Assignment #3

Due Date 1: Friday, 14 June, 2019, 5:00 pm
Due Date 2: Friday, 21 June, 2019, 5:00 pm

- **Questions 1a, 2a, and 3a are due on Due Date 1; Question 1b, 2b, and 3b are due on Due Date 2.**
- On this and subsequent assignments, you will take responsibility for your own testing. This assignment is designed to get you into the habit of thinking about testing *before* you start writing your program. If you look at the deliverables and their due dates, you will notice that there is *no* C++ code due on Due Date 1. Instead, you will be asked to submit test suites for C++ programs that you will later submit by Due Date 2.

  Test suites will be in a format compatible with that of the latter questions of Assignment 1, so if you did a good job writing your `runSuite` script, that experience will serve you well here.
- Design your test suites with care; they are your primary tool for verifying the correctness of your code. Note that test suite submission zip files are restricted to contain a maximum of 40 tests, and the size of each file is also restricted to ~~300~~ 1000 bytes; this is to encourage you not combine all of your testing eggs in one basket.
- You must use the standard C++ I/O streaming and memory management (MM) facilities on this assignment; you may **not** use C-style I/O or MM. More concretely, you may `#include` the following C++ libraries (and no others!) for the current assignment: `iostream`, `fstream`, `sstream`, `iomanip`, `string`, and `utility`. Marmoset will be setup to **reject** submissions that use C-style I/O or MM, or libraries other than the ones specified above.
- Each question on this assignment asks you to write a C++ program, and the programs you write on this assignment each span multiple files. Moreover, each question asks you to submit a `Makefile` for building your program. For these reasons, we **strongly** recommend that you develop your solution for each question in a separate directory. Just remember that, for each question, you should be *in* that directory when you create your zip file, so that your zip file does not contain any extra directory structure.
- We will manually check that you follow a reasonable standard of documentation and style, and to verify any assignment requirements that are not automatically enforced by Marmoset. Code to a standard that you would expect from someone else if you had to maintain their code. Further comments on coding guidelines can be found here: `https://www.student.cs.uwaterloo.ca/~cs246/current/AssignmentGuidelines.shtml`
- We have provided some code and executables in ~~the subdirectory `codeForStudents`~~ the subdirectory for each question under the `a3` subdirectory in your repository.
- **You may not ask public questions on Piazza about what the programs that make up the assignment are supposed to do.** A major part of this assignment involves designing test cases, and questions that ask what the programs should do in one case or another will give away potential test cases to the rest of the class. Questions found in violation of this rule will be marked private or deleted; repeat offences could be subject to discipline.

1. In this exercise, you will write three simple C++ classes, `Date`, `Call`, and `Plan`, each implemented as a `struct`, to represent a simple Bell Canada "Basic Phone Plan". You have been provided with some starter code for each of these classes, `date.{h,cc}`, `call.h` and `plan.h` respectively, and a sample executable. **Read these files for the required definitions of your structures, their methods and the I/O format.** We have also provided a very simple program that shows how to use an enumerated class/struct, `enum.cc`. In the comments at the top of the `enum.cc` file, you can find links that explain what an *enumerated class* is, and why you should be using it.

   You will need to implement the following:

   - `Date`: the constructor, `operator==`, `operator<`, `operator<<` and `operator>>`.
     Implementation notes:
     - The easiest way to make the input operator work is to *overwrite* the contents of the original `Date` object passed in.

- The provided starter code in `date.cc` implements `getDay` for you.
- **You may assume that any date specified is valid.**
- (You may find the `std::setfill`, and `std::setw` routines from `<iomanip>` useful for formatting information. `setprecision` is available from `<iostream>` and is useful for outputting a fixed number of decimal points. C++11 uses `std::locale`, `std::show_base` and `std::put_money` to output currency if you want to experiment with a different approach.[1])

