COMP 2404 A/C – Assignment #4

Due: Wednesday, March 24 at 11:59 pm

1. Goal

For this assignment, you will write a C++ program, in the VM provided for this course, that manages the reservations in a small hotel. The hotel will have multiple rooms of different types, including a suite, some premium rooms, and regular rooms. The hotel will also keep track of the multiple guests that stay at the hotel. The reservations will be managed by a separate control-like object, that serves as the reservation manager. When a new reservation is requested, the reservation manager will be responsible for determining which room of the requested type is available for the duration of the guest's stay.

2. Learning Outcomes

With this assignment, you will:

- apply the OO concepts of inheritance and polymorphism in C++
- implement a simplified version of the Observer design pattern
- practice drawing a UML class diagram

3. Instructions

3.1. Draw a UML class diagram

This program will contain a class hierarchy to implement a simplified Observer design pattern. Polymorphism will be used to implement different behaviours in the derived classes of this hierarchy.

The class hierarchy will relate together different *event recorders* that serve as the observers, and the reservation manager will be the subject. These event recorders will be objects that look for specific characteristics in each new reservation, and they will keep records of new reservations that meet these characteristics. These records can later be used offline by hotel staff to make special offers or give monetary incentives to special guests. Every time that a new reservation is created, all the event recorders will be notified. There will be one abstract recorder class, and three concrete recorder classes, derived from the base class: a guest recorder that flags regular guests who reserve a premium room, a stay recorder that flags guests who stay longer than a specific number of nights, and an upgrade recorder that flags any regular guest that spends more than a specified limit.

You will begin by drawing a UML class diagram, using a drawing package of your choice, and your diagram will follow the conventions established in the course material covered in section 2.3. A partial diagram of the program has been provided for you in Figure 1, as a starting point.

Your diagram will represent all the classes in your program that belong in a UML class diagram, including the control, view, and entity classes in the program, as well as the inheritance hierarchy of event recorder classes. Your diagram will show all attributes, all operations (including the parameters and their role such as in, out, inout), and all associations between classes, including directionality and multiplicity, where applicable. As always, do not show collection classes, as they must be implied by multiplicity, and do not show getters, setters, constructors, or destructors.

3.2. Download and understand the posted code

The `a4-posted.tar` file has been provided for you in *cuLearn*. This file contains the following:

3.2.1. the `View` class that your code must use for most communications with the end user

3.2.2. a skeleton `Control` class, with a data initialization member function that your code is required to call

3.2.3. the `RoomArray` class that holds a collection of the rooms in the hotel

3.2.4. the `defs.h` header file that contains a required enumerated data type

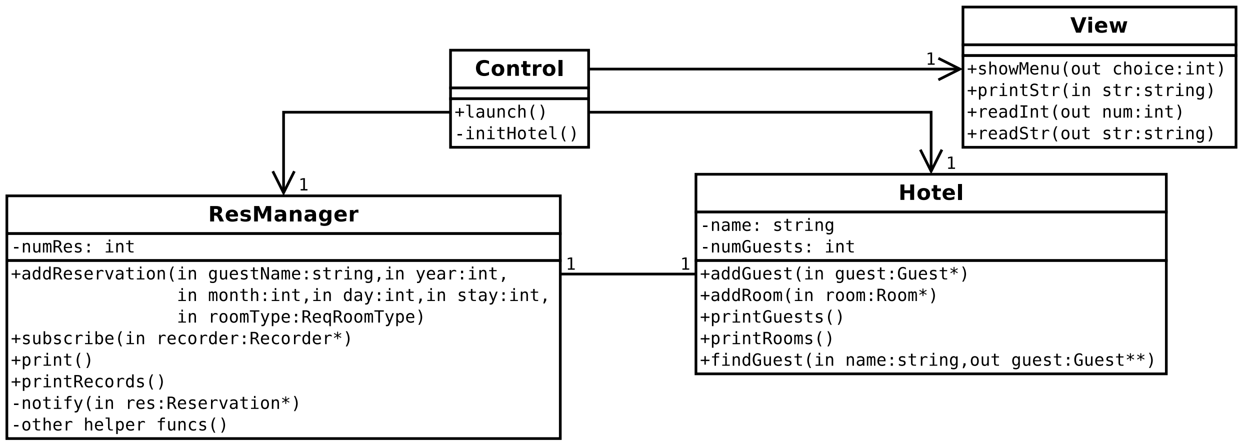
View
+showMenu(out choice:int)
+printStr(in str:string)
+readInt(out num:int)
+readStr(out str:string)
Control
+launch()
-initHotel()
ResManager
-numRes: int
+addReservation(in guestName:string,in year:int, in month:int,in day:int,in stay:int, in roomType:ReqRoomType)
+subscribe(in recorder:Recorder*)
+print()
+printRecords()
-notify(in res:Reservation*)
-other helper funcs()
Hotel
-name: string
-numGuests: int
+addGuest(in guest:Guest*)
+addRoom(in room:Room*)
+printGuests()
+printRooms()
+findGuest(in name:string,out guest:Guest**)
1
1
1
1
1

