CS246—Assignment 2 (Fall 2015)

Due Date 1: Tuesday, October 13th, 04:55pm
Due Date 2: Monday, October 19th, 04:55pm

Questions 1, 2a, 3, 4a are due on Due Date 1; Questions 2b and 4b are due on Due Date 2. You should NOT wait till Due Date 2 to begin working on the deliverables for Due Date 2.

Note: On this and subsequent assignments, you will be required to take responsibility for your own testing. As part of that requirement, this assignment is designed to get you into the habit of thinking about testing *before* you start writing your program. Test suites will be in a format compatible with A1Q5 and A1Q6 (as applicable). So if you did a good job writing your `runSuite` script, it will serve you well on this assignment.

Be sure to do a good job on your test suites, as they will be your primary tool for verifying the correctness of your submission. For this reason, **C++ code due on Due Date 2 will only get one release token per twelve hours**. We want you to rely on your own pre-written test suite, not Marmoset, to verify correctness.

Note: You must use the C++ I/O streaming and memory management facilities on this assignment. Marmoset will be programmed to **reject** submissions that use C-style I/O or memory management.

Note: Further to the previous note, your solutions may only `#include` the headers `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, and `<string>`. No other standard headers are allowed. Marmoset will check for this.

Note: There will be a handmarking component in this assignment, whose purpose is to ensure that you are following an appropriate standard of documentation and style, and to verify any assignment requirements not directly checked by Marmoset. Please code to a standard that you would expect from someone else if you had to maintain their code. Documentation guidelines have been uploaded to the repository (DocumentationGuidelines.pdf).

1. **Note: there is no coding associated with this problem.** We would like to write a program called `change`, which accepts an integer (representing an amount of money, in cents) on standard input. The program then gives the minimal combination of Canadian coins (toonies, loonies, quarters, dimes, nickels, and pennies) whose total value equals the given amount on standard output. For example, if the input is

   ```
   89
   ```

   then the output is

   ```
   3 quarters
   1 dime
   4 pennies
   ```

Note that if only a single coin of a denomination is used, the singular form is used (e.g. dime vs dimes).

Your task is not to write this program, but to design a test suite for this program. Your test suite must be such that a correct implementation of this program passes all of your tests, but a buggy implementation will fail at least one of your tests. Marmoset will use a correct implementation and several buggy implementations to evaluate your test suite in the manner just described.

Your test suite should take the form described in A1P5: each test should provide its input in the file `testname.in`, and its expected output in the file `testname.out`. The collection of all `testnames` should be contained in the file `suiteq1.txt`.

Zip up all of the files that make up your test suite into the file `a2q1.zip`, and submit to Marmoset.

2. In this question we will implement an election that uses `cumulative voting`. In this voting scheme, a voter has X number of votes that they can choose to distribute among the candidates which are numbered from 1 to n, where n is at most 10 (the edge case of 0 candidates is possible and is considered valid input). The value for X may be provided as an optional positive command line argument. If a value of X is not provided as an argument, the default value of n is used. Input begins with the names of candidates (one full name per line). The first name is considered as candidate 1, and second as candidate 2, and so on. A candidate's name will never contain a numeral and consists of at least 1 and at most 15 characters (including any spaces). The list of candidates is followed by some number of lines where each line indicates one voter's distribution of votes, which we call a `ballot`. For a ballot the i^{th} column indicates the number of votes allocated to the i^{th} candidate. A ballot is considered invalid (*spoilt*) if it does not consist of n columns or the sum of the votes in the ballot exceeds X. In addition, the votes allocated to a specific candidate within a ballot are always non-negative. The number of voters is unknown beforehand, but is, of course, non-negative. Votes are terminated by end-of-file.

   As an example, given the following data:

```
Victor Taylor
Denise Duncan
Kamal Ramdhan
Michael Ali
Anisa Sawh
Carol Khan
Gary Owen
3 0 1 0 0 1 2
1 1 1 1 0 1 2
1 1 1 1 1 1 1
2 1 3 1
7 0 0 0 0 0 0
1 1 1 1 1 1 2
```

   your program should produce the following output:

```
Number of voters: 6
Number of valid ballots: 4
Number of spoilt ballots: 2

Candidate      Score

Victor Taylor    12
Denise Duncan     2
Kamal Ramdhan     3
Michael Ali       2
Anisa Sawh        1
Carol Khan        3
Gary Owen         5
```

Use the following format for the data in the two columns:

- Candidate: left-justified, 15 characters wide
- Score: right-justified, 3 characters wide

The program will be run using a test harness equivalent to A1Q6 runSuite.

Note: Look at the sample program `a2/args.cc` to learn how command line arguments are read in by C++ programs.

Note: Do not copy paste the example above to create a test file. The formatting might NOT be correct. This test case has been provided to you (q2.in and q2.out) within the a2 directory

Write a program to read the data and display the results of the election.

(a) **Due on Due Date 1**: Design a test suite (`suiteq2.txt`) for this program. Zip your suite file, together with the associated `.in` and `.out` files, into the file `a2q2.zip`.

