CS105 – Computer Systems Spring 2020

Assignment 3: Debugger Lab

Due: Tuesday, September 15, 2020 at 11:59pm

The purpose of this lab is to introduce you to the GDB debugger and to give you practice using this debugger to read, run, and debug programs in C and assembly. As with future labs, you should work in teams of two. Your partner will be assigned for this assignment.

This assignment will need to be completed on the course VM. If neither you nor your partner are able to access the VM, please put your name in the queue and get help from the course staff immediately. To get started, connect to the VPN, ssh to the course VM, and copy the files to your local directory on the VM.

Step 1: Connect to the Pomona VPN. If you do not already have the Pomona VPN set up on your machine, there are instructions for how to set-up the VPN here: `https://www.pomona.edu/administration/its/help/vpn`

Step 2: `ssh` into the course VM using the following command

```
% ssh username@pom-itb-cs2.campus.pomona.edu
```

(use your Pomona CAS username and password).

Step 3: Copy the starter code to your local directory and unpack it.

```
% cp /data/debuggerlab.tar .
% tar xvf debuggerlab.tar
```

You will now have a directory `debuggerlab-handout` which contains a `Makefile`, and three short C programs.

1 Data Representations in the Debugger

C has a commandline debugger called GDB (GNU Debugger). This is a very useful tool that you will come to know and love (or hate) this semester. For your first problem, you will use GDB to look at data at the bit- and byte-level.

Before we get started, open the file `q1.c` and take a look. This file contains three `static` constants and a short `main` function (our old friend, HelloWorld). The function is only there so that the program will compile; in this problem we are only concerned with the date. Compile the code using the Makefile (`make q1` or `make`). Note that we are compiling this program with the debugger flag `-g`. This is important for getting anything useful out of GDB.

Create a file called `q1.txt`, and put your answers to the following questions in it. Be sure put your name and your partner's name at the top of the file!

To get started, run your compiled program in GDB using the command `gdb q1`.

1. **print.** gdb provides you lots of ways to look at memory. For example, the command `print` will print the value in a variable. Try typ "`print puzzle1`". What is printed? Note that you can explicitly tell GDB how to interpret bytes in memory by adding a / after the command `print` (or its shortcut p) followed by a format. What do you get when you try "`p/x` puzzle1"? Is that more edifying? Note: for future references, there are several other format you can use: `x, d, u, o, t, a, c, f` (see GDB Quick Reference on the course website for details).

2. **examine.** So far, you've looked at puzzle1 in decimal and hex. There's also a way to treat it as a string, although the notation is a bit inconvenient. Recall that strings are just arrays of characters stored in memory, and that a "string" variable really only stores the address of (aka. a pointer to) the first character in the string. To try interpreting `puzzle1` as a string, we therefore need a way to look at (and interpret) the bytes in memory at address `&puzzle1` (recall that the "&" symbol means "address of"). The "`x`" (examine) command lets you look at arbitrary memory in a variety of formats and notations. This command takes three "arguments" after the /: the number of units you want to interpret, the size of each unit (b = one byte, h = 2 bytes, w = 4 bytes, g = 8 bytes), and the format you want to interpret each unit in (same format options as for `print` plus `s` and `i`). For example, "`x/1bx`" examines 1 unit of one byte interpreted as a hexadecimal value. Let's give this a try. Type "`x/4bx &puzzle1`". How does the output you see relate to the result of "`p/x puzzle1`"? (Incidentally, you can look at any arbitrary memory location with `x`, as in "`x/1wx 0x8048500`".)

3. **puzzle1.** OK, that was interesting (and maybe a bit weird), but we still don't know what's in `puzzle1`. Using what you know so far about GDB and about how value are represented, figure out what is the human-friendly value of `puzzle1`? Don't accept an answer that is partially garbage!

   Hint: Although `puzzle1` is declared as an `int`, it's not. But on our machine an `int` is 4 bytes, and those bytes could be interpreted as a different value of some other type.

   Hint: The most efficient way to do this is probably to on `puzzle1` with various forms of the `x` command. For example, you might try "`x/16i &puzzle1`" to display the bytes of memory starting at `&puzzle1` interpreted as a sequence of 16 assembly instructions.

   Hint: If you need help, gdb has help built in. Type "`help x`" to see more information about the options for the examine command.

4. **puzzle2.** Now we can move on to `puzzle2`. It pretends to be an *array* of `ints`, but you might suspect that it isn't. Using your newfound skills, figure out what is the human-friendly value?

