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 (where necessary), compilation and execution instructions, and any additional details you feel are relevant.

2 Learning Outcomes

In this assignment you will learn the basics of classes, arrays, and static and dynamic memory 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. You will learn array basics, and implement a small amount of application logic.

3 Overview

In this assignment you will make a `Hotel` application for tracking `Rooms` and `Reservations` for those `Rooms`. The `Hotel` class will store records of the `Rooms` , and each `Room` will keep track of its own `Reservations`. Each `Room` should be able to print out all of its `Reservations`. The application should be able to print out all of its `Rooms`. In addition, someone may request a `Room` with certain features (such as a King sized bed) for a certain `Date` and for some number of days. The application should verify that a `Room` meeting the needed criteria is free during that time, and if so, create the `Reservation` and add it to the appropriate `Room`.

The `Rooms` have criteria such as capacity (the number of people who can stay in the `Room`), bed type (such as King or Queen or Double), and whether the Room has a fridge (it either has or does not have a fridge and so is represented by a boolean value). When someone reserves a `Room` that `Room` should meet the desired criteria of the customer.

We will be implementing the back-end of the application, but not a user-facing interface. Instead of a user-facing interface you will be supplied with a test suite for your application. In addition to writing your application you will be responsible for writing some of the tests.

4 Classes Overview

This application will consist of 4 classes. A brief description of each class follows.

1. `Date` (you may use the `Date` class from the class demonstration code available on Brightspace) - year, month, day
2. `Reservation` - the name of the client making the reservation, the check in `Date`, and the number of days that the client will be staying.
3. `Room` - details of a room as well as an array of `Reservations` for that room.
4. `Hotel` - the main control class that controls the interaction of the other classes. This class should provide functions for users to be able to add a `Room`, delete a `Room`, print all `Rooms`, request a `Reservation`, and print all Reservations. As a bonus you can implement an `updateReservations` function which removes all `Reservations` that come before the current `Date`, i.e., the `Reservations` that are in the past.

5 Instructions

Download the starting code from Brightspace. It includes some global functions in `main.cc` that you are to use for testing, as well as the `Date` class and `defs.h`.

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). ALL CLASSES MUST HAVE A PRINT FUNCTION. This prints metadata about the class like names or dates but not data contained in a data structure like an array (unless explicitly specified).

You are allowed to pass `string` objects by value *for this assignment only*. Passing by `string&` does not work with string literals (and we will learn why in the lessons on **Encapsulation**). You may use `string` or `const string&`. For the purposes of this assignment they function the same, but the second version is more efficient, as we will learn.

Your finished code should compile into a single executable called `a1` using the command `make a1`, `make`, or `make all`, using a Makefile that you wrote yourself. Your submission should consist of a single zip file with a suitable name (e.g., `assignment1.zip`) that contains a folder containing all your files. This folder should also contain a README with your name, student number, a list of all files that are included, a directory structure (if applicable), compiling and running instructions, and any other information that will make the TAs life easier when they mark your assignment.

5.1 Notes on Automated Marking

Initially `main.cc` will not compile because it references classes and functions that do not exist. Feel free to comment out parts that are not working (the tests for the bonus section are already commented out - if you complete the bonus you should uncomment them). You may also `#include` additional header files for classes other than `Hotel`. You may make your own file with your own `main` function and write your own tests, and compile that separately. But the code in `main.cc` is what will be run to determine your mark.

Likely most of the bugs you experience will be in your own code. However, you might find bugs or inconsistencies in this test code as well. Please bring them to my attention.

5.2 The defs.h file

This file contains some preprocessor constants that you may use to initialize your arrays. You do not have to use these - you may also define your own constants if you wish, as long as the values are appropriate.

5.3 The Date Class

We will use the `Date` class that we saw in class but we will add two functions to allow us to compare and sort by `Date`, and a third function to advance the `Date` by a number of days.

1. Make a `bool equals(Date& d)` function. This should return `true` if the `Date` in question and `d` are equal and `false` otherwise.
2. Make a `bool lessThan(Date& d)` function. This should return `true` if the `Date` in question comes before `d` and `false` otherwise.
3. Make an `addDays(int days)` function that advances this `Date` by the number of days in the `days` parameter. You may call on the `incDate()` function if you wish.