(b) **Due on Due Date 2**: Write the program in C++. Save your solution in `a2q2.cc`.

3. **Note: In order to give you the maximum programming practice, there is no test suite requirements for this question. However, you might still want to create your own test cases**.

Write a program that will process a series of zero or more credit card transactions by reading in the transaction record from standard input, validating the credit card, and then outputting a verdict as to the validity of the credit card. A transaction record consists of the following fields, one per line:

- name on the credit card i.e. first name, optional initial, last name e.g. Jane A. Qiu, Fred Smith
- credit card number e.g. 4556737586899855[1]
- card expiry date (1- or 2-digit month [space] 2-digit year), [space], transaction number (positive integer, up to 5 digits in length),[space] , date (day [space] month [space] year), time (24-hour time), [space] and amount e.g. 05 18 00001 25 05 2015 2105 45.03

You may assume that all of the information is correctly formatted and is of the appropriate type, so the only form of validation that is required is to check whether or not the credit card number is valid. (Ideally, you'll eventually expand the program to validate the full transaction.) The program will only validate VISA card numbers for now. VISA card numbers start

[1]See the file `a2/validCCNums.txt` for a list of 10 valid VISA numbers that you can use in your test cases.

with the digit ‘4’, and are either 13 or 16 digits in length. Note that the number is too large to store into an integer, so you will have to manipulate the string character-by-character, and convert the individual characters to digits as necessary using `stringstream`.

The card number is validated by applying the `Luhn algorithm`. For example, consider the following 16-digit number 4556737586899855.

(1) If the first digit isn’t a 4, the card is invalid.
(2) If the length of the card number is neither 13 nor 16, the card is invalid.
(3) The rightmost digit is the `check digit`. In the example, this is the digit 5. Starting from the check digit and moving to the left, double every second digit (marked in bold). The number **4**5**5**6**7**3**7**5**8**6**8**9**9**8**5**5 now becomes i.e. 8,5,10,6,14,3,14,5,16,6,16,9,18,8,10,5.
(4) For each number over 9, add together the digits of the product. The number now becomes 8,5,1+0,6,1+4,3,1+4,5,1+6,6,1+6,9,1+8,8,1+0,5 = 8,5,1,6,5,3,5,5,7,6,7,9,9,8,1,5
(5) Add all numbers together. The sum is 90.
(6) Calculate the result of the sum modulo 10 i.e. 90 modulo 10 = 0.
(7) If the result is 0, the card number is valid.

The format of each record output is as follows:

00001 25/05/2015 21:00 $45.03 (4556737586899855, Jane A. Qiu, 05/18) valid[newline]
00001 25/05/2015 21:00 $45.03 (4556737586899856, Fred Smith, 05/18) invalid[newline]

Note that the transaction number is specified to have a width of 5 digits, and a fill character of ‘0’ The transaction date has a width of 2 set for the day and the month. The transaction time has a width of 2 for each of the hour and the minutes. The dollar amount has a precision of 2 (hint: investigate use of `fixed` and `showpoint` IO manipulators). The expiry month and year each have a width of 2.

Note: Do not copy paste the example above to create a test file. The formatting might NOT be correct. This test case has been provided to you (q3.in and q3.out) within the a2 directory

The program ends upon reaching the end of the input stream.

Due on Due Date 1: Write the program in C++. Call your program `validate.cc`.

4. In this question, you will write a program called ppmTransform that will read in pixel data for an image from standard input, optionally apply some transformations to the image and write out the final image to standard output in PPM format.

We begin by first describing the intended output format. PPM is a simple image encoding format. While PPM allows for some variation in format, you must follow the specific formatting rules given in this question. Marmoset will not understand PPM files that do not conform to the given guidelines.

In particular, the output from your program will be a Plain PPM file. Refer to `http://netpbm.sourceforge.net/doc/ppm.html` for additional details on the PPM format (Plain PPM section). We add the following specifications for the image files that should be generated by your program.

The file must start with the following header:

```
P3
Width Height
255
```

P3 is the magic number for a Plain PPM file and 255 is the maximum colour value for the pixel data. These two values should never change for the images your program generates for this assignment. Width and Height are numbers representing the width and height of the image, respectively. These two values are separated by whitespace and are positive integers.

Each line following the header, should contain the pixel data for a row of the image in order from top to bottom. The pixel data for an entire row should be on a single line, ending with a newline. A single pixel consists of three numbers, between $0 - 255$, representing the RGB value for that pixel. Consecutive RGB values should be separated by white space. For a single pixel, the individual RGB values should also be separated by white space. The following is an example image file in the correct format:

```
P3
3 4
255
0     0   0    128 128 128   255 255 255
255   0   0    255   0   0   255   0   0
0   255   0    0   255   0     0 255   0
0     0 255    0     0 255     0   0 255
```

Now we describe the input format for the ppmTransform program that you are to implement. The input is a sequence of RGB values for an image, the width of the image, the height of the image, followed by a series of transformations to apply to the image. All the input will be separated by whitespace (note this does not imply usage of newlines at specific points in the input). The possible transformations are "flip", "rotate’", and "sepia’".

