:41
breakpoint already hit 1 time
```

`clear` is another command to delete breakpoints. Type `help clear` to read about it.

Another way to set breakpoints is to use the function name you want to break at.

```
(gdb) break swap_array
Breakpoint 7 at 0x100000d0f: file swap.c, line 19.
(gdb) info b
Num Type           Disp Enb Address            What
7   breakpoint     keep y   0x0000000100000d0f in swap_array at swap.c:19
```

Now it's your turn. Run the program again, delete all breakpoints, and set a breakpoint on the function `swap_array`. Now lets run it again. Do your results look like this?

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /Users/scott/courses/cse_113/Labs/Lab5/swap
0 1 2 3 4 5 6

Breakpoint 7, swap_array (d=0x7fff5fbffb7c, len=7) at swap.c:19
18 for(i = 0; i < len/2; i++)
```

Look at the backtrace and keep stepping until you enter the swap function

```
(gdb) bt full
#0  swap_array (d=0x7fff5fbffb7c, len=7) at swap.c:20
i = 7
#1  0x0000000100000e38 in main () at swap.c:41
d = {0, 1, 2, 3, 4, 5, 6}
(gdb) step
19 swap(d,i,len-1-i);
(gdb) step
swap (d=0x7fff5fbffb7c, i=0, j=6) at swap.c:7
7 temp = d[i];
```

Do a backtrace once you are in the step function.

```
(gdb) bt full
#0  swap (d=0x7fff5fbffb7c, i=0, j=6) at swap.c:7
```

```
temp = 32767
#1  0x0000000100000d36 in swap_array (d=0x7fff5fbffb7c, len=7) at swap.c:20
i = 0
#2  0x0000000100000e38 in main () at swap.c:41
d = {0, 1, 2, 3, 4, 5, 6}
```

Notice how the numbers are changing. #0 is now swap, #1 is swap_array, and #2 is main. So to read the call stack, you can read that main called swap_array, and then swap_array called swap.
Also notice how the backtrace displays how the function was called, i.e. what the parameters were sent with the function call.

14 Watch Variables

Besides printing out variables you can set up watch variables, which pause your program whenever the value of a variable (or expression) being watched changes.

`kill` the debugging session and type `run` to a start a new session. When you enter the `swap_array` enter the command `watch variable_name`. In this case, you are entering `watch d[i]`, you are going to watch as the array changes its values.

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) run
Starting program: /Users/scott/courses/cse_113/Labs/Lab5/swap
0 1 2 3 4 5 6

Breakpoint 2, swap_array (d=0x7fff5fbffb7c, len=7) at swap.c:18
18 for(i = 0; i < len/2; i++)
(gdb) watch d[i]
Hardware watchpoint 4: d[i]
(gdb) step
19 swap(d,i,len-1-i);
(gdb)
swap (d=0x7fff5fbffb7c, i=0, j=6) at swap.c:7
7 temp = d[i];
(gdb)
8 d[i] = d[j];
(gdb)
Hardware watchpoint 4: d[i]

Old value = 1606422617
New value = 6
swap (d=0x7fff5fbffb7c, i=0, j=6) at swap.c:9
9 d[j] = temp;
(gdb)
11 return;
(gdb)
swap_array (d=0x7fff5fbffb7c, len=7) at swap.c:18
18 for(i = 0; i < len/2; i++)
(gdb)
Hardware watchpoint 4: d[i]

Old value = 6
```

```
New value = 1
0x0000000100000d3f in swap_array (d=0x7fff5fbffb7c, len=7) at swap.c:18
18 for(i = 0; i < len/2; i++)
```

If you keep stepping through the program (try it), you will eventually come to an output like this:

```
21 return;
(gdb)
Hardware watchpoint 4 deleted because the program has left the block
in which its expression is valid.
```

The watchpoint for `d[i]` is no longer valid. The reason for this is that watch variables are only valid for the *scope* that they are defined in.

15 Conditional Breakpoints

Stepping through each breakpoint of a program can be tiresome, especially if you know what can potentially trigger the error. One way to catch errors is through the use of conditional breakpoints, which let you set breakpoints based on a criteria you set.

In a text editor, edit swap.c and change line line 28 to read

`for (i = 0; i <= len; i++)`

That is, add an '=' to the conditional.

You do not need to quit gdb to make changes to your program. In a separate text editor make the changes to the source code. In a separate terminal window, compile the newly changed source code. Then return to your gdb session and type `run`. You should see something like this:

```
/Users/scott/cse113/Labs/lab5/swap' has changed; re-reading symbols.
```

Make the change to swap; recompile swap and in gdb `run` the program again. Notice the garbage at the end of the output. This is only happening since `i == len`, so we can set a *conditional* breakpoint to trigger if `i == len`

```
(gdb) break 29 if i == len
Breakpoint 1 at 0x100000d78: file swap.c, line 28.
(gdb) run
Starting program: /Users/scott/courses/cse_113/Labs/Lab5/swap
Reading symbols for shared libraries +....................... done

