CS105 – Computer Systems Spring 2020

Assignment 1: C Lab

Due: Friday, January 31, 2020 at 11:59pm

The purpose of this lab is to introduce you to how data are stored in memory, and to give you practice reading, writing, and debugging programs in the C language. It will also introduce you to the "C mind-set," which may be significantly different from the way you are used to thinking about programming.

In many ways, the C language is like Java. The syntax for variable and function declarations, assignment statements, `for` and `while` loops, and `if`-statements are the same in both languages. The big difference is that in C we have a different view of data, one that is closer to the actual hardware. We must be aware of where in memory values are located and how much space they occupy. The exercises in this laboratory assignment will give you practice in thinking about variables, pointers, and structs—and how they relate to addresses and values in memory.

As with future labs, you should work in teams of two. Your partner will be assigned for this assignment.

To get started, `ssh` into `pom-itb-cs2.campus.pomona.edu` (use your pomona CAS credentials, not your lab credentials). You will then need to acquire the starter files; conveniently they are already on the server located in `/data/clab-handout.tar`. The files are also available on the course website. Create a (protected) directory in which you plan to do you work, go there, and copy the starter files to there. Then, unpack the tar file with the command

```
% tar xvf clab.tar
```

You will now have a directory `clab` which contains this writeup, a `Makefile`, and six short C programs.

Note: The lab filesystem is broken today, so you will need to do all your work in a commandline editor. Personally, I use Emacs. If you are already use another commandline editor (e.g., Vim), feel free to use that.

To start, open the Makefile. Makefiles are files that contain a set of rules. Each rule has the form

```
target: dependencies
        system command(s)
```

The target is the name of the directive. This is often, but not always, the name of a file to be generated. The dependencies are the files that are used as input for the system commands. The system commands are a sequence of one or more command line instructions that will be executed when you make that target. (Note that the system commands must be indented with a tab).

I find Makefiles very useful, and I use them for everything, including compiling programs, pushing to git repos, and updating my website. In this Makefile, most of the rules are commands for compiling the C programs you will use in this assignment. Note that the basic command for compiling c programs is `gcc <filename.c>`; this produces an executable file `a.out`. Here we have added the option `-o filename` to each of the commands; this produces an executable file with the specified name, instead of naming it `a.out`.

Typing `make` will execute the rule `all`. (Try it!) In this case, that rule doesn't do anything interesting. You can specify a rule to execute by typing `make <target>`.

1 Data Representations

Take a look at the program `q1.c`. This program reads six integers from the keyboard (known as `stdin`) using the function `scanf` and stores them in an array. (Take a minute and see if you can figure out how it is doing this!). It then interprets the first four integers as a string and the last two as a `double` and prints the results. Your job is to find integers that will cause the program to print some specified values. Do not change the program's source.

Begin by compiling the program using the command `make q1`. Next, create a text file named `q1.soln` with six integers, one to a line. Begin by making them all zero. Put your names on the seventh and eighth lines. Run the `q1` program with the input redirected from the `q1.solution` file.

```
% ./q1 < q1.soln

0.0000000000000000
```

The blank line in the output shows that the string is empty. The `double` is zero. As a further warm-up, change the first of the six integers to 14132. The string now has two characters—which happen to be digits. (It is a string of characters, not a number!) The `double` is still zero.

```
% ./q1 < q1.soln
47
0.0000000000000000
```

Your task Fill `strings.solution` with six integers to produce this result.

```
% ./q1 < q1.soln
Cecil Sagehen
3.1415926535897931
```

Hints and suggestions It is possible, but long and tedious, to compute by hand the four integers corresponding to "Cecil Sagehen." But it is not practical (or a good use of your lifespan!) to compute the two integers corresponding to the decimal expansion of π. Think about writing a short program, separate from `strings.c`, that will calculate the integers for you.

Reflections The actual values of the six integers are not important. If they were all you cared about, you could ask someone in the lab. Be sure that you understand what is happening with the bytes in memory. Also, take some time to understand the pointer arithmetic and type casts in the source file `q1.c`.

Note: The six-integer sequence you produced is not unique. Other sequences will produce the same result. How many different solutions are there?

2 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 second problem, you will use GDB to look at data at the bit- and byte-level.

Before we get started, open the file `q2.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. (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 `q2.txt`, and put your answers to the following questions in it. To get started, run your compiled program in GDB using the command `gdb q2`.

1. `gdb` provides you lots of ways to look at memory. For example, type "`print puzzle1`" (something you should already be familiar with). What is printed? Sometimes it's worth trying different ways of exploring things. How about "`p/x` puzzle1"? What does that print? Is it more edifying?

2. You've just 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. The "`x`" (examine) command lets you look at arbitrary memory in a variety of formats and notations. For example, "`x/bx`" examines bytes in hexadecimal. Let's give that a try. Type "`x/4bx &puzzle1`" (the "&" symbol means "address of"; it's necessary because the `x` command requires addresses rather than variable names). 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/wx 0x8048500`".)