The RGB values for each pixel should be stored in the following structure.

```
struct Pixel {
    unsigned int r;  // value for red
    unsigned int g;  // value or green
    unsigned int b;  // value for blue
};
```

We typically use arrays to store collections of items (say, integers). We can allow for limited growth of a collection by allocating more space than typically needed, and then keeping track of how much space was actually used. We can allow for unlimited growth of the array by allocating the array on the heap and resizing as necessary.

The latter approach will be needed for the image pixel data as the width and height of the image are not known in advance. As the RGB data is read into a one dimensional array, once the width and height are known, the pixel for location (i,j) is accessed via i * width + j.

The following structure encapsulates a partially-filled PPM pixel array:

```
struct PpmArray {
    int size;      // number of pixels the array currently holds
    int capacity;  // number of pixels the array could hold, given current
                   // memory allocation to pixels
    int width;     // width of image
    int height;    // height of image
    Pixel *pixels;
};
```

For memory allocation, you **must** follow this allocation scheme: every PpmArray structure begins with a capacity of 0. The first time data is stored in a PpmArray structure, it is given a capacity of 5 and space allocated accordingly. If at any point, this capacity proves to be not enough, you must double the capacity (so capacities will go from 5 to 10 to 20 to 40 ...). Note that there is no `realloc` in C++, so doubling the size of an array necessitates allocating a new array and copying items over. Your program must not leak memory.

Your program should support the following image transformations:

- **flip**: Write a function with the following signature

  ```
  void flipHorizontal(PpmArray&)
  ```

  The function modifies the PpmArray structure so that it now contains an image that has been flipped horizontally.

- **rotate**: Write a function with the following signature

  ```
  void rotate(PpmArray&)
  ```

  The function modifies the PpmArray structure so that it now contains an image that has been rotated 90° clockwise.

  If `i` is the row index and `j` is the column index of a pixel in the rotated image, then the following will give the required values for the new pixel array, where `ppm` contains the current image.

  ```
  newPixel[i * ppm.height + j] = ppm.pixels[ppm.width * (ppm.height - j - 1) + i];
  ```

- **sepia**: Write a function with the following signature

  ```
  void sepia(PpmArray&)
  ```

  The function modifies the PpmArray structure so that it now contains an image that has a sepia filter applied to the image.

  Apply the following formulae (exactly as give) to each pixel in the image. For `Pixel p` with result stored in `Pixel np`:

  ```
  np.r = p.r *.393 + p.g * .769 + p.b * .189
  np.g = p.r *.349 + p.g * .686 + p.b * .168
  np.b = p.r *.272 + p.g * .534 + p.b * .131
  ```

  If any of the final values above exceed 255, then set them to the maximum value of 255.

For this question, you are required to verify that input data conforms to a valid image and valid transformations. For any error conditions, print an informative error message to standard error and abort the program with a nonzero exit status.

The following files are provided for testing (`a2/q4-samples` directory):

- small.ppm (original image), small.in (program input), smallFinal.ppm (transformed image)
- castle.ppm (original image), castle.in (program input), castleFinal.ppm (transformed image). Castle image is from `https://commons.wikimedia.org/wiki/File:Castle_Neuschwanstein.jpg` and is distributed through the GNU Free Documentation License. **Note: the castle image is high resolution and takes over 4MB.** It is provided to you for local testing. You MUST NOT submit any test case that uses this image. Marmoset is setup to reject any submission greater than 1MB.
- castleSmall.ppm (original image), castleSmall.in (program input), castleSmallFinal.ppm (transformed image). You may submit a test suite which uses this image at most once.

Each sample input file contains the RGB values for the image, followed by the width and height, followed by a list of transformations. Any sample input files you create should also follow this format. Your program should run as follows:

```
./ppmTransform < small.in > output.ppm
```

Make sure the output of your program matches the given output for the sample files. Your mark depends on it.

To view a PPM file on the `linux.student.cs` environment, the following command will work:

```
display file.ppm
```

To view PPM files on your local machine various options exist. Both `ImageMagick` and `gimp` are open-source programs and should work on OS X, Windows or Linux.

A starter file `ppmTransform.cc` has been provided in your `cs246/1155/a2` directory.

Note carefully: We have set up a special Marmoset project to help you to clarify the input and output requirements. The project is called a2q4InputTest, and it expects a file called `a2q4.in`. **It is not worth any marks,** but if you submit your input in `a2q4.in`, the result of the public test will tell you the corresponding correct output.

(a) **Due on Due Date 1:** Design a test suite for this program. In general, use input/output files of an appropriate size for your test cases. In particular, use small.in as the starting point for your test cases where reasonable. You may use the smallCastle image, found in the `sample-tests` directory, but not the large castle image. Marmoset will be set to reject any submission with files larger than 1MB. Additionally, you should only need at most one test case with the castleSmall image file. All other tests can be completed using the small.* files.

Call your suite file `suiteq4.txt`. Zip your suite file, together with the associated `.in` and `.out` files, into the file `a2q4.zip`.

(b) **Due on Due Date 2:** Write the program in C++. Call your solution `ppmTransform.cc`.