   Hint: Since there are two `ints`, the entire value occupies 8 bytes.

5. **puzzle3.** We have one puzzle left. By this point you may have already stumbled across its value. If not, figure it out; it's often the case that in a debugger you need to make sense of apparently random data. What is stored in `puzzle3`?

2 Running Code in GDB

Create a file called `q2.txt`, and put your answers to the questions in Part A and Part B in it. Be sure to put both your name and your partner's name at the top of the file!

Part A

`q2a.c` contains a function that has a small `while` loop, and a simple `main` that calls it. Briefly study the `loop_while` function to understand how it works.

It will be useful to know what the `atoi` function (pronounced "a to i") does. Type "`man atoi`" in a terminal window to find out.

Compile the program using the Makefile (note that the debugger flag `-g` is set again). Run `gdb q2a` and set a breakpoint in `main` ("`b main`"). Tell `gdb` not to debug the `atoi` function by typing `skip atoi` (yes, ignore function pending future shared library load). Run the program by typing "`r`" or "`run`". The program will stop in `main`.

Then answer the following questions in your `q2.txt` file:

Note: to help you keep track of what you're supposed to doing, we have used italics to list the breakpoints you should have already set at the beginning of each step—except when they don't matter.

1. *Existing breakpoint at `main`.*
   Type "`c`" (or "`continue`") to continue past the breakpoint. What happens?

2. *Existing breakpoint at `main`.*
   Type "`bt`" (or "`backtrace`") to get a trace of the call stack and find out how you got where you are. What file and line number are you on?

   Note: Take note of the numbers in the left column. If you type "`up` *n*", where *n* is one of those numbers, you get to the *stack frame* corresponding to one of your function calls so that you can look at that function's variables. For example, you could try "`up 3`" followed by "`x/s argv[0]`" to see the 0th argument to main. (In general, you can use `up` and `down` to move up or down one frame in the stack.)

3. *Existing breakpoint at `main`.*
   Usually when bad things happen in the library it's your fault, not the library's. In this case, the problem is that `main` passed a bad argument to `atoi`. There are two ways to find out what the bad argument is: look at `atoi`'s stack frame (more on this next week!), or print the argument. Rerun the program by typing "`r`" and let it stop at the breakpoint. Note that `atoi` is called with the argument "`argv[1]`", sou can find out the value that was passed to `atoi` with the command "`print argv[1]`". What is printed? Given what you've discovered, why do you think the program segfaulted in step 1?

4. *Existing breakpoint at `main`.*
   Rerun the program with an argument of 5 by typing "`r 5`". Continue from the the breakpoint. What does the program print?

5. *Existing breakpoint at `main`.*
   Without restarting `gdb`, type “`r`” (without any further parameters) to run the program yet again. (If you restarted `gdb`, you must first repeat Step 4.) When you get to the breakpoint, examine the variables `argc` and `argv` by using the `print` command. For example, type “`print argv[0]`.” Also try “`print argv[0]@argc`”, which is `gdb`’s notation for saying “print elements of the `argv` array starting at element 0 and continuing for `argc` elements.” What is the value of `argc`? What are the elements of the `argv` array? Where did they come from, given that you didn’t add anything to the `run` command?

6. *Existing breakpoint at `main`.*
   The `step` or `s` command is a useful way to follow a program’s execution one line at a time. Type “`s`”. Where do you wind up?

7. *Existing breakpoint at `main`.*
   `gdb` always shows you the line that is about to be executed. Sometimes it’s useful to see some context. Type “`list`” What lines do you see? Hit the return key. What do you see now?

8. *Existing breakpoint at `main`.*
   Enter “`s`” to step to the next line. Then hit the return key three times. What do you think the return key does?

9. *Existing breakpoint at `main`.*
   What are the current values of `result`, `a`, and `b`?

Type “`quit`” to exit `gdb`. (You’ll have to tell it to kill the “inferior process”, which is the program you are debugging.)

Part B

Look at the file `q2b.c` This file contains three functions. Read the functions and figure out what they do. (If you’re new to C, you might need to consult a C book, some online references, and/or the course staff.) Here are some hints: recall that `argv` is an array containing the strings that were passed to the program on the command line (or from `gdb`’s `run` command); `argc` is the number of arguments that were passed. By convention, `argv[0]` is the name of the program, so `argc` is always at least 1. The `malloc` line allocates a variable-sized array big enough to hold `argc` integers (which is slightly wasteful, since we only store `argc-1` integers there, but ¯_(ツ)_/¯).