Breakpoint 1, print_array (d=0x7fff5fbffb7c, len=7) at swap.c:29
29 printf("%d ", d[i]);
```

What is the value of `i`? What is the value of `d[i]`? Is this valid?

16 What you need to turn in

In summary, you only need to turn in your man page as part of the README for the lab. It will replace the pseudocode for this week.

Part II

Now that you have some understanding of how `gdb` works it is time to apply that knowledge. You are given three programs that you have to debug.

For all programs, comment out the offending code (**do not delete it**) and replace it with correct code.

Before each line of corrected code place a one line comment //corrected your_name date

For example if Jane Smith fixed a line on 3/11/2015 the comment would read

//corrected jane smith 3/11/2015

Ignore compiler warnings until after you have debugged the program. However, you will turn in code that will compile without warnings.

Please note: You cannot use "script" to capture gdb output. You must physically copy and paste from the terminal into a text file!

17 Bad strings

Use bad.c from the tarball. Compile with debugging symbols and figure out out what is wrong.

For the correction make `msg` and `message` print the same thing. Do not change the declaration of `message` in any way. You need to add one line of code.

18 Recursive Function

Use `product.c` from the tarball. This recursive function should calculate the product of the numbers from a to b. That is, product(5,10) = 5 * 6 * 7 * 8 * 9 * 10= 151,200.

Fix the program and walk through the backtrace (the command `bt` is fine, no need for full option) for the recursive calls of product(5,10). Copy the (gdb) output of the backtraces to a text file and save. (You might notice that once your code has "exited normally" there will no longer be a stack. This is good! But in order to view the backtrace, set a breakpoint at your recursive call and step through your code in gdb until you are on your last recursive call. Then you can view the backtrace so you can see what your stack looks like right before exiting the program.)

19 Pointers

Use `pointer.c` from the tarball. Fix it so it works correctly. You need to change the code so that it use pointers correctly. **Do not add any new lines of code!** The output should be `0` and `1`.

20 Sort Easy

Use `sort_easy.c` from the tarball. This is a simple sorting function. See if you can find what is wrong with it. Correct the error and initial and date your correction.

21 Insertion Sort

Use `insert_sort.c` from the tarball and compile with debugging symbols.

The code, wrongly, uses an insertion sort to sort the data. You should read Wikipedia's article on insertion sort to get some background on the sort. The pseudo code for the algorithm `insert_sort.c` is:

```
Set y array to empty
get num_inputs from command line
set x array to hold data from command line

for i = 1 to num_inputs
   get new element (new_y)
   find first y[i] for which new_y < y[i]
   shift y[i], y[i+1], ... to the right to make room for new_y
   set y[i] = new_y
end for loop
```

`insert_sort.c` gets its input, the data to sort, from the command line. So prior to running it in gdb you first need to use the command

(gdb) r 12 5

to set the command line arguments you are going to provide `insert_sort.c`. In this case, you are only sorting two values 12 and 5. The correct output of the program should be 5 12.

When you first run `insert_sort` in gdb it will sit in an infinite loop

```
(gdb) run
```

Enter CTRL + C to return to the gdb prompt..

To begin testing, figure out what is wrong with the program using two values 12 and 5 `(gdb) r 12 5`. Once you solved that correctly pass the values 12 5 9 32 25 8 19 200 10 to the program (`(gdb) r 12 5 9 32 25 8 19 200 10`) and test. Find and fix the errors. The command `show args` will display the command line arguments you are passing to the program.

Start by setting a breakpoint in the main function on the `process_data()` call. The function `get_args(int ac, int **av)` works correctly.

Step through the code and find the errors. Correct them. You know you are finished when the program correctly sorts the set (12 5 9 32 25 8 19 200 10) and compiles without warnings. You are on you own for this one!

22 What you need to turn in

In summary, for part II you need turn these things in:

- bad.c corrected file
- product.c corrected file
- product.txt gdb output text file for product.c
- pointer.c corrected file
- sort_easy.c corrected file
- insert_sort.c corrected file
- README
  PLEASE NOTE! This README does not require any pseudo code!

Part III

These puzzles are From the *C Puzzle Book*. At first, try to solve these puzzles without compiling the source code.

1. What are the values of x,y, and z at the various print statements?

```
#include <stdio.h>

#define PRINT3(x,y,z) printf(#x "=%d\t" #y "=%d\t" #z "=%d\n",
 x, y, z)

int main(void)
{
        int x, y, z;

        x = y = z = 1;
        /* short-circuiting */
        ++x || ++y && ++z;
        PRINT3(x,y,z);

        x = y = z = 1;
        ++x && ++y || ++z;
        PRINT3(x,y,z);

        x = y = z = 1;
        ++x && ++y && ++z;
        PRINT3(x,y,z);

        x = y = z = -1;
        ++x && ++y || ++z;
        PRINT3(x,y,z);

        x = y = z = -1;
        ++x || ++y && ++z;
        PRINT3(x,y,z);

        x = y = z = -1;
        ++x && ++y && ++z;
        PRINT3(x,y,z);

        return 0;
}
```

2. What is the error in this code?

