ECS 34: Programming Assignment #1

Instructor: Aaron Kaloti

Fall 2020

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Added autograder details. Clarified that arithmetic overflow will not be an issue unless you go out of your way to use an unusual type.
- v.3: Added a note of caution about copying a single quotation mark (') from a PDF generated from LaTeX to part #4.
- v.4: Pushed deadline back one day.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Wednesday, October 14. Gradescope will say 12:30 AM on Thursday, October 15, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

You should use the `-Wall` and `-Werror` flags when compiling. The autograder will use these flags when it compiles your program.

3 Grading Breakdown

Each of the four parts will take up 25% of this assignment's worth. Your grade on each part is completely determined by the autograder.

4 Submitting on Gradescope

You should only submit your C files with the names indicated below. You may be penalized for submitting additional files. You have infinite submissions until the deadline.

During the 10/02 lecture, I talked about how to change the active submission, just in case that is something that you find yourself needing to do.

4.1 Regarding Autograder

There are in total eight test cases (four for each part). The last case for part #1 and the last two cases for part #2 are hidden; you cannot see if you got them correct or not until after the deadline. Your score will always appear as a - until the hidden test cases are revealed after the deadline; Gradescope does not let me change this. You can still of course see if you got the visible cases correct or not and count up the points.

Your output must match mine *exactly*, including whitespace. This is so that I do not have to write an autograder that deals with every possible way to display the output of these programs. As a minor convenience, the autograder does ignore trailing whitespace, however.

You will be told if you got the visible test cases correct or not a little bit after you submit to Gradescope (should not take more than 30 seconds, unless something is wrong with your programs). For each visible test case in this particular programming assignment (might not be true in future programming assignments), you will be told where the output differs. Below is an example. The first line of the output is probably useless; you should try to understand the rest of the output. Any line that begins with a `-` is a line that your program output that it was not supposed to; any line with a `+` is a line that your program *did not output* that it was supposed to. In computer science, it is common for software that finds the differences between two bodies of text (i.e. that "diffs" two bodies of text) to report it in this line-by-line fashion, so you should become accustomed to it. The line that starts with a question mark `?` later has a `-` in it (as you can see in the third line of the output below) that shows where the two lines begin to mismatch.

Test part01_01 (0.0/5.0)

```
Test Failed: 'Ente[15 chars]: Enter second integer: Enter third intteger: The sum is: 25\n' !=
- Enter first integer: Enter second integer: Enter third intteger: The sum is: 25
?                                                    -
+ Enter first integer: Enter second integer: Enter third integer: The sum is: 25
```

If either of your programs failed to compile, you will see a message like the below. Make sure to read the suggestions; students sometimes ignore those.

Test part01_01 (0.0/5.0)

```
Test Failed: 'COMPILATION ERROR' != '...'
- COMPILATION ERROR
+ ...
 : Failed to compile your program for this part, due to a compilation error.
Below are some suggestions:
1. Make sure that your program compiles on the CSIF with the -Wall and -Werror flags.
2. Make sure that you did not give the file the wrong name or submit the wrong file(s).
```

When looking at the output mismatches for programs that never outline newline characters, the autograder will display both the expected output and (likely) your output on one line each. Once you read about output redirection for Linux HW #2, you may be able to understand why. Below is an example. You are still shown where the output begins to differ.

Test part03_01 (0.0/4.0)

```
Test Failed: 'Ente[70 chars]otherr integer: Enter anotherr integer: Enter [15 chars]r:\n' != 'E
- Enter first integer: Enter second integer: Enter another integer: Enter anotherr integer: Ent
?                                                                                  -
+ Enter first integer: Enter second integer: Enter another integer: Enter another integer: Ente
```

4.1.1 Test Cases' Inputs

Below are the inputs of the test cases. The hidden test cases' inputs won't be revealed until after the deadline.

