Lab 1: C

Deadline: Monday, January 30, 11:59:59 PM PT

Before you come to lab 1, make sure that you are comfortable with either editing your files on the hive by using your text editor of choice. This lab is designed to familiarize you with basic C concepts and prepare you for project 1.

We expect your submission to follow the spirit of the question. If your submission does not follow the spirit of the question (e.g. hardcodes the correct output while not applying any of the concepts introduced in this lab), we may manually grade your submission. If you feel like you may potentially be violating the spirit of the question, feel free to make a private question on Ed or ask your lab TA.

The slides used during lab sections are here .

Setup

> You must complete this lab on the hive machines. See Lab 0 for a refresher on using them.

In your `labs` directory, pull the files for this lab with:

```
$ git pull starter main
```

If you get an error like the following:

```
fatal: 'starter' does not appear to be a git repository
fatal: Could not read from remote repository.
```

make sure to set the starter remote as follows:

```
$ git remote add starter https://github.com/61c-teach/sp23-lab-starter.git
```

and run the original command again.

Exercise 1: Hello!

Compiling and Running a C Program

In this lab, we will be using the command line program gcc to compile programs in C. Use the following command to compile the code for exercise 1 (make sure that you have `cd`ed into the proper directory)

```
$ gcc ex1_hello.c
```

This compiles `ex1_hello.c` into an executable file named `a.out`. If you've taken CS61B or have experience with Java, you can kinda think of `gcc` as the C equivalent of `javac`. This file can be run with the following command:

```
$ ./a.out
```

The executable file is `a.out`, so what is the `./` for? Answer: when you want to execute an executable, you need to prepend the path to the name of the executable. The dot refers to the "current directory." Double dots (`..`) would refer to the directory one level up.

`gcc` has various command line options which you are encouraged to explore. In this lab, however, we will only be using `-o`, which is used to specify the name of the executable file that `gcc` creates. By default, the name of the executable generated by `gcc` is `a.out`. You can use the following commands to compile `ex1_hello.c` into a program named `ex1_hello`, and then run it. This is helpful if you don't want all of your executable files to be named `a.out`.

```
$ gcc -o ex1_hello ex1_hello.c
$ ./ex1_hello
```

At this point, you should see the string `Hello World` printed out. If you edit the source code (such as `ex1_hello.c`), you must recompile the program using `gcc` to produce a new executable, otherwise the executable will still run the old source code.

Edit `ex1_hello.c` with your editor of choice and make the program print out the string "Hello 61C" instead of "Hello World". Make sure to save your edited file, but do NOT recompile yet.

Run the executable with `./ex1_hello`. You should still see `Hello World`, which is the old output.

Now, **recompile** your program with `gcc -o ex1_hello ex1_hello.c`, and then **run** the executable with `./ex1_hello` again. You should now see `Hello 61C`.

Review: Pointers

A pointer is a variable whose value is the memory address of another variable. Note that every variable declaration is always located in a memory, where every element has a corresponding address. Think of it like an array: every variable value is contained on a specific array index (address), and the pointer to that variable is another variable within that same array that contains the index (address) of the variable it is pointing at.

Consider the following example:

```
int main() {
    int my_var = 20;
    int* my_var_p;
    my_var_p = &my_var;
}
```

For the first line, we declared an `int` variable called `my_var` which is then assigned with a value of 20. That value of 20 will be placed somewhere in the memory.

For the second line, we declared an `int` pointer variable called `my_var_p`. Note that you can also write `int *my_var_p`, where the asterisk glued to the variable name instead of the variable type.

For the third line, we assigned `my_var_p` to have a value that is equal to the address of `my_var`. This is done by using the `&` operator before the `my_var` variable. At this point, the value contained in the variable `my_var_p` is the address in memory of the variable `my_var`.

Note that whenever you want to change the value of `my_var`, you could do it by changing `my_var` directly.

```
my_var += 2;
```

Alternatively, you could also change the value of `my_var` by dereferencing `my_var_p`

```
*my_var_p += 2;
```

In a nutshell, `&x` gets the address of `x`, while `*x` gets the contents at address `x`.

Here's a more complete example:

```
int main() {
    int my_var = 20;
    int* my_var_p;
    my_var_p = &my_var;

    printf("Address of my_var: %p\n", my_var_p);
    printf("Address of my_var: %p\n", &my_var);
    printf("Address of my_var_p: %p\n", &my_var_p);

    *my_var_p += 2;

    printf("my_var: %d\n", my_var);
    printf("my_var: %d\n", *my_var_p);
}
```

A sample execution of this code gave out the following:

```
Address of my_var: 0x7fffebafb32c
Address of my_var: 0x7fffebafb32c
Address pf my_var_p: 0x7fffebafb330
my_var: 22
my_var: 22
```

The first line prints out the value of `my_var_p`, which was assigned to the address of the variable `my_var`.