Figure 1: Partial UML class diagram

3.3. **Modify the `Date` class**

You will begin with the `Date` class that we worked on during lectures, in section 3.1, program #3, and you will make the following changes:

3.3.1. implement the `bool lessThan(Date&)` member function that compares two dates; one date is considered less than another if it occurs earlier

3.3.2. you may require the implementation of the `bool equals(Date&)` member function

3.3.3. remove the printing functions, and replace them with a `print()` member function that prints out the date using the format YYYY-MM-DD; for example, the due date for this assignment is 2021-03-24

3.3.4. implement a `void add(int duration)` member function that increases the date object (the `this` object) into the future by the number of days indicated by the `duration` parameter

(a) you must take into account the different number of days in each month, and the fact that dates may roll into the next month

(b) you must take into account the fact that dates in December may roll into a new year

(c) for simplicity, we will disallow `duration` values that exceed 31, so that the modified date will never be more than one month in the future

3.4. **Implement the `Guest` class**

You will create a new `Guest` class that contains the information for a guest of the hotel. The class will contain the following:

3.4.1. a data member that represents the guest's name, stored as a string

3.4.2. a data member that represents whether or not the guest is a *premium* guest, stored as a boolean; premium guests have special (offline) privileges in the hotel that the regular guests do not

3.4.3. a data member that represents the current number of loyalty points accrued by the guest, stored as an integer; when a guest first signs up at the hotel, they begin with zero loyalty points, and new points are added every time the guest makes a reservation

3.4.4. a default constructor that takes parameters for the name and premium indicator, and initializes all the data members

3.4.5. getter member functions for the name and premium indicator

3.4.6. an `addPts(int)` member function that adds the given amount to the guest's loyalty points

3.4.7. a `print()` member function that prints to the screen all the data members, correctly formatted

3.5. **Implement the `Room` class**

You will create a new `Room` class that contains the information for a room in the hotel. The class will contain the following:

3.5.1. a data member that represents the type of room (suite, premium, or regular), stored as a value of the provided `ReqRoomType` enumerated data type

3.5.2. a data member that represents the room number, stored as an integer

3.5.3. a data member that represents the daily rate of the room (how much it costs), stored as a `float`

3.5.4. a default constructor that takes parameters for all the data members, and initializes them

3.5.5. getter member functions for all data members

3.5.6. a `print()` member function that prints to the screen all the data members, correctly formatted

3.5.7. a `void computePoints(int& pts)` member function that computes the daily number of loyalty points earned for reserving the room, as follows:

- (a) for a suite, compute 20% of the room's daily rate, and return this value using the output parameter `pts`
- (b) for a premium room, compute 15% of the room's daily rate, and return this value using the output parameter `pts`
- (c) for a regular room, compute 10% of the room's daily rate, and return this value using the output parameter `pts`

3.6. Implement the `Hotel` class

You will create a new `Hotel` class that contains the all information for the hotel. The class will contain the following data members:

3.6.1. the hotel name, stored as a string

3.6.2. a collection of the guests registered with the hotel, stored as a statically allocated, primitive array of `Guest` pointers; as always when we work with primitive arrays, we must track the current number of elements in the array

3.6.3. a collection of the rooms in the hotel, stored as a `RoomArray` object

3.6.4. the reservation manager that will handle all the reservations for the hotel, stored as a `ResManager` pointer; this class is described in a later step

NOTE: The hotel and reservation manager objects will have a *bidirectional* association between them. This means that they will each contain a pointer to the other. One goal of this assignment is to show you how to set up a bidirectional association in the C++ programming language.

