CMPUT275—Assignment 2 (Winter 2024)

X. Li R. Hackman

Due Date: Friday February 16th, 8:00PM

Per course policy you are allowed to engage in *reasonable* collaboration with your classmates. You must include in the comments of your assignment solutions a list of any students you collaborated with on that particular question.

In this assignment and all following assignments you should test against the sample executables. The sample executables try to help you out and print error messages when receiving invalid input (though they do not catch *all* invalid input). You do not have to replicate any error messages printed, unless the assignment specification specifically asks for error messages. These messages are only in the sample executable to help you catch when you write an invalid test case.

Memory Management: In order to complete some of these questions you will be required to use dynamic memory allocation. Your programs must not leak any memory, if you leak memory on a test case then you are considerd to have failed that test case. You can test your program for memory leaks by using the tool `valgrind`.

Memory Requirements: In addition to not leaking memory your programs must not use at any one time more than double of the maximum amount of memory they require. That is if implementing a dynamic array you may should use the doubling strategy. If you simply allocate a very large array hoping input sizes will never exceed that then you will not receive marks for that question. For initializing dynamic arrays you may intialize them to have a capacity of 4.

Allowed libraries: `stdlib.h`, `stdio.h`, and `string.h`. No other libraries are allowed.

1. **Wordl**

   In this question you will develop a game called *wordl*, wordl is a word game much like the game wordle. In the game of wordl there is a secret code word that the user is trying to guess, and whenever a user makes a guess the following happens:

   - The users guess is printed out colorized with the following rules:
     - If the letter in the guess is not in the code at all then it is printed out as white.
     - If the letter in the guess is the same letter at that position as the code then the letter is printed out as green.
     - If the letter in the guess occurs in the code, but is in the wrong spot then you print it out as yellow *if* the corresponding letter in the code is not matched as green by the previous rule and you have not printed out a number of that character in yellow equal to the number of that character in the code minus the number of green of that character.

   For example, if the code was `verge` and you guessed `bevel` then the printed out response would be:

   

However, if the code was `bevel` an you guessed `eenie` then the printed out response would be:

With the important distinction that the last letter `e` was coloured white, despite the fact that the letter `e` appears in the code. This is because there was already a yellow `e` previously printed and there was a green `e` printed, so there were already a coloured `e` to account for both letters `e` in the code.

Your program expects one command line argument which is the code word that must be guessed. Your program then should repeat the following process:

(a) Print the message `"Enter guess:  "` and read a string from standard input, that string will be the users guess.

(b) Print out the colourized version of the user's guess followed by a newline.

(c) If the user did not guess the code word correctly, and has not already made 6 guesses, then begin these steps again.

(d) If the user did not guess the code correctly, and has already made 6 guesses then print out the message `"Failed to guess the word: <word>"` followed by a newline, where `<word>` is replaced with the secret code word. Then your program terminates.

(e) If the user does guess the code correctly, then print the message `"Finished in <n> guesses"` followed by a newline, where `<n>` is replaced with the number of guesses the user made.

You may assume the user always gives you a guess exactly equal to the length of the code, and you may assume the code is never more than 12 characters long.

In order to print out colours you will have to print special colour characters, and these are characters that show up in your output, so you have to be careful about when you print them. You should only print out a colour character whne you need to switch colours for the immediate next thing you'd like to print. In order to help with this we have given you some starter code that uses some global variables as well as a function `setColour`. You should use the function `setColour` and provide as the argument one of the global variables `YELLOW`, `GREEN`, or `WHITE` for those colours. The function makes sure not to print out the colour character if that colour is already set, which will help in ensuring you do not print out erroneous colour characters, but does not guarantee you are printing them exactly only when you need to.

Deliverables: For this question include in your final submission zip your c source code file named `wordl.c`

2. **Reverse Polish Notation**

In this question you will be writing a simple interpreter for arithmetic expressions written in *reverse polish notation*. Reverse polish notation, or postfix notation, is a notation for expressions where operators follow their operands. This is opposed to the usual *infix* notation that we are used to where operators are placed inbetween their operands. That is the infix expression `5 + 3 - 10` would be written in reverse polish notation as `5 3 + 10 -`.

For this program you need to be able to interpret any **valid** reverse polish notation expression that includes any valid combination of integers and operators `p, s, *, /`, where `p` means the addition operator, `s` means the subtraction operator, `*` means the multiplication operator and `/` means the integer division operator.

You may assume no operations will result in integer overflow. That is if any calculation would result in a value outside the bounds of `[INT_MIN, INT_MAX]` then that input is invalid. You may also assume that any expression you are given is valid, which is that all operators have the appropriate amount of operands and all operands are consumed to result in one final value.

Note: There may be any amount of whitespace inbetween each operator/operand. You must be able to handle this arbitrary amount of whitespace.

Hint 1: It will help you to create a *stack*. A stack is a simple data structure in which you can only add and remove elements to/from the back of it. Due to this behaviour a stack is called a *last in, first out (LIFO)* data structure. What should you do to your stack when you see an integer in the input stream? What should you do to your stack when see an operator in the input stream?

Deliverables For this question include in your final submission zip your c source code file named `rpn.c`

3. **Integer Sets**

In this question you will be writing a program that allows you to create and manipulate two arbitarary sets of integers.

A *set* is a collection of objects with no duplicates, so your data structure should have no duplicates integers in it. Your program will need to read input that specifies several commands the user would like to do to interact with the two sets your program manages (which the user will refer to as `x` and `y`).

Your program should read commands from the user executing them until receiving the commmand `q` upon which time your program terminates. In all of the following command `<targ>` must be replaced with either `x` or `y` to refer to that particular set, and `<int>` must be replaced with an integer. The commands your program must handle are:

- `a <targ> <int>` — the command for adding an integer to a set. When you receive this command you should add the specified integer to the specified set. If the integer already exists in the set then you should not add it, as sets should not contain duplicates.
- `r <targ> <int>` — the command for removing an integer from a set. When you receive this command you should remove the specified integer from the specified set. If the integer does not already exist in the set then no change should occur.
- `p <targ>` — the command for printing out a set. This should print out the elements of the specified set in increasing order with one space between each element and ending in a newline. If there are no elements in the set then print nothing.
- `u` — this command for *set union*. When receiving this command you should calculate and print out the *union* of your two sets. The union of two sets s1 and s2 is the set which contains every element that is in either s1 or s2. When printing out the union you should print out the elements in increasing order with one space between each element and ending in a newline. If there are no elements in the union then print nothing.
- `i` — the command for *set intersection*. When receinv this command you should calculate and print out the *intersection* of your two sets. The intersect of two sets s1 and s2 is the set which contains only elements that occur in *both* s1 and s2. When printing out the intersection you should print out the elements in increasing order with one space between each element and ending in a newline. If there are no elements in the intersection then print nothing.
- `q` — the command for quitting your program, when received your program should terminate.

Note: For each command there may be any amount of whitespace inbetween the components of the commands, or inbetween separate comamnds. You must be able to handle this arbitrary amount of whitespace.

Deliverables For this question include in your final submission zip your c source code file named `int_set.c`

How to submit: Create a zip file `a2.zip`, make sure that zip file contains your C source code files `wordl.c`, `rpn.c`, and `int_set.c`. Assuming all three of these files are in your current working directory you can create your zip file with the command

```
$ zip a2.zip wordl.c rpn.c int_set.c
```

Upload your file `a2.zip` to the a1 submission link on eClass.