School of Computer Science, McGill University

COMP-206 Introduction to Software Systems, Winter 2020

Mini Assignment 4: C Programming - Control Structures

Due Date Feb 20, 23:55

This is an individual assignment. You need to solve these questions on your own. Use the discussion forum on Piazza if you have any questions. You can also reach out to the course email address, utilize TA/Instructors office hours as necessary. Late penalty is -5% per day. Even if you are late only by a few minutes it will be rounded up to a day. Maximum of 2 late days are allowed.

You MUST use `mimi.cs.mcgill.ca` to create the solution to this assignment. You must not use your Mac command-line, Windows command-line, nor a Linux distro installed locally on your laptop. You can ssh or putty from your laptop to `mimi.cs.mcgill.ca`, or you can go to the third floor of Trottier and use any of those labs to ssh to `mimi` to complete this assignment. All of your solutions should be composed of commands that are executable in `mimi.cs.mcgill.ca`.

Questions in this exercise requires you to turn in two C programs. Instructors/TAs upon their discretion may ask you to demonstrate/explain your solution. No points are awarded for commands that do not execute at all or programs that do not compile in `mimi`. (Commands/programs that execute/compile, but provide incorrect behavior/output will be given partial marks.). All questions are graded proportionally. This means that if 40% of the question is correct, you will receive 40% of the grade.

Please read through the entire assignment before you start working on it. You can loose up to 3 points for not following the instructions.

Lab D will provide some background help for this mini assignment.

Unless otherwise stated, all the names of the scripts and programs that you write, commands, options, input arguments, etc. are implied to be case-sensitive.

Total Points: 20

Ex. 1 — A Simple Interger Calculator (9 Points)

In this exercise, you will create a C program to implement a simple integer calculator.
Your calculator should accept three arguments as input, `x`, `op`, `y`, where `x` and `y` are integers and `op` is one of the operators `+`, `-`, `*`, `/` to perform the corresponding arithmetic operations.

1. Use vi to create a C program source file `simplecalc.c`
2. **(1 Point)** The program should include a comment section at the beginning that describes its purpose (couple of lines), the author (your name), your department, a small "history" section indicating what changes you did on what date.

```
/*
 Program to implement a simple calculator
 ************************************************************
 * Author          Dept.              Date          Notes
 ************************************************************
 * Joseph D        Comp. Science. Feb 10 2020    Initial version.
 * Joseph D        Comp. Science. Feb 11 2020    Added error handling.
*/
```

The code should be properly indented for readability as well as contain any additional comments required to understand the program logic.

3. **(1 Point)** The source code should be compilable by (exactly) using the command `gcc -o simplecalc simplecalc.c`
4. **(3 Points)** Your program should accept the three arguments mentioned above as its input. If the user does not pass this argument, you should display an error message and terminate with error code `1`.

```
$ ./simplecalc
Error: usage is simplecalc <x> <op> <y>
$ echo $?
1
```

5. **(4 Points)** Your program performs simple arithmetic operations on the input as indicated by the operator and displays the output. You should use a `switch` statement to decide which operator to execute. You will loose 2 points if you do not use the `switch`. If the user enters one of the operators not listed above, print an error message and terminate with error code `2`. Successfull execution of the program should result in a return code of 0 from the program.

```
$ ./simplecalc 5 + 3
8
$ echo $?
0
$ ./simplecalc 5 - 3
2
$ ./simplecalc 5 * 3
15
$ ./simplecalc 5 / 3
1
$ ./simplecalc 5 @ 3
@ not a valid operator
$ echo $?
2
```

Ex. 2 — A C Program to implement Caesar's Cipher(11 Points)

In this exercise, you will create a C program to implement a basic version of Caesar's cipher.
`https://en.wikipedia.org/wiki/Caesar_cipher`
The object of Caesar's cipher is to replace each letter in the alphabet with another letter that is 'X' offset away from it to scramble the message.
For example consider the following sample output

```
$ ./caesarcipher 3
abcdefghijklmnopqrstuvwxyz
defghijklmnopqrstuvwxyzabc
```

Here we asked the program to set an offset of 3, and typed the following line and pressed enter

```
$ ./caesarcipher 3
abcdefghijklmnopqrstuvwxyz
```

At which point, it changed each letter in the input line with the letter that is 3 letters to its right.

```
defghijklmnopqrstuvwxyzabc
```

