**COMP2404B (Fall 21) -"Introduction to Software Engineering"**

**Assignment 1 of 4 - Due Sunday October 3rd, 11:59 pm**

## 1 Submission Instructions

Submit to Brightspace on or before the due date a compressed file (.tar or .zip) that includes

1. Header and source files for all classes instructed below.
2. A working Makefile that compiles and links all code into a single executable. The Makefile should be specific to this assignment - do not use a generic Makefile.
3. A README file with your name, student number, a list of all files and a brief description of their purpose, compilation and execution instructions, and any additional details you feel are relevant.

## 2 Learning Outcomes

In this tutorial you will learn the basics of Classes in C++. You will make a few simple classes, populate the members, and use functions and constructors in a meaningful and appropriate way. You will learn how to provide a working Makefile and do some rudimentary testing.

## 3 Overview

The Carleton Library has rooms that students may book and use. In this assignment you will make a room booking application. A student may request a certain room at a certain day and time. Or search for rooms that match certain criteria (such as maximum capacity, computers and whiteboards) and check their availability.

## 4 Classes Overview

This application will consist of 5 classes. A brief description of each class follows.

1. `Student` - name and student number
2. `Room` - name and other criteria
3. `Date` - year, month, day, hour and duration
4. `Reservation` - a `Student`, a `Room`, and a `Date` object representing the who, where and when of a `Reservation`.
5. `Library` - a collection of `Students`, `Rooms` and `Reservations` along with some functions for managing them (add, find, isFree etc).

## 5 Instructions

Download the starting code from Brightspace. It includes some global functions that you are to use for testing. All member variables are `private` unless otherwise noted. All member functions are `public` unless otherwise noted. Some return values are not explicitly given. You should use your best judgment (they will often be `void`, but not always).

### 5.1 The Student Class

Make a student class. This will be a simple data class with 2 data members and a `bool lessThan` method for sorting.

1. Make two member variables:
   (a) `name`: a string
   (b) `number`: a string
2. Make a no argument constructor that initializes `name` and `number` to suitable default values. Make a second constructor that takes two strings as arguments. The first string should be used to initialize `name` and the second string should used to intialize `number`.
3. Make a copy constructor.
4. Make getters and setters for the member variables.
5. Make a function `bool lessThan(Student& s)`. Return true if `this->number` comes before `s.number` alphabetically, return false otherwise.

### 5.2 The Room Class

Make a `Room` class. This room has certain features that we will track using member variables. Instead of using room numbers, to make the rooms more distinct we will give them names. Each room has a capacity of at least 1 person. There is no practical maximum for the capacity (at some point in the future the library could have an auditorium). Each room has a number of computers from 0 to infinity. And each room has a whiteboard or does not. When a student makes a room request they will request certain criteria which we will try and match (though this will be done in the `Library` class).

1. Give your room class the following member variables: `string name`, `int capacity`, `int computers`, `bool whiteboard`.
2. Make a constructor with parameters in the same order given above. Make a copy constructor.
3. Make getters and setters for the member variables. The getter for whiteboard should be `bool hasWhiteboard()`.
4. Make a function `bool meetsCriteria(int capacity, int computers, bool whiteboard)`. This function should return true if the capacity, computers, and whiteboard parameters of the Room meets or exceeds the parameters given. For example, if you ask for a capacity 2 a Room with capacity 3 and any number of computers and a whiteboard is acceptable. Give the parameters have appropriate default values. So, for instance, someone could call `bool meetsCriteria(3)` if they wanted a room with capacity of 3, regardless of whether there are computers or a whiteboard.
5. Make a function `bool lessThan(Room& r)`. Return true if `this->name` comes before `r.name` alphabetically, and return false otherwise.

### 5.3 The Date Class

Modify the Date class that we saw in class. We want to modify it so that, in addition to year, month and day, it keeps track of the hour and duration of when we are to reserve and / or use a room.

1. Add a global integer constant[1] to the header file to indicate the maximum allowable duration and set it to 3.

[1] In future assignments we will get away from using constants since they are bad practice. But as a C++ programmer it is still necessary that you understand how they function.