Once you understand what this code is doing, answer the following questions in your `q2.txt` file:

1. Open `q2b` in GDB. Set a breakpoint in `fix_array`. Run the program with the arguments `1 1 2 3 5 8 13 21 44 65`. When it stops, print `a_size` and verify that it is 10. Did you really need to use a `print` command to find the value of `a_size`? (**Hint:** look carefully at the output produced by `gdb`.)

2. *Existing breakpoint at `fix_array`.*
   What is the value of `a`?

3. *Existing breakpoint at `fix_array`.*
   Type "`display a`" to tell `gdb` that it should display `a` every time you stop. Step six times. You'll note that one of the lines executed is a right curly brace; this is common when you're in `gdb` and often indicates the end of a loop or the return from a function. After returning, what is the value of `a`?

4. *Existing breakpoint at `fix_array`.*
   Step again (a seventh time). What is the value of `a` now? What is `i`?

5. *Existing breakpoint at `fix_array`.*
   At this point you should (again) be at the call to `hmc_pomona_fix`. You already know what that function does, and stepping through it is a bit of a pain. The authors of debuggers are aware of that fact, and they always provide two ways to step line-by-line through a program. The one we've been using (`step`) is traditionally referred to as "step into"—if you are at the point of a function call, you move stepwise *into* the function being called. The alternative is "step over"—if you are at a normal line it operates just like `step`, but if you are at a function call it does the whole function just as if it were a single line. Let's try that now. In `gdb`, it's called `next` or just `n`. What line do we wind up at? (Incidentally, in `gdb` as in most debuggers, the line shown is the *next* line to be executed.)

6. *Existing breakpoint at `fix_array`.*
   Use `n` to step past that line, verifying that it works just like `s` when you're not at a function call. What's `a` now?

7. *Existing breakpoint at `fix_array`.*
   It's often useful to be able to follow pointers. `gdb` is unusually smart in this respect; you can type complicated expressions like `p *a.b->c[i].d->e`. (Recall that `*` dereferences a pointer. The `.` symbol access a field in a struct, and `x->y` is a shortcut for `(*x).y`) By this point, we have kind of lost track of `a`, and we just want to know what it's pointing at. Type "`p *a`". What do you get?

8. *Existing breakpoint at `fix_array`.*
   Often when debugging, you know that you don't care about what happens in the next three or twelve lines. You could type "`s`" or "`n`" that many times, but we're computer scientists, so we might prefer to avoid doing work that computers could do for us—especially mentally taxing tasks like counting to twelve. So on a guess, type "`next 12`". What line are you at?

9. *Existing breakpoint at `fix_array`.*
   What is the value of `a` now?

10. *Existing breakpoint at `fix_array`.*
    What is the value of `*a`?

3 Assembly-level Debugging

Usually you will be able to debug your programs using only the C source code, but sometimes it's necessary to inspect the assembly code. For this part, we will again use the program `q2b.c` from the previous problem. To be sure we're all on the same page, assemble that program using the Makefile ("`make q3`") and bring it up with `gdb q3`.

Create a file called `q3.txt` and put your answers to the following questions there. Be sure to put both your names at the top of the file!

1. Set a breakpoint in `main`. Run the program with arguments of `1 42 2 47 3`. Where does it stop? Type "list" to see what's nearby, then type "`b 29`" and "`c`". Where does it stop now?

2. *Existing breakpoints at `main` lines 26 and 29.*
   So since that's the start of the loop, typing "`c`" will take you to the next iteration, right? Oops. Good thing we can start over by just typing "`r`". Continue past that first breakpoint to the second one, which is what we care about. But why, if we're in the `for` statement, didn't it stop the second time? Type "`info b`" (or "`info breakpoints`" for the terminally verbose). Lots of good stuff there. The important thing is in the "address" column. Take note of the address given for breakpoint 2, and then type "`disassem main`". You'll note that there's a helpful little arrow right at breakpoint 2's address, since that's the instruction we're about to execute. Looking back at the corresponding source code, what part of the `for` statement does this assembly code correspond to?