As a result, `a` got replaced with `d` and so forth. Towards the end of the alphabets, you have to "wrap around". In the above example, There is no alphabet 3 letters to the right of `x`, so it wrapped around to `a` and so forth.
You can "decrypt" the message by running the program with the proper offset in the opposite direction, as shown below.

```
$ ./caesarcipher -3
defghijklmnopqrstuvwxyzabc
abcdefghijklmnopqrstuvwxyz
```

Here, we started with program with an offset -3 which asked the program to shift the letters to the left and the entered the original encrypted line.

```
$ ./caesarcipher -3
defghijklmnopqrstuvwxyzabc
```

To which it responded with the decrypted message.

```
abcdefghijklmnopqrstuvwxyz
```

1. Use vi to create a C program source file `caesarcipher.c`
2. **(1 Point)** The program should include a comment section at the beginning that describes its purpose (couple of lines), the author (your name), your department, a small "history" section indicating what changes you did on what date. The code should be properly indented for readability as well as contain any additional comments required to understand the program logic.
3. **(1 Point)** The source code should be compilable by (exactly) using the command `gcc -o caesarcipher caesarcipher.c` .
4. **(2 Points)** Your program should accept an integer argument as its option for the user to indicate the offset. If the user does not pass this argument, you should display an error message and terminate with error code `1`.

```
$ ./caesarcipher
Error: usage is caesarcipher <offset>
$ echo $?
1
```

5. **(4 Points)** Your program should use the offset to shift the letters typed in a line either to the right.

```
$ ./caesarcipher 5
this is a secret message
ymnx nx f xjhwjy rjxxflj
```

or to the left.

```
$ ./caesarcipher -5
this is a secret message
ocdn dn v nzxmzo hznnvbz
```

6. **(2 Points)** The program should be able to read multiple lines, encrypting and displaying one line at a time, till it runs out of input/reaches `EOF` (can be simulated by pressing CTRL+D at the keyboard input in an empty line). If you are using the tester script, you shouldn't have to do CTRL+D (if you do, there is an issue with your implementation). The program should terminate with return code `0` for any sucessfull execution.

```
$ ./caesarcipher 5
this is a secret message
ymnx nx f xjhwjy rjxxflj
delete after reading
ijqjyj fkyjw wjfinsl
$ echo $?
0
```

7. **(1 Point)** The program should only encrypt lowercase english alphabets (a-z) and should merely print any other letters, numbers, white space characters, etc., as is.

```
$ ./caesarcipher 6
Hello there! How is COMP206?
Hkrru znkxk! Huc oy COMP206?
```

WHAT TO HAND IN

Turn in the C program source codes `simplecalc.c` and `caesarcipher.c`, named properly. You do not have to zip all of the files together. The files must be uploaded to mycourses. DO NOT turn in the executables `simplecalc` or `caesarcipher`. TAs will compile your C program on their own as indicated in the problem descriptions above.

MISC. INFORMATION

There is a tester script `mini4tester.sh` that is provided with the assignment that you can use to test how your programs are behaving. TAs will be using the exact same tester script to grade your assignment.

```
$ ./mini4tester.sh
```

However, it is recommended that when you start writing your program, test it yourself first with the above examples. Once you are fairly confident that your program is working, you can test it using the tester script.

You can compare the output produced by running the mini tester on your C programs to that produced by the mini tester on the solution programs which is given in `mini4tester.out.txt`.

**** Important **** If your program "hangs" / is stuck while executing it through the tester script and requires TAs to interrupt it, you will loose 4 points per program!.

FOOD FOR THOUGHT!

The following discussion is meant to encourage you to search independently for creative and optimal ways to perform rudimentary tasks with less effort and/or make your program robust and sophisticated. It does not impact the points that you can achieve in the above questions.

- Will your cipher program work for the following offsets? (Assuming that you keep continuing to wrap around multiple times).

  ```
  $ ./caesarcipher 306
  ```

  and

  ```
  $ ./caesarcipher -306
  ```

  If not, can you think of a very simple way to address it?

- Here is a fun activity you can have with your friends using the cipher program that you just developed. You should have the cipher program running, with a secret offset key which you do not tell anyone. Now encrypt a message using it and use the `write` Unix command to send it to a friend who is also logged into the same Unix host. Have your friend also do the same back to you (using their cipher program and their own secret offset key). Now compete in who can decrypt the other's message first.