5.4 The Reservation Class

`Reservations` will be stored in the `Room` class. It is essentially a *wrapper* class for the `Date` class that adds extra information and functionality.

1. Data members: choose appropriate data members for the `Reservation` object. You may use whatever you wish as long as the `Reservation` object functions correctly. Don't overthink it. There are probably two valid ways to represent a `Reservation` object, and they both are pretty obvious.

2. Make a 3 argument constructor with parameters `string customerName, Date& checkIn, int duration` in that order. Use these parameters to initialize your `Reservation` object.

3. Make a void `setDuration(int duration)` function. The duration of someone's stay is always at least 1 day.

4. Make an `bool overlaps(Reservation& r)` function. This function returns true if the two `Reservations` have a day in common. Note: a Reservation may end on the same day another Reservation begins. Examples:

   (a) A `Reservation r1` with `Date(2023, 1, 3)` and `duration 2` does NOT overlap with a `Reservation r2` with `Date(2023, 1, 7)` and any `duration`, since `r1` ends before `r2` begins.

   (b) A `Reservation r1` with `Date(2023, 1, 5)` and `duration 2` does NOT overlap with a `Reservation r2` with `Date(2023, 1, 7)` and any `duration`, since `r1` ends on the same day that `r2` begins.

   (c) A `Reservation r1` with `Date(2023, 1, 5)` and `duration 3` DOES overlap with a `Reservation r2` with `Date(2023, 1, 7)` and any `duration`, since `r1` begins before `r2` begins and ends after `r2` begins (which means that two different guests will be sleeping in the same room on the same night).

5. Make a `lessThan(Reservation& res)` function. If the `Reservations` overlap return `false`. Otherwise return `true` if `this Reservation` occurs before the `res Reservation` and `false` otherwise. As before, if `res` starts on the same day that `this Reservation` ends you should return `true`.

6. Make a `lessThan(Date& d)` function. This function should return `true` if this `Reservation` occurs before `d` (where the last day of this `Reservation` could be equal to `d` but not greater than).

7. Make a `print` function. This function should print all the `Reservation` data nicely formatted. For example:

   Reservation for Bobby on December 25, 2022 for 3 days.

5.5 The Room Class

Make a `Room` class.

1. Make six member variables:

   (a) `roomNumber`: an `int`

   (b) `bedType`: a `string`

   (c) `capacity`: an `int`

   (d) `hasFridge`: a `bool`

   (e) A *statically allocated* array of `Reservation` pointers. The `Room` class is responsible for deleting any dynamic memory contained in this array.

   (f) An `int` that tracks how many `Reservations` are currently in the array.

2. Make a four-argument constructor that takes an `int`, a `string`, an `int` and a `bool` as arguments. The first int should be used to initialize `roomNumber`, the `string` should used to intialize `bedType`, the second `int` should be used to initialize the `capacity`, and the `bool` should initialize `hasFridge`. This constructor should also appropriately initialize any other member variables.

3. (Bonus) Make a destructor that deletes all dynamic memory contained in this `Room`.

4. Make a getter for the `roomNumber` member (`getRoomNumber()`).

5. Make a function `bool isMatch(string bt, int cap, bool f)`. This function verifies that the `Room` in question has the right criteria to serve the client's needs. To return `true` the following is required:
   (a) The `bedType` must match `bt` (that is, they are equal)
   (b) The `capacity` is at least `cap`.
   (c) If `f` is `true` then a fridge is required, so `hasFridge` must also be `true`. If `f` is `false` then `hasFridge` may be `true` or `false`. That is, we don't mind if the room has a fridge even if we did not request it, but if we requested a fridge then the `Room` must have it.
6. Make a function `bool lessThan(Room& r)`. Return `true` if this `Room`'s number comes before `r`'s number in numerical order, otherwise return `false`.
7. Make a function `bool addReservation(string customerName, Date& d, int duration)`. If the array of `Reservation*` is full, return `false`. Otherwise add a new `Reservation` with the given parameters *only if* it does not overlap with another `Reservation`. For full marks add the `Reservation` in the order defined by the `Reservation::lessThan` function. Return `true` if a new `Reservation` is added, and `false` otherwise. Be careful to not introduce any memory leaks.
8. Make a `printReservations()` function (this is in addition to the regular `print` function). This function should print out the `Room` metadata, then print every `Reservation` of that `Room`.