3. OK, that was interesting (and maybe a bit weird), but we still don't know what's in `puzzle1`. We need help! And fortunately `gdb` has help built in. So type "`help x`". Then experiment on `puzzle1` with various forms of the `x` command. For example, you might try "`x/16i &puzzle1`". (`x/16i` is one of our favorite `gdb` commands—but since here we suspect that `puzzle1` is data, not instructions, the results might be interesting but probably not correct.) Keep experimenting until you find a sensible value for `puzzle1`. (Hint: Although `puzzle1` is declared as an `int`, it's not. But on our machine an `int` is 4 bytes, 2 halfwords, or one—in `gdb` terms—word.) What is the human-friendly value of `puzzle1`? (Don't accept an answer that is partially garbage!)

4. Now we can move on to `puzzle2`. It pretends to be an *array* of `int`s, but you might suspect that it isn't. Using your newfound skills, figure out what it is. (Hint: since there are two `int`s, the entire value occupies 8 bytes. What is the human-friendly value?)

5. 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`?

3 Running Code in GDB

Create a file called `q3.txt`, and put your answers to the questions in Part A and Part B in it.

Part A

`q3a.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 q3` and set a breakpoint in `main` ("`b main`"). Tell `gdb` not to debug the `atoi` function by typing `skip atoi`. Run the program by typing "`r`" or "`run`". The program will stop in `main`. (Ignore any warnings; they're meaningful but we'll work around them.)

Then answer the following questions in your `q3.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. Take note of the numbers in the left column. Type "`up` *n*", where *n* is one of those numbers, to get to `main`'s *stack frame* so that you can look at main's variables. (In general, you can use `up` and `down` to move up or down one frame in the stack.) What file and line number are you on?

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 in step 2, we saw that the problem occurred when `atoi` was called with the argument "`argv[1]`". You 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 `q3b.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: `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 `q3.txt` file:

1. Open `q3b` 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 (more on that later). `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, and CS types sneer at work that computers could do for them—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`?

4 Arrays and Pointers

This part of the assignment is an exploration into how arrays are stored in memory. Suppose that we have a two-dimensional 4×7 array `tda` of integers whose values encode the indices. That is, `tda[i][j]` has the value $10i + j$. If we print the array row-by-row we obtain this:

```
00 01 02 03 04 05 06
10 11 12 13 14 15 16
20 21 22 23 24 25 26
30 31 32 33 34 35 36
```

Two-dimensional arrays are stored in row-major order, so in memory the array looks like this:

```
00 01 02 03 04 05 06 10 11 12 13 14 15 16 20 21 22 23 24 25 26 30 31 32 33 34 35 36
```

Hint: How would you use GDB to see those values in memory?

If we take that block of memory—without changing the values stored there—and consider it as a 7×4 array, we get this:

```
00 01 02 03
04 05 06 10
11 12 13 14
15 16 20 21
22 23 24 25
26 30 31 32
33 34 35 36
```

The same memory can be considered a one-dimensional 28-element array, a two-dimensional 4×7 array, a two-dimensional 7×4 array, a three-dimensional $7 \times 2 \times 2$ array, and so on.

There is another way to represent two-dimensional arrays—as arrays of arrays. The idea is to have a one-dimensional array of *rows.* Each element in that array is a pointer that points to the beginning of another one-dimensional array which contains the data values of that particular row. With this representation, the 7×4 array shown above would be an array of seven pointers, each pointing to an array of four integers. The strategy uses a little more memory (for the pointers), but it is more flexible. The rows need not all be the same size.

Your task Your starter materials contains an almost-complete program `q4.c`. It declares and initializes a 4×7 array `tda` like the one above. It also declares an array `aoa` of seven rows. Your job is to fill in seven assignment statements in the `main` function so that `aoa` is a 7×4 array like the one above, except with the rows in reverse order.

Make no changes to the program, except to add your names and complete the assignment statements. When you are ready, compile and run your program If your assignments are correct, you will see the output shown below, showing the original array and the modified one.

```
00 01 02 03 04 05 06
10 11 12 13 14 15 16
20 21 22 23 24 25 26
30 31 32 33 34 35 36

33 34 35 36
26 30 31 32
22 23 24 25
15 16 20 21
11 12 13 14
04 05 06 10
00 01 02 03
```

Hints and suggestions Avoid trial and error. Think about the starting points of the various rows of the result, relative to the beginning of `tda`. You can solve this using either array notation or pointer arithmetic.

Reflections Notice that the assignments to aoa are made *before* tda is initialized. Why does that work? Notice also that the bodies of print_two_dim_array and print_array_of_arrays are character-for-character *identical.* Why is it necessary to have two functions? Be sure that you understand the difference between the two ways of representing two-dimensional arrays.

