CMPUT275—Assignment 3 (Winter 2025)

R. Hackman

Due Date: Friday March 14th, 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 (and 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.

Compilation Flags: each of your programs should be compiled with the following command:

```
gcc -Wall -Wvla -Werror
```

These are the flags we'll compile your program with, and should they result in a compilation error then your code will not be compiled and ran for testing.

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

1. **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`

2. **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`

3. **Image Translation**

In this question you will be developing a program that can apply some translations to image files. This question will operate on text that constitutes a "Plain or Raw PPM" image format. This image format is a plaintext format that is readable by a human. Each Plain PPM file begins with a header that gives you some information about the file, and then followed by the "payload" which is the image itself. The format of a Plain PPM image file is as follows:

- The first line will have the string `P3` indicating that this is a P3 PPM file.
- The second line contains two integers separated by whitespace, the first is the width of the image (in pixels) and the second is the height of the image (in pixels).
- The third line indicates the maximal value for any colour component of an individual pixel, for this assignment we will assume this is always 255.
- After the third line is the payload. There will be one line for each pixel in the height of your image.
- In each line in the payload there will be sets of three integers that represent the colour of an individual pixel. Each of these three integers in order will represent the red (R), green (G), and blue (B) value of that pixel. There will be a number of pixels described in each line equal to the width of your image.

For example, consider the following PPM file which has one row of blue pixels, below that a row of red pixels, and below that row of green pixels.

```
P3
4 3
255
0 0 255 0 0 255 0 0 255 0 0 255
255 0 0 255 0 0 255 0 0 255 0 0
0 255 0 0 255 0 0 255 0 0 255 0
```

Pixels are stored in order from the top left of the image to the bottom right. So the first pixel described in your PPM file is the leftmost pixel of the top row, the second pixel in that same line is the next pixel in the top row, so on and so forth. Each line represents one rows of pixels in your image.

You must write a program `transformer.c` that reads from standard input an image in the PPM format. Your program must also accept optionally the command line arguments `-f` and `-s`. Depending on which (if any) command line arguments your program received, your program will then apply transformations to the image it read in. Your program must then print to standard output the transformed image (or the original image, if no transformations were requested). The two command line arguments have the following behaviour:

- `-f` indicates that the user of your program wants to flip the image. Flipping the image means that in each row the pixel at index 0 becomes the pixel at index n-1 (and vice versa), the pixel at index 1 becomes the pixel at index n-2 (and vice versa), etc.
- `-s` indicates the user of your program wants to apply the sepia filter to the image. For our purposes the sepia filter will exactly be determined by applying the following transformation to every pixels R, G, and B values:

$$newR = min(255, \lfloor R * 0.393 + G * 0.769 + B * 0.189 \rfloor)$$
$$newG = min(255, \lfloor R * 0.349 + G * 0.686 + B * 0.168 \rfloor)$$
$$newB = min(255, \lfloor R * 0.272 + G * 0.534 + B * 0.131 \rfloor)$$

Hint 1: It will be very useful to use a `struct` to represent a pixel! A simple struct containing 3 `ints` will work very well!

Hint 2: Once you know how you are going to store your image, this program can be broken down very nicely into a few functions. It is worth writing one function each to do the following operations: read the image in, apply a sepia filter to an image, flip an image, and print an image out.

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

How to submit: Create a zip file `a2.zip`, make sure that zip file contains your C source code files `int_set.c`, `rpn.c`, and `transformer.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 rpn.c int_set.c transformer.c
```

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