5.6 The Hotel Class

The `Hotel` class will maintain a collection of `Rooms` and functions related to making, adding, and deleting `Rooms` and `Reservations`.

1. Member variables: There is one collection in the `Hotel` class, implemented as an array.
   (a) Make a *statically allocated array* of `Room` *pointers*. The `Hotel` class is responsible for deleting any dynamic memory contained in this array.
   (b) Make sure you track the number of elements in the above array.
2. Make a default (no-argument) constructor that initializes all member variables.
3. (Bonus) Make a destructor that deletes all dynamic memory contained in `Hotel`.
4. Make a `bool addRoom(int roomNumber, string bedType, int capacity, bool fr)` function to add a `Room` to `Hotel`. If there is no space in the array return `false`. If the room number is duplicated return `false`. Otherwise make a new `Room` object using the given parameters and add it to the array and return `true`. `Rooms` should be stored in increasing order as defined by their `lessThan` function. However, for part marks the `Room` may be added to the **back** of the array.
5. Make a `bool deleteRoom(int roomNumber)` function. If there is a `Room` with the given number in the array, remove it and close any gaps in the array.
6. Make a `bool getRoom(int roomNumber, Room** room)` function. If a `Room` with the given `roomNumber` exists, assign it to the `room` output parameter and return `true`. If there is currently no `Room` with that number, return `false`.
7. Make a `bool addReservation(string customer, string bedType, int capacity, bool fr, Date& date, int duration)` function. This function should find a `Room` that matches the given criteria and add a new `Reservation` to it. If there is a `Room` that matches the criteria *and* you successfully add the `Reservation` to that `Room`, return `true`, otherwise return `false`.

8. (Bonus) Make a `void updateReservations(Date& currentDate)` function. For each `Room` in the `Hotel` remove every `Reservation` that comes before the given `Date` (according to the `Reservation::lessThan` function). To complete this task you should also add an `void updateReservations(Date& currentDate)` function to the `Room` class that updates the `Reservations` for that `Room`.

9. Have the `print` function call `print` on all the `Rooms`.

10. Make a `void printReservations()` function. This should print out every `Room` and its `Reservations`.

5.7 The `testResOverlap` Function

1. In the `main.cc` file there is a function `resOverlap(Reservation& r1, Reservation& r2, bool shouldOverlap)` for testing the `overlaps` function of the `Reservation` class. You should finish the `int testResOverlap` function by writing at least 4 additional tests. The first test is given to you and explained below.

   To use `resOverlap`, supply two `Reservations` and a boolean indicating whether the `Reservations` should overlap. For instance, the first test, which is given to you, is as follows:

   ```
   Date d1(2022,9,30);
   Date d2(2022,9,28);
   Reservation res1("Joe", d1, 3);
   Reservation res2("Bob", d2, 2);
   if (!resOverlap(r1,r2,false)) score = 0;
   ```

   Since `res1` ends on the same day as `res2` begins they should not overlap, hence we supply `false` to the boolean `shouldOverlap` parameter. The `resOverlap` function then returns `true` if the test succeeds or `false` if the test fails. If `resOverlap` returns false you should set `score = 0`.

   You should come up with at least 4 tests to add to the `testResOverlap` function, and each of these should test for something different. Be sure to comment each test describing what it is testing for.

6 Grading

The marks are divided into three main categories. The first two categories, **Requirements** and **Constraints** are worth 26 and 8 marks respectively. The third category, **Deductions** is where you are penalized marks.

6.1 Specification Requirements

These are marks for having a working application (even when not implemented according to the specification, within reason). These are the same marks shown in the test suite, repeated here for convenience. Marks are awarded for the application working as requested.