2. Add two new integer member variables, one for `hour` and one for `duration`.
   (a) Make setters for both of these member variables that do appropriate checks on the values. We will use a 24-hour clock, so the `hour` should be an integer from 0 to 23. The `duration` should be an integer from 1 to the maximum allowable duration defined above. If the values fall outside of the allowed range, set the member variables to the nearest allowable value.
3. Adjust the 3-argument constructor to accept two more integer arguments. It should now look like

   `Date(int year, int month, int day, int hour, int duration)`
4. Make any required changes to the copy constructor, the no-argument constructor, the `print()` function, the setters, getters, and any other relevant functions.
5. Change the `lessThan(Date&)` method to also compare the `hour` members. The purpose of this method is to sort the reservations. As such, we should NOT compare durations in this method. We only compare when the Dates begin.
6. Make a method `bool overlaps(Date& d)`. This function should return true if two `Dates` overlap and false if they do not overlap. Two dates, `Date d1` and `d2`, overlap if they start on the same date and at the same hour. They also overlap if `d1` and `d2` are on the same date and, for example, `d1.hour == 13` and `d2.hour == 14` and `d1.duration > 1`. Essentially both `Dates` define a range of time, and if those ranges overlap this function should return true. You may assume that different days do not overlap regardless of duration, that is, you may ignore the case where a Date starts at hour 23 on one day with a duration of 2 and another Date starts at hour 0 the next day.

## 5.4 The Reservation Class

The Reservation class's main function is to group together a Student, a Room and a Date. It also provides functions to be used to order the Reservation and check if it overlaps with another Reservation. However, much of this functionality has already been implemented in the Date class, so these functions will utilize the Date functions.

1. Your Reservation class should have 3 members, a `Student` pointer, a `Room` pointer, and a `Date` pointer.
2. Make a 3 argument constructor with parameters `Student*, Room*, Date&` in that order. Initialize the member variables.
3. Make a destructor. Consider carefully which of its 3 member variables the Reservation class should be responsible for deallocating.
4. Make getters for the 3 members. We do not want setters; the members are initialized in the constructor. If we need to change the members we make a new Reservation.
5. Make a `lessThan(Reservation& res)` function. It should return true if the `this->date` is before `res.date`, false otherwise.
6. Make an `bool overlaps(string r, Date& d)` function. This function returns true if the `r` argument matches the name of the `room` member variable and `this->date` overlaps `d`, and false otherwise.
7. Make a `print` function. This function may call on the `print` functions of the Student, Date, and Room classes, however there should some additional output mentioning that these are part of a Reservation.

### 5.5 The Library Class

The Library class will maintain a collection of Students, a collection of Rooms, and a collection of Reservations. There will also be functions related to making Reservations.

1. Member variables: There are 3 collections in the Library class, all implemented as arrays.
   (a) Make a *dynamically allocated array* of Student *pointers*. The Library is responsible for allocating and deallocating this array and the objects it contains pointers to. Students should be stored increasing order in the order defined by their `lessThan` function.
   (b) Make a *statically allocated array* of Room *pointers*. The Library is responsible for allocating and deallocating the objects it contains pointers to. Rooms should be stored in increasing order in the order defined by their `lessThan` function.
   (c) Make a *dynamically allocated array* of Reservation *pointers*. The Library is responsible for allocating and deallocating this array and the objects it contains pointers to. Reservations should be stored in increasing order in the order defined by their `lessThan` function.
   (d) Make sure you track the number of elements in each of the above arrays.
2. Make a default (no-argument) constructor that initializes member variables.
3. Make a destructor that deletes all memory allocated in the Library.
4. Make a `bool addStudent(const string& name, const string& number)` function to add a Student to the array. If there is no room in the array return false. Otherwise make a new `Student` object, add it to the proper array and return true. Be sure that the `Student` is added to the array in the proper sorted order.
5. Make a `bool addRoom` function whose parameters are the same as those in the 4 argument constructor in the `Room` class. Give each parameter, except for the `name` parameter, a default value. The room capacity should be default 1, the number of computers should be default 0, and there should be no whiteboard. If there is no space left in the `Room` array return false. Otherwise make a new `Room` object using the supplied parameters, add it to the `Room` array and return true. Be sure the `Room` is added to the array in the proper sorted order.
6. Make a `bool getStudent(const string& name, Student** student)` function. If there is a `Student` named `student` in the Library it should be stored in `Student** student` and the function should return true. Otherwise the function should return false.
7. Make a `bool getRoom(const string& roomName, Room** room)` function. If there is a `Room` named `roomName` in the Library it should be stored in `Room** room` and the function should return true. Otherwise the function should return false.
8. Make a `bool isFree(const string& room, Date&)` method. If the room doesn't exist in the Library return false. Otherwise return true if there is currently no Reservation for the given Room that overlaps the given Date, and false otherwise.
9. Make a `bool makeReservation(const string& student, const string& room, Date&)` method. If either the room or student does not exist return false. Otherwise this method should check if the room is free on the given Date. If it is not free, or the Reservation array is full, return false and make no other changes. If the room is free, make a new Reservation object and add it to the Reservation array in the correct location (sorted using `compare`)and return true.
10. (Bonus) Make a `void update(Date&)` function. This should remove all Reservations from the array that are on or before the Date given. Be sure to manage the memory properly and to close all gaps in the array.