The second line shows that `my_var_p` is indeed equal to `&my_var`, the address of the variable `my_var`.

The third line prints out the address of `my_var_p`. Note that since `my_var_p` is in fact a variable itself (the variable type is an int pointer), therefore it has to be placed somewhere in the memory as well. Thus, printing out `&my_var_p` allows us to see where in the memory the `my_var_p` variable is located.

After the first three print outs, we changed the value of `my_var` indirectly using `*my_var_p`. Since `my_var_p` is a pointer to `my_var` (i.e. `my_var_p` is the address of `my_var`), performing `*my_var_p` allows us to modify the contents at the address in `my_var_p`.

The fourth line shows that we have indeed modified `my_var`, since the value is now 22.

The fifth line confirms that `*my_var_p` is indeed equal to `my_var`.

▶ What happens if we did the following: `my_var_p += 2`?

▶ After doing `my_var_p += 2` earlier, what is the value of `&my_var_p`?

▶ After doing `my_var_p += 2` earlier, what happens then if we try to print the value of `*my_var_p`?

Exercise 2: Pointer Basics

Edit `ex2_pointer_basics.c` using your editor of choice and fill in the blanks. Feel free to refer back to the pointer review section if you are stuck.

Compile and run the program and **check** that the output matches what you expect. If you need a refresher on `gcc`, please refer back to exercise 1.

Exercise 3: Pointers and Functions

Edit `ex3_pointers_and_functions.c` using your editor of choice and fill in the blanks. Feel free to refer back to the pointer review section if you are stuck.

Compile and run the program and **check** that the output matches what you expect. If you need a refresher on `gcc`, please refer back to exercise 1.

Exercise 4: Pointers to Stack vs Heap

Edit `ex4_ptr_heap_stack.c` using your editor of choice and fill in the blanks. Feel free to refer back to the pointer review section if you are stuck.

Compile and run the program and **check** that the output matches what you expect. You will see a compiler warning. What does it mean?

Read the output of `gcc`. Note that it throws an error about the address of a stack variable being returned. Similarly, the output of the program should show address of the stack variable as `(nil)`, indicating that it is not usable. This is because `x` cannot be accessed outside of the function `int_on_stack` using a pointer. In the future, make sure to allocate memory on the heap if you'd like to use it later.

Review: Arrays

An array is a fixed length data structure that can hold one or more elements of the same type. Unlike lists, arrays do not resize automatically when you add an element.

In C, arrays are represented as a pointer to the first element. Each element of the array is stored in memory and they are stored in contiguous memory locations (side by side). Because arrays are represented only with a pointer to the first element, the pointer itself is not enough to deduce the length of the array. If you need to keep track of the length of an array, you must use another variable.

You can perform arithmetics on pointers to access different elements of the array. Recall that a pointer is just an address, so if you add to or subtract from the address, you can get the address of later or earlier elements of an array.

Exercise 5: Arrays

Edit `ex5_arrays.c` using your editor of choice and fill in the blanks. Feel free to refer back to the arrays section if you are stuck.

Compile and run the program and **check** that the output matches what you expect.

Read the output of your program. Note that the relationship between the address of the start of the array and the address of index 2 (hint: they're two bytes apart).

Review: Pointer Arithmetic

In exercise 5, your program performed basic pointer arithmetic when you computed the address of index 2. This worked because the size of each element is 1 byte (since the size of `int8_t` is 1 byte). However, most types you'll work with take up more than 1 byte in memory.

When performing pointer arithmetic, C automatically accounts for the type of the pointer and adds the correct number of bytes. For example, if you write `ptr + 5`, C will not always add 5 to `ptr`. Instead, C will add 5 times the size of the datatype that `ptr` points to. If `ptr` was an `int*` and `int`s take up 4 bytes in memory, `ptr + 5` adds 20 to the address held in `ptr`.

Exercise 6: Pointer Arithmetic

Edit `ex6_pointer_arithmetic.c` using your editor of choice and fill in the blanks. Feel free to refer back to the arrays review section if you are stuck. Your solution should be similar to exercise 5.

Compile and run the program and **check** that the output matches what you expect.

Read the output of your program. Note that the relationship between the address of the start of the array and the address of index 2 is **different** than the relationship in the previous exercise.

Review: Strings

In C, strings are represented as an array of `char`s. Strings are a special type of `char` arrays because they always end in a null terminator (`\0`). Recall that arrays in C do not contain any information about their length, so the null terminator allows us to determine when the string ends.

When allocating memory for a string, there must be enough memory to store the characters within the string **and** the null terminator. Otherwise, you might run into undefined behavior. However, the array could be larger than the string it stores.

