

Lab #1 - C Programming Basics  
CMPSC311 - Introduction to Systems Programming  
Spring 2022 - Prof. Sencun Zhu and Prof. Suman Saha  
**Due date: February 11, 2022 (11:59pm) EST**

In this lab assignment, you will write simple functions in C. The purpose of this lab assignment is to assess your basic programming skills. You should have no difficulty in completing this assignment. If you experience difficulty, you should take some time to brush up on programming.

---

Like all lab assignments in this class, you are prohibited from copying any content from the Internet or discussing, sharing ideas, code, configuration, text, or anything else or getting help from anyone in or outside of the class. Consulting online sources is acceptable, but under no circumstances should anything be copied. Failure to abide by this requirement will result in dismissal from the class as described in our course syllabus.

---

Below are the files in this assignment and their descriptions:

1. `student.h`: a header file with the declarations of the functions you will implement.
2. `student.c`: a source file in which you will implement the functions whose declarations appear in `student.h`. In other words, you will provide the definitions of the functions declared in `student.h`.
3. `ref.h`: a header file with the declarations of the functions identical to those defined in `student.h` except they are preprocessor macros. These macros are used for testing against which your implementations are compared.
4. `ref.o`: an object file. This is a binary file that contains compiled reference functions declared in `ref.h`.
5. `tester.o`: an object file that contains unit tests for the functions you implement. Each unit test passes input to your implementation of a function and compares the output with the reference implementation of the same function and compares the outputs. This file will compile into an executable, `tester`, which you will run to see if you pass the unit tests.
6. `Makefile`: instructions for compiling and building `tester` used by the `make` utility.

Your workflow will consist of (1) implementing functions by modifying `student.c`, (2) typing `make` to build the `tester`, and (3) running `tester` to see if you pass the unit tests, and repeating these three steps until you pass all the tests. Although you only need to edit `student.c` for successfully completing the assignment, you can modify any file you want if it helps you in some way. When testing your submission, *however, we will use the original forms of all files except `student.c`*. Do not forget to add *comments* to your code to explain your implementation choices.

Below are the functions you need to implement:

1. `smallest`: Takes an array of integers and the length of the array as input and returns the smallest integer in the array. You can assume that the input array contains at least one element.
2. `sum`: Takes an array of integers and the length of the array as input and returns the sum of the integers in the array.
3. `swap`: Takes pointers to two integers and swaps the values of integers.

4. `rotate`: Takes a pointer to three integers and rotates the values of integers. For example, if the inputs are pointers to integers `a`, `b`, and `c`, then after a call to `rotate`, `c` contains the value of `b`, `b` contains the value of `a`, and `a` contains the value of `c`.
5. `sort`: Takes a pointer to an array of integers and the length of the array as input and sorts the array in descending order (larger to smaller). That is, after a call to `sort` the contents of the array should be ordered in descending order. You can implement any sorting algorithm you want, but you have to implement the algorithm yourself—you cannot use sort functions from the standard C library. We recommend using something simple, such as Bubble sort or Selection sort.
6. `cube_primes`: Takes an array of integers and the length of the array as input and cubes every prime element of the array. For example, if the input array contains `[1, 7, -2]`, then after a call to `cube_primes`, the array will contain `[1, 343, -2]`.
7. `double_armstrongs`: Takes an array of integers and the length of the array as input and *double every positive element of the array that is an Armstrong number*. An Armstrong number (also called a Narcissistic number) is a number that is equal to the sum of its digits, each raised to the power of the number of digits. Read more about them [here](#). For example, 153 is an Armstrong number because it has 3 digits and  $1^3 + 5^3 + 3^3 = 153$ . Thus, if an input array contains `[2, -153, 153]`, then after a call to `double_armstrongs`, the array will contain `[4, -153, 306]`.
8. `negate_happy`: Takes an array of integers and the length of the array as input and *negates every happy number in the array*. A happy number is a number that eventually reaches 1 when you repeatedly replace it by the sum of the squares of its digits. For example, 19 is a happy number because  $1^2 + 9^2 = 82$ ,  $8^2 + 2^2 = 68$ ,  $6^2 + 8^2 = 100$ , and  $1^2 + 0^2 + 0^2 = 1$ . Thus, if an input array contains `[4, 19, 2]`, then after a call to `negate_happy`, the array will contain `[4, -19, -2]`.

## Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat `edu_assist_pro`

You are encouraged to write helper functions to simplify the implementation of the above functions. You should, however, **not use a library function** and implement all helper functions yourself. For example, if you need a function to raise number `a` to power `b`, you should implement that function yourself.

Assignment Project Exam Help

<https://eduassistpro.github.io/>

Add WeChat edu\_assist\_pro