- `Call`: the constructor, `operator<<` and `operator>>`.
  Implementation notes:
  - Your input operator should use the `Date` input operator as part of its logic. The output operator should be structured similarly.
  - **You may assume that the information for the call is valid, and that the duration will never make the end date of the call differ from the start date; otherwise, the calculation is more complex than desirable for this level of problem. Similarly, you do not need to worry about call times overlapping.**
  - The minimum length of a call is 1 minute.
  - You may add comparison operators to `Call` if you wish.
- `Plan`: the constructor, destructor, `operator<<`, `add` and `calculateBill` methods.
  Implementation notes:
  - You may add an input operator if you wish.
  - The `Plan` class implements the logic for the phone plan.
  - The monthly fee for the plan is $25.
  - There are 150 free minutes, either local or long distance, for calls made neither in the evening nor on weekends.[2]
  - If the customer exceeds their 150 minutes, they are charged $0.50 per minute for every minute over that they have gone.
  - An unlimited number of calls may be made in the evening or on weekends.
  - Over the course of a month, the calls are added to the plan. The calls may arrive in any order, though realistically they'd be in chronological order.
  - It is up to you how to store the calls i.e. array or linked list, and whether you insert the calls in order, or sort them as necessary.
  - The output operator prints the current set of calls in the list, in ascending order by date and then by time. If you choose to sort them, you do not need to worry about efficiency. Each call appears on a separate line.
  - At the end of the month, the bill is calculated and printed as part of the bill calculation operation, and the number of calls made is reset to zero.
  - See the provided sample executable for the output format of the bill.

You may not change the contents of the header files other than by adding your instance variables, helper methods, and comments i.e. the interface must stay exactly the same.

The test harness `a3q1.cc` is provided with which you may interact with your plan for testing purposes. **The test harness is not robust and you are not to create tests for it i.e. don't test how it handles errors in the commands provided, just for the `Plan` class. Do not change this file.** See the file for the command descriptions.

(a) **Due on Due Date 1:** Design the test suite `suiteq1.txt` for this program and zip the suite into `a3q1a.zip`.

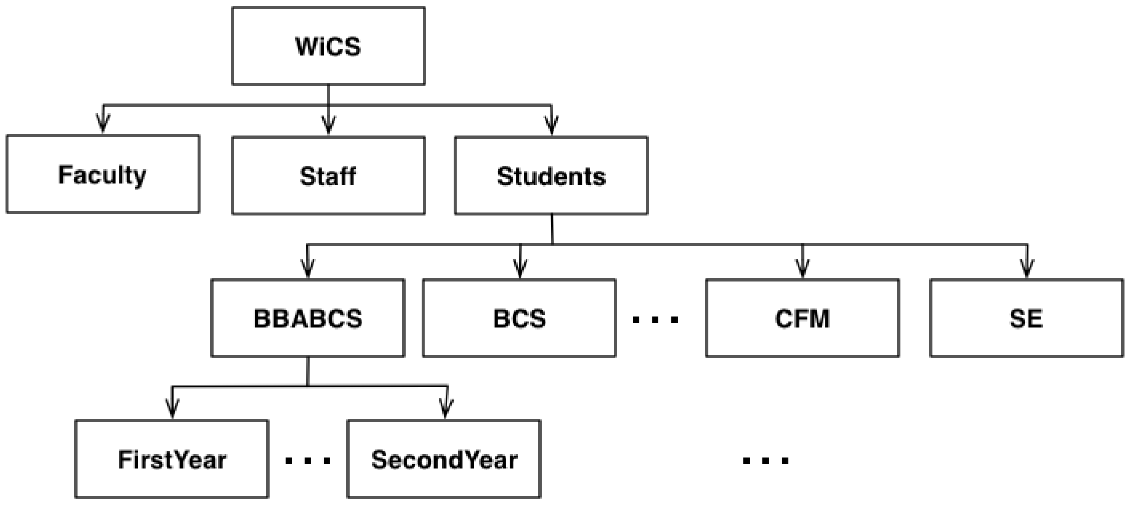
(b) **Due on Due Date 2:** Implement this in C++ and place the files `Makefile`, `a3q1.cc`, `date.h`, `date.cc`, `call.h`, `call.cc`, `plan.h` and `plan.cc` in the zip file, `a3q1b.zip`. Your Makefile must create an executable named `plan`. Note that the executable name is case-sensitive.