The test script provides a mark out of 22. If you have implemented everything correctly, this will *probably* be your mark for these sections. However, you are still responsible for, and may be penalized for, any errors the test suite does not catch, or any drastic departure from the specification (such as using outside libraries). We reserve the right to modify the mark given by the test script in these cases.

General Requirements

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

Application Requirements: 22 marks + 1 bonus

- 2 marks: `Reservation overlaps` function works correctly.
- 2 marks: `Room isMatch` function works correctly.
- 2 marks: Can add a `Room` to the `Hotel` application.
- 2 marks: `Rooms` are added to the `Hotel` application in numerical order.
- 2 marks: Can delete a `Room` from the `Hotel` application.
- 4 marks: Can add a `Reservation` to a `Room`.
- 2 marks: `Reservations` are added to `Rooms` in the correct order (in order by `Date`).
- 6 marks: Can add a `Reservation` to the `Hotel` application.
- Bonus, 1 mark: Can update the `Reservations`.

Testing Requirements: 4 marks

- 4 marks: Proper implementation of at least four unique **tests** in the `testResOverlap` function with proper commenting.

Requirements Total: 26 marks + 1 bonus

6.2 Constraints

The previous section awards marks if your program works correctly. In this section marks are awarded if your program is written according to the specification and using proper object oriented programming techniques. This includes but is not limited to:

- Proper declaration of member variables (correct type, naming conventions, etc).
- Proper instantiation and initialization of member variables (statically or dynamically).
- Proper instantiation and initialization of objects (statically or dynamically).
- Proper constructor and function signatures.
- Proper constructor and function implementation.
- Proper use of arrays and data structures.
- Passing objects by *reference* or by *pointer*. Do not pass by value.
  - **Note:** For this assignment only, `strings` may be passed by value.
- Reusing existing functions, within reason.
- Proper error checking - check array bounds, data in the correct range, etc.
- Reasonable documentation (remember the best documentation is expressive variable and function names, and clear purposes for each class).
- **Bonus (1 mark):** All dynamically allocated memory should be deallocated. Use `valgrind` to ensure there are no memory leaks.

6.2.1 Constraint marks:

1. 2 marks: Proper implementation of the `Date` class.
2. 2 marks: Proper implementation of the `Room` class.
3. 2 marks: Proper implementation of the `Reservation` class.
4. 2 marks: Proper implementation of the `Hotel` class.
5. 1 mark (bonus): no memory leaks.

Constraints Total: 8 marks (+1 bonus)

6.3 Deductions

The requirements listed here represent possible deductions from your assignment total. In addition to the constraints listed in the specification, these are global level constraints that you must observe. For example, you may only use approved libraries, and your programming environment must be properly configured to be compatible with the virtual machine. This is not a comprehensive list. Any requirement specified during class but not listed here must also be observed and may be penalized if done incorrectly.

6.3.1 Packaging and file errors:

1. 5%: Missing README
2. 10%: Missing Makefile (assuming this is a simple fix, otherwise see 4 or 5).
3. up to 10%: Failure to use proper file structure (separate header and source files for example), but your program still compiles and runs
4. up to 50%: Failure to use proper file structure (such as case-sensitive files and/or Makefile instructions) that results in program not compiling, but is fixable by a TA using reasonable effort.
5. up to 100%: Failure to use proper file structure or other problems that severely compromise the ability to compile and run your program.

As an example, submitting Windows C++ code and Makefile that is not compatible with the Linux VM would fall under 4 or 5 depending on whether a reasonable effort could get it running.

6.3.2 Incorrect object-oriented programming techniques:

- Up to 10%: Substituting C functions where C++ functions exist (e.g. don't use `printf`, do use `cout`).
- Up to 25%: Using smart pointers.
- Up to 25%: Using global functions or global variables other than the `main` function and those functions and variables expressly permitted or provided for initialization and testing purposes.

6.3.3 Unapproved libraries:

- Up to 100%: The code 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.
- Up to 100%: Your program must not use any classes, containers, or algorithms from the standard template library (STL) *unless expressly permitted.*

Assignment Total (Requirements and Constraints): 34 marks + 2 bonus