3. *Existing breakpoints at `main` lines 26 and 29.*
   The code at `+33` jumps to `main+68`, which has two instructions that jump back to `main+35`. This is all part of the loop pattern we covered in class (in this case, a `for`). We've successfully breaked ("broken?" "Set a breakpoint?") at the initialization of the loop. But we'd like to have a breakpoint *inside* the for loop, so we could stop on every iteration. The jump to `main+35` tells us that we want to stop there. But that's not a source line; it's in the middle clause of the `for` statement. No worries, though, because `gdb` will let us set a breakpoint on *any* instruction even if it's in the middle of a statement. Just type "`b *(main+35)`" or "`b *0x400677`" (assuming that's the address of `main+35`, as it was when I wrote these instructions). The asterisk tells `gdb` to interpret the rest of the command as an address in memory, as opposed to a line number in the source code. What does "`info b`" tell you about the line number you chose? (Fine, we could have just set a breakpoint at that line. But there are more complicated situations where there isn't a simple line number, so it's still useful to know about the asterisk.)

4. *Existing breakpoints at `main` lines 26 and 29, and instruction `main+35`.*
   We can look at the current value of the array by typing "`p array[0]@argc`". But the current value isn't interesting. Let's continue a few times and see what it looks like then. Typing "`c`" over and over is tedious (especially if you need to do it 10,000 times!) so let's continue to breakpoint 3 and then try "`c 4`". What are the full contents of `array`?

5. *Existing breakpoints at `main` lines 26 and 29, and instruction `main+35`.*
   Perhaps we wish we had done "`c 3`" instead of "`c 4`". We can rerun the program, but we really don't need all the breakpoints; we're only working with breakpoint 3. Type "`info b`" to find out what's going on right now. Then use "`d 1`" or "`delete 1`" to completely get rid of breakpoint 1. But maybe breakpoint 2 will be useful in the future, so type "`disable 2`". Use "`info b`" to verify that it's no longer enabled ("Enb"). Continue past breakpoint 3, where we're stopped. Where do we stop next? (Hopefully that wasn't too much of a surprise!)

6. Sometimes, instead of stepping through a program line by line, we want to see what the individual instructions do. Of course, instructions manipulate registers. Quit `gdb` and restart it, setting a breakpoint in `fix_array`. Run the program with arguments of `1 42 2 47 3`. Type "`info registers`" to see all the processor registers in both hex and decimal. What flags are set right now? Note: sometimes it is not necessary to see all the registers. You can use the print commands `p` or `p/x` to print the value of an individual register (just remember to put a $ in front of the register name).

7. *Existing breakpoint at fix_array.*
   In question 1, we looked at lots of different ways to interpret data, but there is one that we didn't use: `x/i`. I particularly like "`x/16i $rip`". Try this command: what do you see? Compare that to the result of "`disassem fix_array`".

8. *Existing breakpoint at fix_array.*
   Finally, we mentioned stepping by instructions. That's done with "`stepi`" ("step one instruction"). Type that now, and note that `gdb` gives a new instruction address but still says that you're in the for loop. Hit return to `stepi` again, and keep hitting return until the displayed line `doesn't` contain a hexadecimal instruction address. Where are you?

9. *Existing breakpoint at fix_array.*
   It's useful to use "`x/16i $rip`" or `disassem fix_array` here to make sure we understand what's about to happen. You should see a `mov` instruction followed by an absolute jump, followed by a conditional jump, followed by another `mov` followed by a `call`. Use `stepi 5` to get past all of this. What instruction address will be executed next?

10. *Existing breakpoint at fix_array.*
    As with source-level debugging, at the assembly level it's often useful to skip over function calls. At this point you have a choice of typing "`stepi`" or "`nexti`". If you type "`stepi`", what do you expect the next instruction to be (hexadecimal address)? What about "`nexti`"? (By now, your debugging `gdb` skills should be strong enough that you can try one, restart the program, and try the other if you want to.)

And now you know everything you need to know about debugging with GDB!

4 Feedback

Please create a file called `feedback.txt` that answers the following questions:

1. How long did each of you spend on this assignment?

2. Any comments on this assignment?

How you answer these questions **will not affect your grade**, but whether you answer them will.

Submission

You should submit four files on the course submission page: `q1.txt`, `q2.txt`, `q3.txt`, and `feedback.txt`. Before you do so, double check that both your name and your partner's name are at the top of all the files.

Only one member should submit your solution files, and all files should be submitted together in a a single submission (hint: command-click is your friend). You may, of course, submit updates to your work; just be sure that everything is submitted by the same team member. All three problems on this assignment will be weighted equally; you will get 2 points for submitting your feedback file. Your score will be based on a total of 32 points.