

CMPUT201—Assignment 3 (Winter 2025)

R. Hackman

Due Date: Friday March 28th, 8:00PM

Per course policy you are allowed to engage in 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. However, for purposes of your learning it is highly recommended that you be able to complete each of these questions on your own.

In each assignment you should test against the sample executables. **Important:** provided sample test cases are just that - *samples*. They are **NOT** sufficient to test your program! You must be writing additional test cases if you want to ensure your program works correctly.

Efficiency Requirements: While there are not strict efficiency guidelines, your code should strive to not be incredibly inefficient. If your solution is not comparable for large test cases in how long it takes to run compared to the sample executable then you may lose marks due to your program hitting the time limit. As such, it is good to compare your program to the sample executable not only for output but also timing purposes.

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 considered 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 initialize them to have a capacity of 4. Failure to use the doubling strategy will likely result in your program being too slow.

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 for this assignment: `stdio.h`, `stdlib.h`, `string.h`, `assert.h`. No other libraries are allowed.

1. Integer Sets

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

A *set* is a collection of objects with no duplicates, so your data structure should have no duplicate 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 command **q** upon which time your program terminates. In all of the following command **<target>** 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 <target> <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 <target> <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 <target>** — 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 receiving 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 commands. You must be able to handle this arbitrary amount of whitespace.

For examples, run the provided test cases with the sample executable.

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

2. Linear Sequences

In this question you will be writing a C program `sequences.c` that will be able to compose arithmetic operators to form a linear sequence, and be asked to print out the next item in the sequence at any time. Your program will read in pairs of operators and operands which will make up the series of arithmetic operations to apply to each integer in the sequence to form the next integer in the sequence. Your program will have the following specifications for its arguments and its input.

- Your program will expect one command line argument which is an integer, representing the initial value of the sequence.
- Your program will then read input from standard input until reaching EOF, correctly formed input will always be one of the following
 - One of the operator strings `add`, `sub`, `mul`, or `div` followed by one integer.
 - Or, the string `n`
- When receiving one of the operator strings, your program will add that operator to the series of operators to apply to each number in the sequence to produce the next. The integer provided after each operator string is the right hand operand to be used with that operator when applying it, the left hand operand of each operator will always be the current integer in your series, or the result of applying a previous operation. When calculating the next integer in the series the operators must always be applied in the order they were received in input!
- When receiving the string `n` your program will calculate the next integer in the series, as currently defined by your operators, and print it out followed by a newline.

So, for example, if your command line argument was `3` and your program receives the following input

```
add 1 mul 2 n n
```

then your program would print out

```
8
18
```

because $(3 + 1) * 2 = 8$, and $(8 + 1) * 2 = 18$.

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

$$\begin{aligned}\text{newR} &= \min(255, \lfloor R * 0.393 + G * 0.769 + B * 0.189 \rfloor) \\ \text{newG} &= \min(255, \lfloor R * 0.349 + G * 0.686 + B * 0.168 \rfloor) \\ \text{newB} &= \min(255, \lfloor R * 0.272 + G * 0.534 + B * 0.131 \rfloor)\end{aligned}$$

Hint: 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.

Extra exercise: This is not mandatory for the assignment, this is only a recommended practice problem. Try adding a command line option of `-r` to your program that also rotates the image 90 degrees. This will be harder than options you must implement for the assignment! It is safest to try this on a new copy of your assignment once you have already completed the original assignment, so that you do not accidentally break your working solution.

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 `a3.zip`, make sure that zip file contains your C source code files `int_set.c`, `sequences.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 a3.zip int_set.c sequences.c transformer.c
```

Upload your file `a3.zip` to the a3 submission link on eClass.