# Assignment #3

**Due Date 1:** Friday, 12 October, 2018, 5:00 pm
**Due Date 2:** **Tuesday, 23 October**, 2018, 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 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`.
- **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, each implemented as a `struct`, to represent a simple Bell Canada "Basic Phone Plan".

   Use the following structure definition for a date:

```
struct Date {
   enum class DayOfWeek {Sunday, Monday, Tuesday, Wednesday,
       Thursday, Friday, Saturday};
   int year, month, day;
   Date( int y, int m, int d );
   DayOfWeek getDay();
   bool operator==( Date other );
   bool operator<( Date other );
};
```

The provided starter code for `Date` implements `getDay` for you. You will need to provide the implementation of the constructor, the specified comparison operators, and the appropriate input and output operators. A `Date` is specified as "year/month/day", so October 31, 2018, would appear as `2018/10/31` when being read from input, or printed. (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]) **You may assume that any date specified is valid.**

A phone call (`Call`) consists of the date of the start of the call, the time the call started (in a 24-hour format, where midnight is listed as `0000`)[2] and the number of minutes the call lasted, rounded up to the nearest minute (a call's duration must therefore be greater than 0). Use the following structure definition for a call:

```
struct Call {
   Date date;
   int startTime, duration;
   Call( Date date, int startTime, int duration );
};
```

You will need to provide the implementation of the constructor, and the appropriate input and output operators. (Note that your `Call` input operator should use the `Date` input operator as part of its logic. The output operator should be structured similarly.) The information for a `Call` appears on a single line. It consists of the date the call took place, followed by two positive integer values, the start time and the duration, separated by one or more whitespace characters (spaces or tabs). **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.**

The `Plan` class implements the logic for the phone plan. In particular, 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.[3] 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. 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 reset to zero.

The `Plan` class must be properly initialized by a constructor, cleaned up by a destructor, and support the following operations:

| *Method* | *Description* |
|---|---|
| `void add( Call call )` | Adds a call to the calls made so far for the current month. In order to read in a `Call`, you must implement the overloaded `operator>>` for a `Call`. |
| `void calculateBill()` | Uses the output operator to print the list of calls made, outputs the amount of money owed, and then clears out the list of calls made. Note that you must implement and use the `operator<<` for the `Call` as part of your implemenation. |

Implement the specified operations for the `Plan`. (Some starter code has been provided for you in the files `date.h`, `call.h`, and `plan.h`, along with a sample executable, `phone`.) Note that the output operator for the plan provides a list of the calls made, in ascending order by date and then by time. Each call appears on a separate line. **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 devise tests for it, just for the `Plan` class. Do not change this file.**

[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.

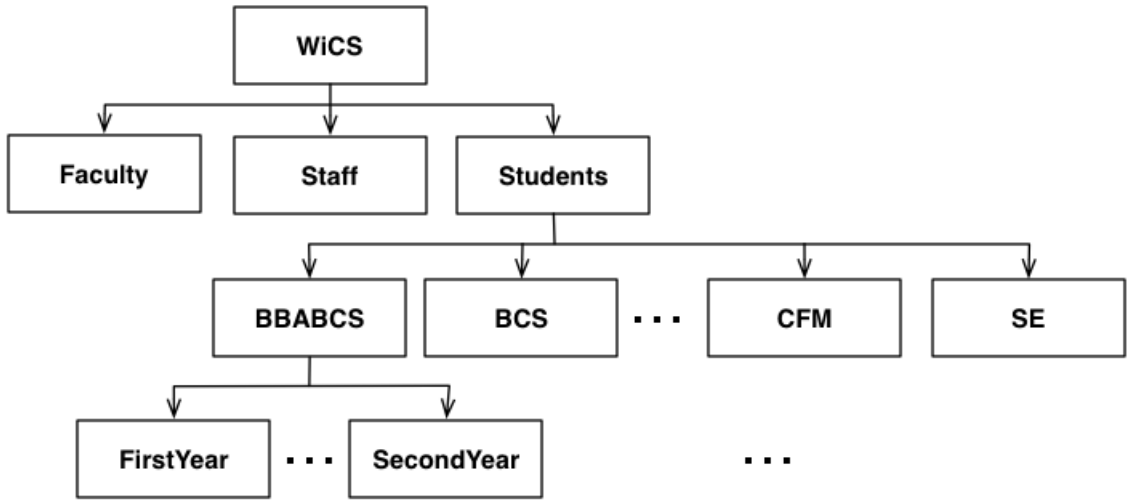
[2] In C++, starting an integer with a 0 implies that you are entering a number in octal i.e. base 8, so when entering the number in your code as opposed to typing it as input from the keyboard, omit leading zeroes.