### 5.6 The `main` Function

1. In the `a1-global.cc` file there is a function `testDates(Date& d1, Date& d2, bool shouldOverlap)` for testing the `overlaps` function of the Date class. To use `testDates`, supply two Dates and a boolean indicating whether the Dates should overlap. For instance, for the first test you may use
   ```
   Date d1(2020,1,24,11,3);
   Date d2(2020,1,23,11,3);
   testDate(d1,d2,false);
   ```
   Since d1 and d2 are on different days they should not overlap, hence we supply `false` to the boolean `shouldOverlap` parameter. At the top of your `main` function add at minimum the test cases described below.
   (a) **This is the example test above, which you may use.** Make two Dates that are on different *days*. They should not overlap.
   (b) Make two Dates on the same Year, Month and Day. The difference in the `hour` member should be at least 3. They should not overlap.
   (c) Make two Dates on the same Year, Month and Day. The difference in the `hour` member should be 1. The `duration` of the earlier Date should be 1 and the `duration` of the later Date should be 3. They should not overlap.
   (d) Make two Dates on the same Year, Month and Day. The `hour` members for each Date should be 1 or 2 hours apart, and the `duration`s should cause them to overlap.
   (e) Make two Dates on the same Year, Month and Day and hour. They should overlap.
2. Make a new Library object (it can be statically or dynamically allocated).
3. Call the `populate(Library&)` global function given to you in the `a1-global.cc` file to add Rooms and Students to your Library.
4. Call the `testReservations(Library&)` function given to you in the `a1-global.cc` file.
5. Correct any errors you see.

## 6 Constraints

Your program must comply with all of the rules of correct software engineering that we have learned during the lectures, including but not restricted to:

1. The code must be written in C++98 and it must compile and execute in the default course VM provided. It must NOT require any additional libraries, packages, or software besides what is available in the standard VM.
2. Your program must not use any classes, containers, or algorithms from the standard template library (STL) *unless expressly permitted.*
3. Your program must be written in Object-Oriented C++ (do not substitute C functions where C++ functions exist). To wit:
   (a) Do not use any global functions or variables other than the `main` function and those functions expressly permitted or provided for initialization and testing purposes.
   (b) Do not use `struct`s. Use classes instead.
   (c) Input and output should be done using `cin` and `cout` objects - don't use `printf`.
   (d) Pass objects by *reference* or by *pointer*. Do not pass by value.
   (e) Reuse existing functions wherever possible.

(f) Basic error checking must be performed.

(g) All dynamically allocated memory must be deallocated. Every time you use the `new` keyword to allocate memory, you should know exactly when and where this memory gets `delete`d. Use `valgrind` to ensure there are no memory leaks.

4. All classes should be reasonably documented (remember the best documentation is expressive variable and function names, and clear purposes for each class).

## 7 Grading

### 7.1 Marking Components

- 10 marks: `Student`
- 10 marks: `Room`
- 12 marks: `Date`
- 14 marks: `Reservation`
- 20 marks: `Library`
- 4 marks: `main` function
- 10 marks: `Date` class testing

Total: 80 marks

### 7.2 Execution and Testing Requirements

1. All marking components must be called and execute successfully to earn marks.
2. All data handled must be printed to the screen to earn marks (make sure `print` prints useful information).

### 7.3 Deductions

#### 7.3.1 Packaging errors:

1. 10 marks: Missing Makefile
2. 5 marks: Missing README
3. up to 10 marks: Failure to separate code into header and source files.
4. up to 10 marks: Readability - bad style, missing documentation.

#### 7.3.2 Major design and programming errors:

1. 50% of a marking component that uses unauthorized global variables or `struct`s.
2. 50% of a marking component that consistently fails to use correct design principles.
3. 50% of a marking component that uses prohibited library classes or functions.
4. up to 10 marks: memory leaks reported by `valgrind`.

#### 7.3.3 Execution errors:

1. 100% of any marking component that cannot be tested because it doesn't compile or execute in the course VM, or the feature is not used in the code, or data cannot be printed to the screen. In short: your program must convince the TA that it works and works properly **without** the TA having to fix or modify your code. TAs are not required to debug, inspect, or fix non-working code after an assignment has been submitted.