Part #1, Case #1:

```
5
17
3
```

Part #1, Case #2:

```
17
-2
0
```

Part #1, Case #3:

```
-3
-2
-17
```

Part #1, Case #4:

Part #2, Case #1:

```
a
b
c
```

Part #2, Case #2:

```
  x
 y
z
```

Part #2, Case #3:

Part #2, Case #4:

Part #3, Case #1:

```
5
-7
3
88
1000
5000
```

Part #3, Case #2:

```
-10
2
-7
-11
83
-83
-100
-50
```

Part #3, Case #3:

```
10
2
3
```

Part #3, Case #4:

Part #3, Case #5:

Part #4, Case #1:

```
q
q
q
q
q
q
q
q
q
q
q
q
q
q
q
q
q
q
q
r
r
r
s
```

Part #4, Case #2:

```
q
d
d
q
d
x
l
l
n
p
q
q
p
q
q
r
q
q
q
q
q
q
q
l
l
l
k
s
```

Part #4, Case #3:

```
p
p
n
l
l
r
r
p
n
q
k
l
d
q
q
q
q
```

```
q
q
q
q
q
s
```

Part #4, Case #4:

Part #4, Case #5:

5 Programming Problems

Except where otherwise specified, you may assume that the user's input is always valid, e.g. the user will not enter the wrong type of input, you don't need to worry about arithmetic overflow (so long as you don't do something strange such as using `char` variables to store the integers in part #1), etc. The output of your program must match mine; **this includes your input prompts**.

5.1 Part #1

File: `add_three.c`

Write a C program that prompts the user to enter three integers and tells the user what the sum of those three integers are.

Below are examples of how your program should behave on a Linux command line.

```
$ gcc -Wall -Werror add_three.c -o add_three
$ ./add_three
Enter first integer: 5
Enter second integer: 17
Enter third integer: 3
The sum is: 25
$ ./add_three
Enter first integer: -2
Enter second integer: -5
Enter third integer: 14
The sum is: 7
$ ./add_three
Enter first integer: 0
Enter second integer: 5
Enter third integer: 10
The sum is: 15
$
```

5.2 Part #2

File: `get_three.c`

Write a C program that prompts the user to enter three characters, one at a time. When you ask the user for the second and third characters, remind the user what character they last entered. The program should then print the three characters backwards. Any whitespace entered by the user should be completely ignored.

Here are examples of how your program should behave on a Linux command line.

```
$ gcc -Wall -Werror get_three.c -o get_three
$ ./get_three
Enter first character: a
You just entered a. Enter second character: b
You just entered b. Enter third character: c
Backwards, the three characters are cba.
$ ./get_three
Enter first character: x
You just entered x. Enter second character: y
You just entered y. Enter third character: y
Backwards, the three characters are yyx.
```

```
$ ./get_three
Enter first character: bla
You just entered b. Enter second character: You just entered l. Enter third character: Backwards, the
    three characters are alb.
$ ./get_three
Enter first character:        d
You just entered d. Enter second character:   e
You just entered e. Enter third character:               f
Backwards, the three characters are fed.
$
```

5.3 Part #3

Write a C program that keeps prompting the user to enter an integer. The program should continue to do this until the user enters an integer that is not greater than the product of the last two integers that they entered. The user will have to always enter at least three integers.

Here are examples of how your program should behave on a Linux command line.

File: `grow_prod.c`

```
$ gcc -Wall -Werror grow_prod.c -o grow_prod
$ ./grow_prod
Enter first integer: 5
Enter second integer: 2
Enter another integer: 9
$ ./grow_prod
Enter first integer: 5
Enter second integer: 2
Enter another integer: 11
Enter another integer: 25
Enter another integer: 300
Enter another integer: 4000
$ ./grow_prod
Enter first integer: -10
Enter second integer: 3
Enter another integer: -7
Enter another integer: 6
Enter another integer: 0
Enter another integer: 1
Enter another integer: 1
Enter another integer: 1
$ ./grow_prod
Enter first integer: 6
Enter second integer: 5
Enter another integer: 40
Enter another integer: 1000
Enter another integer: 3000
$ ./grow_prod
Enter first integer: -12
Enter second integer: 3
Enter another integer: -30
Enter another integer: 2
Enter another integer: 15
Enter another integer: -20
$
```

5.4 Part #4

File: `snack_machine.c`

In this part, you will write a C program that simulates a vending machine that has snacks. Your program should keep prompting the user to enter a character until the user enters `s`, at which point the user should be told to have a good day and the program should end. Here is how the program should react for other characters:

- `q`: Simulates the user entering a quarter, which is worth 25 cents.
- `d`: Simulates the user entering a dime, which is worth 10 cents.
- `n`: Simulates the user entering a nickel, which is worth 5 cents.
- `p`: Simulates the user entering a penny, which is worth 1 cent.
- `l`: The user is asking for Lay's chips, which costs $2.00. If they have enough money, then you should tell them that; otherwise, you should tell them that they don't have enough money.