[1]Note that it assumes that the two rightmost digits before the decimal (in Canadian currency) are the ”cents”, so you'd have to multiply your amount by 100 to make it output properly. Note that only English (and C) locales have been installed in the student environment.

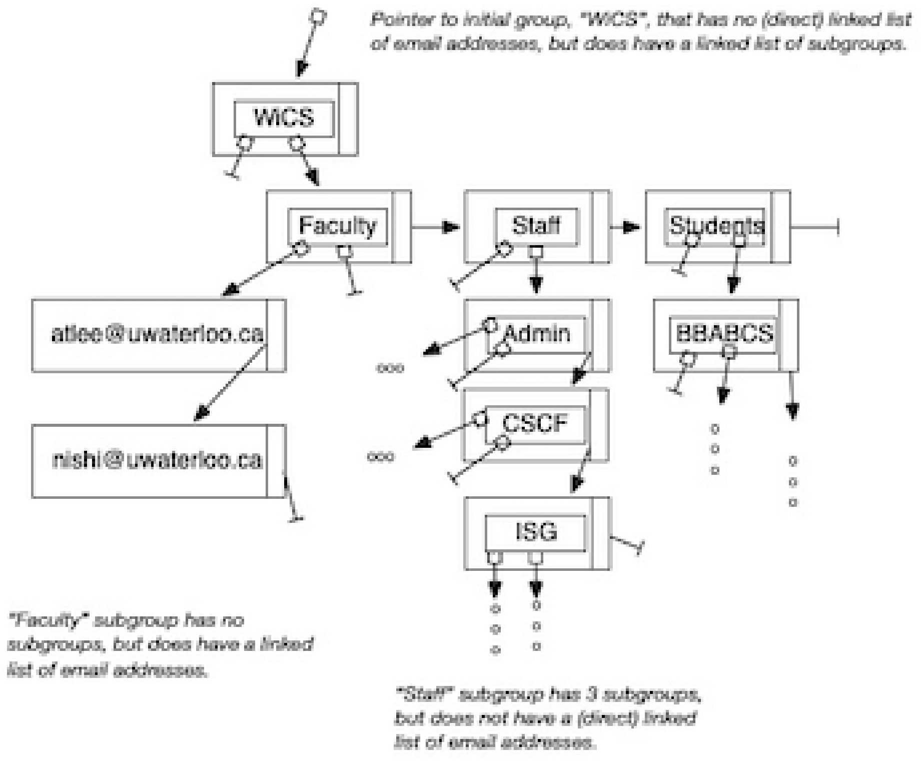
[2]Evenings start at 6pm (1800) and end at 7am (0700). Weekends start at 6pm on Friday, and end at 7am on Monday.

2. As part of an email system, you've been asked to implement a tree-like data structure to represent email groups. In its simplest form, the tree consists of a single node containing a single email address. The tree could also consist of 1 or more nested groups. For example, the Women In CS (WiCS) mailing list group could contain a group for the faculty representatives, one for the staff representatives, and then a group for the students, where the students are subdivided by plan, and then by year so that the entire group or particular subsets can be targeted in mass mailings.

   For example, the following diagram



   could be implemented by a data structure like the one shown in the following diagram.


Pointer to initial group, "WiCS", that has no (direct) linked list of email addresses, but does have a linked list of subgroups.
WiCS
Faculty
Staff
Students
atlee@uwaterloo.ca
nishi@uwaterloo.ca
Admin
CSCF
ISG
BBABCS
"Faculty" subgroup has no subgroups, but does have a linked list of email addresses.
"Staff" subgroup has 3 subgroups, but does not have a (direct) linked list of email addresses.

   You have been provided with some starter code in the file `group.h`. **Read this file for the required definitions of your structures, their methods and the I/O format.** For your submission you must add all requisite definitions, declarations to `group.h` and all routine and member definitions to `group.cc`. **You may not change the contents**

of the header file other than by adding your instance variables, helper methods, and comments i.e. the interface must stay exactly the same.