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

(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.

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 and a single email address[4]. It could also consist of nested groups. For example, the Women In CS (WiCS) mailing list could have 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.




Some implementation notes follow:

- You have been provided with the files `group.h` and `a3q2.cc`. There is a sample executable named `emailgroups`.
- **You may not change the contents of the header files other than by adding your private instance variables, private helper methods, and comments i.e. the interface must stay exactly the same.**
- The declaration of the `Group` type can be found in the provided `group.h` file. For your submission you must add all requisite declarations to `group.h` and all routine and member definitions to `group.cc`.
- In order to complete the `Group` implementation, you will also need to implement the nested inner classes: `GroupNode`[5] and `EmailNode`[6] classes. For your submission you must add all requisite declarations to `group.h` and all routine and member definitions to `Group.cc`. **Note:** `Group` has been declared as a `friend` of both `GroupNode` and `EmailNode` so that its code can access their private information if necessary.
- A `Group` may have 0 or more email addresses, and 0 or more subgroups.
- Searching for an email address first starts in the list of email addresses for the root `Group` node. The search stops as soon as the first occurrence is found. If the address cannot be found there, then each subgroup in turn is searched. Since each subgroup is a `Group`, the search action follows the previously specified order.
- Searching a group for a group name starts with the root `Group` node. The search stops as soon as the first occurrence is found. If the group name cannot be found there, then each subgroup in turn is searched. Since each subgroup is a `Group`, the search action follows the previously specified order.
- A `Group` can only be deleted if it is a subgroup of the `Group` node currently being indexed; otherwise, the command fails by doing nothing. For example, if the group $g_0$ is the pointer to the WiCS group as in the diagram, then the WiCS group will *not* be removed from $g_0$.

[4]Email addresses are just simple strings, and there is no requirement that they constitute 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. You are not required to test for invalid email addresses.

[5]Linked list of `Group` pointers, used to contain subgroups to the current group.

[6]Linked list of email addresses, implemented as `string` objects embedded in the nodes.

- Note that the information in each list is stored in the standard `string` lexicographic[7] order using the standard `string` comparison operators. Thus, the output will be in lexicographic order.
- As well, since there is no requirement that email addresses or group names be unique, duplicate group names or email addresses added to any collection of group names or email addresses are added **after** the last existing email address or group name of the same name/address, as appropriate.
- **It is *strongly* suggested that you first implement and test your linked list code before you work on the rest to ensure that it is correct.**
- The provided test harness, `a3q2.cc`, can be compiled with your solution to test (and then debug) your code. **The test harness is not robust and you are not to devise tests for it, just for the `Group` class. Do not change this file.** The test harness allows you to have up to 10 groups defined at one time, identified as `g0` to `g9`. If a group has not been initialized, it consists of a `nullptr`. Most of the test harness commands cannot be performed upon an uninitialized group. Additionally, the user prompts are printed to standard error so that they will not interfere with the output produced, and thus make it easier to write your test files.

The test harness commands consist of:

| *Command* | *Description* |
|---|---|
| b $g_i$ *name* | Initializes group $g_i$ by calling its constructor and passing in the group name, *name*. $g_i$ must initially be a `nullptr`. |
| aa $g_i$ *email* | Uses `Group::addAddress` to add *email* to $g_i$. $g_i$ must not be a `nullptr`. |
| ag $g_i$ $g_j$ | Uses `Group::addGroup` to add $g_j$ to $g_i$ and sets $g_j$ to `nullptr`. Neither $g_i$ nor $g_j$ must be a `nullptr`. |
| ra $g_i$ *email* | Uses `Group::removeAddress` to remove the first occurrence of *email* from $g_i$. $g_i$ must not be a `nullptr`. |
| rg $g_i$ *name* | Uses `Group::removeGroup` to remove the first subgroup of $g_i$ that has a name that matches *name*. $g_i$ must not be a `nullptr`. |
| sa $g_i$ *email* | Uses `Group::findAddress` to return an `Group::EmailNode*` set to the node that contains the first occurrence of *email* in $g_i$ or `nullptr` if no such address can be found. $g_i$ must not be a `nullptr`. |
| sg $g_i$ *name* | Uses `Group::findGroup` to return a `Group::GroupNode*` set to the node that contains the first occurrence of *name* in $g_i$ as a *subgroup* or `nullptr` if no such subgroup can be found. $g_i$ must not be a `nullptr`. |
| p $g_i$ | Uses `operator<<` to output group $g_i$ to standard output. $g_i$ must not be a `nullptr`. |