5 Implementing a C Program

You now know everything you need to know about data representations and debugging C code. Time to start programming! For this part of the lab, you will complete a simple linked list program. It is the sort of exercise that you may have done in a data structures course. We have given you a partially-complete program q5.c. Open it up and look inside.

This file defines a type struct cell, that is a structure (the C equivalent of an object) with two members, value and next. Since working with a type named struct cell is inconvenient, this code renames the type to cell_t using the keyword typedef.

For us, a *list* is a pointer to a cell_t (the first cell in the linked list). The next field in the structure, a pointer to a cell_t, is *the rest of the list.* The empty list is a pointer whose value is NULL.

Looking in this file, you will see that three functions—append, printlist, main—are already implemented for you. main creates a couple of lists and calls all the other functions in the program. printlist prints the elements of the list. append creates a new cell_t with the specified value and places it at the end of the specified list.

Your task Implement the following three functions:

```
void makeempty(cell_t** thelist)
void prepend(int newvalue, cell_t** thelist)
void reverse(cell_t** thelist)
```

All three functions take a pointer to a list (a cell_t**) and operate on that list. The function makeempty removes and recycles all the elements of the given list. The function prepend creates a new cell_t with the specified value and places it at the front of the list. The function reverse reorders the elements in the list. It does so by moving the elements of the list, not by creating new copies of elements.

Do not change the functions append, printlist, and main.

When you have finished, compile and run the program. Make sure that the output is correct by comparing it to the listing in Figure 1.

What you need to know Pointers are used to refer to elements in the list. Remember that a pointer simply holds an address in memory. If you want it to point to something useful, you must allocate space in memory and set the pointer to the address of that space. Here is how to create an element for a list:

```
cell_t *p = (cell_t *) malloc(sizeof(cell_t));
```

You can then initialize the fields of the list element by assigning to p->value and p->next. Keep in mind the distinction that p is the address in memory of an element and *p is the element itself. You will need to allocate space for new elements in prepend.

```
% ./q5
backward
9, 0x971130
8, 0x971110
7, 0x9710f0
6, 0x9710d0
5, 0x9710b0
4, 0x971090
3, 0x971070
2, 0x971050
1, 0x971030
0, 0x971010
backward reversed
0, 0x971010
1, 0x971030
2, 0x971050
3, 0x971070
4, 0x971090
5, 0x9710b0
6, 0x9710d0
7, 0x9710f0
8, 0x971110
9, 0x971130
empty
forward
0, 0x971130
1, 0x971110
2, 0x9710f0
3, 0x9710d0
4, 0x9710b0
5, 0x971090
6, 0x971070
7, 0x971050
8, 0x971030
9, 0x971010
forward reversed
9, 0x971010
8, 0x971030
7, 0x971050
6, 0x971070
5, 0x971090
4, 0x9710b0
3, 0x9710d0
2, 0x9710f0
1, 0x971110
0, 0x971130
empty again
```

Figure 1: The correct output for the program q5. The addresses, the hexadecimal values after the commas, may differ in your output, but they should be spaced apart by the same amounts.

When an element is created with malloc, it lasts until it is explicitly disposed, or until the program ends. Failing to explicitly dispose of memory when you are done with it is called a *memory leak* and can hurt the performance of your program. Note: This is different from languages like Python and Java which perform *garbage collection*, that is they automatically free memory for you. To dispose of an element and recycle the memory that has been allocated to it, make a call to free with a pointer to the element.

```
free(p);
```

The value of the pointer must have come from a call to malloc. There should be only one call to free for each call to malloc (double-freeing can cause a security vulnerability!). In makeempty, you will need to free all the elements in the given list.

Pay special attention to the list argument in the functions you write. A list for us is a pointer to cell_t. The argument to your functions is a *pointer to a list,* of type cell_t**. In the function

```
void makeempty(cell_t** thelist)
```

the variable thelist is a pointer to a list. That is, it is the address of a place in memory that holds the address of the first element of the list. The reason for using a double pointer is so that the function makeempty can change the value of the list back in the caller. The last act of makeempty (after it has recycled the memory of all the original list elements) is to make the assignment

```
*thelist = NULL;
```

so that the caller's list will now be empty.

Hints and suggestions Do not get carried away! You are to write three functions, and each one will be no more than 15 lines long, often shorter. On the other hand, programming with pointers is delicate work. The few lines of code that you write must be precisely correct.

Reflections The function printlist prints the addresses of the various list elements. One reason for that is to be able to check that reverse creates a list with *the very same elements,* just in a different order. You can use those addresses to determine how many bytes are used for each cell_t. How many bytes are actually required by the data in the structure? How many bytes are actually used by the system? (Hint: the answers are different.)

6 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 six files on the course submission page: `q1.soln`, `q2.txt`, `q3.txt`, `q4.c`, `q5.c`, and `feedback.txt`. Before you do so, double check that both your name and your partner's name are at the bottom of `q1.soln` and at the top of all the other 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 five questions 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 52 points.