You have also been provided with a test harness, `a3q2.cc`, a sample input file, `sample.in`, and the corresponding output, `sample.out`, produced by the sample executable `emailgroups`. Read these files carefully to understand how to interact with the test harness and the sample executable. **Do not change the test harness.**

The test harness can be compiled with your solution to test (and then debug) your code. **The test harness is not robust and you are not to create tests for it, just for the `Group` structure. Similarly, do not test for invalid email addresses.** An email address is just a string. There is no requirement that it be a valid email address or follow the format of an email address. For our purposes, any arbitrary string that does not contain whitespace could be a valid email address.

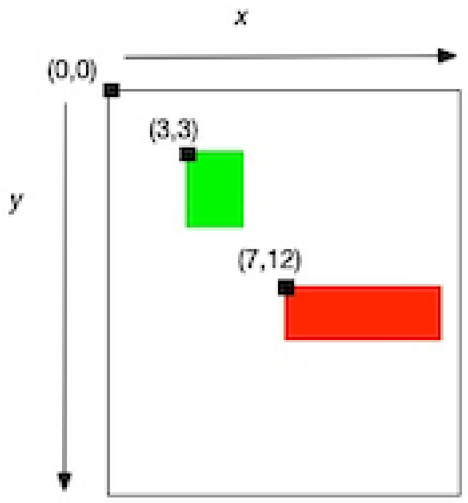
It is *strongly* suggested that you first implement and test your linked list code before you work on the rest of the program to ensure that it is correct.

(a) **Due on Due Date 1**: Design the test suite `suiteq2.txt` for this program and zip the suite into `a3q2a.zip`.

(b) **Due on Due Date 2**: Implement this in C++ and place your `Makefile`, `a3q2.cc` and all `.h` and `.cc` files that make up your program in the zip file, `a3q2b.zip`. Your Makefile must create an executable named `emailgroups`. Note that the executable name is case-sensitive.

3. In this question you will implement the `Canvas` class where each `Canvas` holds zero or more `Rectangle` objects. While you are allowed to add, remove, and manipulate rectangles within a given canvas, and perform some simple operations upon the canvas itself, the key idea being tested is that you have correctly implemented the move and copy semantics for your structures to make deep copies.

Following the form of classic raster screen display such as a Cathode Ray Tube monitor uses, a `Canvas` is defined such that its upper-left corner has the coordinates (0,0).[3] X-coordinates increase towards the right (there is no negative value). Y-coordinates increase downwards (there is no negative value).



A `Canvas` starts initially empty, with the height and width both integers with the value 0. When a `Rectangle` is added to a `Canvas`, the `Canvas` ”stretches” to accommodate the new `Rectangle` (if necessary) by calculating its new dimensions based upon the `Point` of the `Rectangle` and its dimensions. Rectangle indices j are used to uniquely identify each rectangle within the canvas. The indices start at 0, and if a ~~canvas~~ rectangle is removed, the indices of the rectangles after the one removed need to be decreased by 1. (Hint: this may not require much extra work, depending upon your implementation of how rectangles are stored.)

You are given some header files that define the interfaces for the `Point`, `Rectangle`, and `Canvas` classes, named `point.h`, `rectangle.h`, and `canvas.h` respectively. **Read these files for the required definitions of your classes, their methods and the I/O format. You may not change the public interfaces**; however, you may add

[3]Ideally we'd prevent points from having negative values for their x- and y-coordinates, but that is not a requirement of this question.

private instance variables, private helper methods, and comments. You will also need to fill in the necessary private declarations for the information held in these classes.

You have also been provided with a simple test harness, `a3q3.cc`, a sample input file, `sample.in`, and the corresponding output, `sample.out`, produced by the sample executable, `canvas`, for the output format of the `Canvas` `Rectangle` and `Point` objects. Read these files carefully to understand how to interact with the test harness and the sample executable. **Do not change the test harness.**

(a) **Due on Due Date 1**: Design the test suite `suiteq3.txt` for this program and zip the suite into `a3q3a.zip`.

(b) **Due on Due Date 2**: Implement this in C++ and place your `Makefile`, `a3q3.cc` and all `.h` and `.cc` files that make up your program in the zip file, `a3q3b.zip`. Your Makefile must create an executable named `canvas`. Note that the executable name is case-sensitive.