CS105 – Computer Systems Fall 2020

Assignment 1: C Lab

Due: Tuesday, September 1, 2020 at 11:59pm

The purpose of this lab is to introduce you to the C language and to give you additional practice with data representations in C.

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.

This lab is divided into three parts. You should begin each part at the appropriate point during lab. If you do not finish the full assignment during lab, you and your partner should arrange to finish the assignment together outside of class.

1 Your First C Program

Part A

When instructed to do so, open a file called part1.c in your favorite text editor (my favorite is Emacs). In that file, define a function that takes two integers as arguments and returns an integer. If the second argument is greater than or equal to the first, your function should return the sum of the integers between the two numbers given as arguments (inclusive). Otherwise the function should return -1.

Part B

Add a main function to your program in the file part1.c that calls your first function with two (hardcoded) values and prints the value returned by that function.

Next, you will need to compile and run your program. If you have a C compiler installed on your computer (which you probably do if you are working on OS X or any version of Linux), then you should be able to compile your code by entering the following command on the commandline:

```
% gcc -o part1 part1.c
```

You will the be able to run your program with the command

```
% ./part1
```

If you are running Windows or are unable to compile and run C programs for any other reason, you best option is to complete this assignment on a Pomona machine. This will require a little extra effort to get started; but don't worry, everyone else will need to do this next week, so you are really just getting a head start.

Step 1: Connect to the Pomona VPN. If you are a Pomona student and 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` If you are an off-campus student, use your partner's credentials for next week. I am working with ITS to get accounts set up for you, but unfortunately that isn't up and running yet.

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).

Note: you can copy files from your local machine to the VM using the following command:

```
% scp filename username@pom-itb-cs2.campus.pomona.edu:~
```

where filename is the name of the file (or the path to the file) you want to copy. Note that this needs to be done from your local machine; this command will not work from the VM.

Step 3: Compile and run your code on the VM.

2 Arrays and Pointers

At this point, you will then need to acquire the starter files for Parts 2 and 3. If you are working on the VM, they are conveniently already located on that machine in the file `/data/clab-handout.tar`. Copy that file to your current location with the following command

```
% cp /data/clab.tar .
```

If you are working on your local machine, you can download the starter files from the course website.

Once you have a local copy of the tar file, unpack the tar file with the command

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

You will now have a directory `clab` which contains this writeup, a `Makefile`, a tester file for Part 2, and a starter file for Part 3.

You are now ready to start on Part 2. To begin, open a file called part2.c in your favorite editor. In that file, write a program that computes the pair of unsigned integers x, y such that the array [x, y] has the same binary representation as the string "CS 105!".

To test your answer, you will need to compile and run the tester file. To make this easier, I've provide a Makefile for you. Before you compile anything, open the Makefile and take a look. 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>`. In this case, you can compile the tester using the command

```
% make part2
```

You can then run the program part2-tester. This program expects you to enter two integers (one per line) and will print out the string with the corresponding representation. If your integers are correct, it should print out the string "CS 105!"

3 Implementing a Linked List

You now know everything you need to know about writing C code, and you should be ready to starting writing a more interesting program. For the final part of this 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 `part2.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`.

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.)

```
% ./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 `part3`. The addresses, the hexadecimal values after the commas, may differ in your output, but they should be spaced apart by the same amounts.

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: `part1.c`, `part2.c`, `part3.c`, and `feedback.txt`. Before you do so, double check that both your name and your partner's name are in a comment at the top of each of your files.

Only one member should submit your solution files (but remember to tag your partner as your collaborator!), 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.