(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 classes to make deep copies.

   A `Canvas` is defined such that its upper-left corner has the coordinates (0,0).[8]

   Initially empty, its height and width are both integers with the value 0. When a `Rectangle` is added to a `Canvas`, the `Canvas` "stretches" to accommodate the new `Rectangle` (if necessary). A `Rectangle` has a colour[9], a height, width, and a position for its upper-left corner. You are given some header files that define the interfaces for the `Point`,

[7] In lexicographic order, the string "2" comes after the string "11" since they are not treated as numbers when the comparison is performed. As per usual, the string "cat" comes before "dog" in lexicographic ordering.

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

[9] Colours are defined to be red, green, blue, yellow, orange, black and white. When reading in a colour as a character, the first letter of the colour name corresponds to the colour, except for black where it uses the letter 'a' since blue already uses 'b'.

`Rectangle`, and `Canvas` classes, named `point.h`, `rectangle.h`, and `canvas.h` respectively. You may not change the public interface; however, you may add 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`, to let you interact with your classes. It starts by defining 5 canvases, with ids in the range 0 to 4. It performs no error-checking upon the commands, other than ensuring that the canvas id specified is in the range of 0 to 4, and that the rectangle index is in the range 0 to (*number of rectangles in the canvas*). The test harness commands consist of:

| *Command* | *Description* |
|---|---|
| a $c_i$ *colour* $x$ $y$ *height width* | Adds a rectangle with the Rectangle::Colour corresponding to *colour*, `Point` ($x$,$y$) and dimensions of *height* and *width* to the canvas $c_i$ by calling the rectangle input operator and the `Canvas::add` method. |
| r $c_i$ $r_j$ | Removes the rectangle at index $r_j$ from canvas $c_i$. |
| p c $c_i$ | Prints the contents of canvas $c_i$ using the `Canvas` output operator. |
| p r $c_i$ $r_j$ | Prints the rectangle at index $r_j$ of canvas $c_i$ using the `Rectangle` output operator. |
| s c $c_i$ $r_j$ *colour* | Changes the colour of the rectangle at index $r_j$ of canvas $c_i$ to the new colour, *colour*. |
| s t $c_i$ $r_j$ $x$ $y$ | Moves the upper-left `Point` of the rectangle at index $r_j$ of canvas $c_i$ by translating it by the specified amounts in the $x$ and $y$ directions. You may assume that the coordinate values do not go below 0. |
| s s $c_i$ $r_j$ $s_1$ $s_2$ | Scales the height and width of the rectangle at index $r_j$ of canvas $c_i$ by multiplying the dimensions by the specified amounts. The resulting values are integers, and you may assume that they will stay greater than 0. |
| c c $c_i$ | Creates a local copy of the specified canvas, $c_i$, using the copy constructor and outputs it. Note that the copy is modified to see if the original remains untouched as part of checking that your copy is indeed a deep copy. |
| c m $c_i$ | Creates a local copy of the specified canvas, $c_i$, using the move constructor and outputs it. Note that the copy is modified to see if the original remains untouched as part of checking that your copy is indeed a deep copy. |
| = c $c_i$ $c_j$ | Uses the copy assignment operator to copy the contents of canvas $c_j$ to canvas $c_i$. |
| = m $c_i$ $c_j$ | Uses the move assignment operator to move the contents of canvas $c_j$ to canvas $c_i$. |

See the sample executable, `canvas`, for the output format of the `Canvas` and `Rectangle` objects.

(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.