- `r`: Same as for Lay's chips, except that the user is asking for Ruffles chips instead, which costs $2.25.
- `k`: Same as for Lay's chips, except that the user is asking for a Kit Kat candy bar instead, which costs $1.50.
- Any other character: Tell the user that their input was invalid (see the beginning of the examples below).

I can't remember if I only mentioned this in ECS 36A and not in ECS 34, but in C, there is a distinction between single quotes and double quotes: the former is used to denote a `char` value (e.g. `'q'`), whereas the latter is used to denote a string (e.g. `"Hello"`). You don't need strings for this part.

Here are examples of how your program should behave on a Linux command line. **As a reminder, your output must match mine exactly.** If you copy specific parts of the output below (to form your print statements), then be careful about copying single quotation marks (’), i.e. from "`Enjoy your Lay's chips.`". When you copy such marks from a PDF that was generated from LaTeX, it could be the case that the quotation mark is distorted once you paste it; it will *look* like a normal quotation mark, but C code with such a distorted quotation mark will fail to compile on the autograder.

```
$ gcc -Wall -Werror snack_machine.c -o snack_machine
$ ./snack_machine
Machine currently has $0.00 in it.
Enter: f
Invalid input.
Machine currently has $0.00 in it.
Enter: d
Machine currently has $0.10 in it.
Enter: q
Machine currently has $0.35 in it.
Enter: p
Machine currently has $0.36 in it.
Enter: n
Machine currently has $0.41 in it.
Enter: k
You need $1.09 more to afford a Kit Kat.
Machine currently has $0.41 in it.
Enter: q
Machine currently has $0.66 in it.
Enter: q
Machine currently has $0.91 in it.
Enter: q
Machine currently has $1.16 in it.
Enter: q
Machine currently has $1.41 in it.
Enter: q
Machine currently has $1.66 in it.
Enter: k
Enjoy your Kit Kat candy.
Machine currently has $0.16 in it.
Enter: k
You need $1.34 more to afford a Kit Kat.
Machine currently has $0.16 in it.
Enter: q
Machine currently has $0.41 in it.
Enter: s
Have a nice day.
$ ./snack_machine
Machine currently has $0.00 in it.
Enter: q
Machine currently has $0.25 in it.
Enter: q
Machine currently has $0.50 in it.
Enter: q
Machine currently has $0.75 in it.
Enter: q
Machine currently has $1.00 in it.
Enter: q
Machine currently has $1.25 in it.
Enter: k
You need $0.25 more to afford a Kit Kat.
Machine currently has $1.25 in it.
Enter: l
You need $0.75 more to afford Lay's chips.
Machine currently has $1.25 in it.
Enter: r
You need $1.00 more to afford Ruffles chips.
Machine currently has $1.25 in it.
Enter: q
```

```
Machine currently has $1.50 in it.
Enter: q
Machine currently has $1.75 in it.
Enter: q
Machine currently has $2.00 in it.
Enter: r
You need $0.25 more to afford Ruffles chips.
Machine currently has $2.00 in it.
Enter: l
Enjoy your Lay's chips.
Machine currently has $0.00 in it.
Enter: k
You need $1.50 more to afford a Kit Kat.
Machine currently has $0.00 in it.
Enter: p
Machine currently has $0.01 in it.
Enter: r
You need $2.24 more to afford Ruffles chips.
Machine currently has $0.01 in it.
Enter: q
Machine currently has $0.26 in it.
Enter: s
Have a nice day.
$ ./snack_machine
Machine currently has $0.00 in it.
Enter: q
Machine currently has $0.25 in it.
Enter: q
Machine currently has $0.50 in it.
Enter: q
Machine currently has $0.75 in it.
Enter: q
Machine currently has $1.00 in it.
Enter: q
Machine currently has $1.25 in it.
Enter: q
Machine currently has $1.50 in it.
Enter: q
Machine currently has $1.75 in it.
Enter: q
Machine currently has $2.00 in it.
Enter: q
Machine currently has $2.25 in it.
Enter: r
Enjoy your Ruffles chips.
Machine currently has $0.00 in it.
Enter: s
Have a nice day.
$
```

UCDAVIS
COMPUTER SCIENCE