C has a library of functions for manipulating strings, such as:

- `strlen`: computes the length of a string by counting the number of characters before a null terminator
- `strcpy`: copies a string from one memory location to another, one character at a time until it reaches a null terminator (the null terminator is copied as well)

Exercise 7: Strings

Edit `ex7_strings.c` using your editor of choice and fill in the blanks. Feel free to refer back to the strings review section if you are stuck.

Compile and run the program and **check** that the output matches what you expect.

Exercise 8: Structs

Edit `ex8_structs.c` using your editor of choice and fill in the blanks.

Compile and run the program and **check** that the output matches what you expect.

Optional: `typedef`s

Sometimes, you may see a `typedef` when declaring a struct:

```
typedef struct {
    int id;
} Student;
```

In these cases, you may use `Student` as the type instead of `struct Student`. We won't go into detail here, but feel free to check out this link if you're interested.

Exercise 9: Double Pointers

Edit `ex9_double_pointers.c` using your editor of choice and fill in the blanks.

Compile and run the program and **check** that the output matches what you expect.

Exercise 10: Putting It All Together

Here's one to help you in your interviews. In `ex10_ll_cycle.c`, complete the function `ll_has_cycle()` to implement the following algorithm for checking if a singly-linked list has a cycle.

1. Start with two pointers at the head of the list. One will be called fast_ptr and the other will be called slow_ptr.
2. Advance fast_ptr by two nodes. If this is not possible because of a null pointer, we have found the end of the list, and therefore the list is acyclic.
3. Advance slow_ptr by one node. (A null pointer check is unnecessary. Why?)
4. If the fast_ptr and slow_ptr ever point to the same node, the list is cyclic. Otherwise, go back to step 2.

If you want to see the definition of the `node` struct, open `ex10_ll_cycle.h` (FAQ: What is a header file?).

Action Item

Implement `ll_has_cycle()`. Once you've done so, you can execute the following commands to run the tests for your code. If you make **any** changes, make sure to run **ALL** of the following commands again, in order.

```
$ gcc -g -o ex10_test_ll_cycle ex10_test_ll_cycle.c ex10_ll_cycle.c
$ ./ex10_test_ll_cycle
```

Here's a Wikipedia article on the algorithm and why it works. Don't worry about it if you don't completely understand it. We won't test you on this.

Submission

Save, commit, and push your work, then submit to the **Lab 1** assignment on Gradescope.

FAQ

What is a header file?

Header files allow you to share functions and macros across different source files. For more info, see the GCC header docs .

What is a null terminator?

A null terminator is a character used to denote the end of a string in C. The null terminator is written as `'\0'`. The ASCII value of the null terminator is `0`. When you make a character array, you should terminate the array with a null terminator like this

```
char my_str[] = {'e', 'x', 'a', 'm', 'p', 'l', 'e', '\0'};
```

If you are using double quotes to create a string, the null terminator is implicitly added, so you should not add it yourself. For example:

```
char *my_str = "example";
```

What is an executable?

An executable is a file composed of binary that can be executed on your computer. Executables are created by compiling source code.

What is `strlen`?

See the man pages for a full description. Type the following into your terminal

```
$ man strlen
```

To exit the man pages, press `q`.

What is a macro?

A macro is a chunk of text that has a name. Whenever this name appears in code, the preprocessor replaces the name with the text. Macros are indicated with `#define` For example:

```
#define ARR_SIZE 1024
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))

int main() {
    int arr1[ARR_SIZE];
    int arr2[ARR_SIZE];
    int arr3[ARR_SIZE];

    for (int i = 0; i < ARR_SIZE; ++i) {
        arr3[i] = min(arr1[i], arr2[i]);
    }
}
```

In this code, the preprocessor will replace `ARR_SIZE` with `1024`, and it will replace

```
arr3[i] = min(arr1[i], arr2[i]);
```

with

```
arr3[i] = ((arr1[i]) < (arr2[i]) ? (arr1[i]) : (arr2[i]));
```

Macros can be much more complex than the example above. You can find more information [in the GCC docs]

What is a segfault?

A segfault occurs when you try to access a piece of memory that "does not belong to you." There are several things that can cause a segfault including

1. Accessing an array out of bounds. Note that accessing an array out of bounds will not always lead to a segfault. The index at which a segfault will occur is somewhat unpredictable.
2. Derefrencing a null pointer.

3. Accessing a pointer that has been `free`'d (`free` is not in the scope of this lab).
4. Attempting to write to read-only memory. For example, strings created with the following syntax are read only. This means that you cannot alter the value of the string after you have created it. In other words, it is immutable.

```
char *my_str = "Hello";
```

However, a string created using the following syntax is mutable.

```
char my_str[] = "hello";
```

Why is the first string immutable while the second string is mutable? The first string is stored in the data portion of memory which is read-only while the second string is stored on the stack.