```
#include<stdio.h>

void OS_Solaris_print()
{
        printf("Solaris - Sun Microsystems\n");
}

void OS_Windows_print()
{
        printf("Windows - Microsoft\n");
}
void OS_HP-UX_print()
{
        printf("HP-UX - Hewlett Packard\n");
}

int main()
{
        int num;
        printf("Enter the number (1-3):\n");
        scanf("%d",&num);
        switch(num)
        {
                case 1:
                        OS_Solaris_print();
                        break;
                case 2:
                        OS_Windows_print();
                        break;
                case 3:
                        OS_HP-UX_print();
                        break;
                default:
                        printf("Hmm! only 1-3 :-)\n");
                        break;
        }

        return 0;
}
```

3. What's the expected output for the following program and why?

```
enum {false,true};

int main()
{
        int i=1;
        do
        {
                printf("%d\n",i);
                i++;
                if(i < 15)
                        continue;
        }while(false);
        return 0;
}
```

4. What do you think the output of the following program is and why? (If you are about to say "f is 1.0", think again)

```
#include <stdio.h>

int main()
{
        float f=0.0f;
        int i;

        for(i=0;i<10;i++)
```

```
                f = f + 0.1f;

        if(f == 1.0f)
                printf("f is 1.0 \n");
        else
                printf("f is NOT 1.0\n");

        return 0;
}
```

5. What is the mistake in this program?

```
#include <stdio.h>

int main()
{
        int a = 1,2;
        printf("a : %d\n",a);
        return 0;
}
```

6. What does this program print and why?

```
  #define SIZE 10
  void size(int arr[SIZE])
  {
          printf("size of array is:%d\n",sizeof(arr));
  }

  int main()
  {
          int arr[SIZE];
          size(arr);
          return 0;
  }
```

7. What is the differnce between the following function calls to scanf? Notice the space in the second call. Remove the space in the second call, what happens?

```
  #include <stdio.h>
  int main()
  {
      char c;
      scanf("%c",&c);
      printf("%c\n",c);

      scanf(" %c",&c);
      printf("%c\n",c);

      return 0;
  }
```

8. What is the output of the following program?

```
  #include <stdio.h>
  int main()
  {
      int i;
```

```
        i = 10;
        printf("i : %d\n",i);
        printf("sizeof(i++) is: %d\n",sizeof(i++));
        printf("i : %d\n",i);
        return 0;
}
```

9. The following is a simple C program to read a date and print the date. Why when you enter a date like 02-02-2012 it prints out 2-2-2012? What is the fix? Read `man 3 printf` and read about zero padding and field width. Fix it so it prints out 02-02-2012.

```
#include <stdio.h>
int main()
{
        int day,month,year;
        printf("Enter the date (dd-mm-yyyy) format including
        -'s:");
        scanf("%d-%d-%d",&day,&month,&year);
        printf("The date you have entered is %d-%d-%d\n",
        day,month,year);
        return 0;
}
```

10. This is supposed to print 20 - characters, but as written produces an infinite loop. By adding only one character you can correct the loop. What is the change?

```
#include <stdio.h>
int main()
{
        int i;
        int n = 20;
        for( i = 0; i < n; i-- )
                printf("-");

        printf("\n");
        return 0;
}
```

11. What is the output of the following program?

```
#include <stdio.h>
int main()
{
        int cnt = 5, a;

        do {
                a /= cnt;
        } while (cnt--);

        printf ("%d\n", a);
        return 0;
}
```

12. What is the output of the following program?

```
#include <stdio.h>
int main()
{
    int i = 6;
    if( ((++i < 7) && ( i++/6)) || (++i <= 9))
        ;
    printf("%d\n",i);
    return 0;
}
```

23 Requirements and Submission

This is part III of Lab 6. You are required to answer all 12 of these puzzles **in as much depth as possible**. Explain why you *expected* a certain outcome, and then justify whether you were right or wrong. Save your answers as a text file called `puzzles.txt`. Also, please remember that you cannot copy and paste from a PDF without consequences...

Please include your `puzzles.txt` file with the rest of your files for Lab 6. (You will see a lot of problems like this on the class final, so it is beneficial to you if you can do them without a computer!)

24 What you need to submit

- bad.c corrected file
- product.c, corrected file
- product.txt, gdb output text file for product.c
- pointer.c, corrected file
- sort_easy.c, corrected file
- insert_sort.c, corrected file
- README
  PLEASE NOTE! This README does not require any pseudo code! The README has your man page for gdb from Part I. For Part II, describe how you solved the three problems.
- puzzles.txt

Submit your files as a tarball to Canvas before the due date.

The name of your tarball should follow that of previous labs:
`cse113_firstname_lastname_lab6.tar.gz`