The class will contain the following member functions:

3.6.5. a default constructor that takes parameters for the hotel name and the reservation manager, and initializes all the data members

3.6.6. a destructor that deallocates the required dynamically allocated objects

3.6.7. a getter member function for the rooms collection

3.6.8. an `addGuest(Guest*)` member function that adds the given guest to back of the guest collection

3.6.9. an `addRoom(Room*)` member function that adds the given room to the back of the room collection

3.6.10. a `printGuests()` member function that prints to the screen all the guests in the hotel

3.6.11. a `printRooms()` member function that prints to the screen all the rooms in the hotel

3.6.12. a `bool findGuest(string n, Guest** g)` member function that searches the guest collection to find the guest with the name `n`, and returns this guest using the `g` parameter

3.7. Implement the `Reservation` class

You will create a new `Reservation` class that contains the information for a reservation in the hotel, including the guest making the reservation, and the room assigned.

The `Reservation` class will contain the following data members:

3.7.1. the guest who makes the reservation, stored as a `Guest` pointer

3.7.2. the arrival date of the reservation, stored as a `Date` pointer; this represents the date when the guest is arriving, and when their hotel stay begins

3.7.3. the number of days that the guest is staying, stored as an integer; this assignment specification will refer to this data member as the `stay`; you can give it a better name, if you prefer

3.7.4. the room that gets assigned to the guest for the duration of their stay, stored as a `Room` pointer

3.7.5. the total charge for the reservation, stored as a `float`; this is the dollar amount that will be charged to the guest for their stay, as the product of the room's daily rate and the number of days

The `Reservation` class will contain the following member functions:

3.7.6. a default constructor that does the following:
  (a) take parameters for the guest, the room, the arrival date, and the stay (the number of days that the guest is staying), and initialize the corresponding data members
  (b) compute the total charge for the reservation, and initialize the corresponding data member

3.7.7. a destructor that deallocates the required dynamically allocated objects

3.7.8. getter member functions for most data members, as required

3.7.9. a `bool lessThan(Reservation*)` member function that compares two reservations; one reservation is considered lesser than another if its arrival date is earlier

3.7.10. a `print()` member function that prints to the screen all the data members, including the guest's name and the room number

3.8. Implement the `Recorder` classes

You will create an inheritance hierarchy of `Recorder` classes that represent the different kinds of event recorders that will keep record of certain events, where reservations are created that meet specific criteria. These event recorders will serve as the observers in this program.

The `Recorder` class will be the base class. It will be an abstract class, and it will contain the following:

3.8.1. a data member that represents the recorder object's name, stored as a string

3.8.2. a data member that stores the records collection, stored as a STL `vector` of strings

3.8.3. a default constructor that take a name as parameter, and initializes it

3.8.4. a `printRecords()` member function that prints out the event recorder's name, loops over its records collection, and prints each record to the screen

3.8.5. a pure virtual `void update(Reservation*)` member function that contains no implementation in this class, but that will be overridden by all derived classes

The `StayRecorder` class will be a concrete class, derived from the `Recorder` base class. It will contain:

3.8.6. a default constructor that takes a recorder name as parameter, and uses *base class initializer syntax* to initialize it

3.8.7. an implementation of the overridden `update(Reservation*)` member function that checks if the duration of the reservation (its stay) is longer than 3 days; if so, it formats an informative string with the guest's name and the duration of the stay, and adds this string to its records collection

The `GuestRecorder` class will be a concrete class, derived from the `Recorder` base class. It will contain:

3.8.8. a default constructor that takes a recorder name as parameter, and uses base class initializer syntax to initialize it

3.8.9. an implementation of the overridden `update(Reservation*)` member function that checks if the reservation is for a regular guest (not a premium one), staying in a premium room or a suite; if so, it formats an informative string with the guest's name, and adds this string to its records collection

The `UpgradeRecorder` class will be a concrete class, derived from the `Recorder` base class. It will contain:

3.8.10. a default constructor that takes a recorder name as parameter, and uses base class initializer syntax to initialize it

3.8.11. an implementation of the overridden `update(Reservation*)` member function that checks if the reservation is for a regular guest (not a premium one), who will be spending more than $1500 with this reservation; if so, it formats an informative string with the guest's name and and the total amount charged, and adds this string to its records collection

NOTE #1: The `update(Reservation*)` member functions must be implemented as *polymorphic functions* using dynamic binding. Do not dumb this down with an `if-else` control structure.

NOTE #2: For the recorder classes only, you may group together the class definitions in a single header file, and the member function implementations in a single source file.

3.9. Implement the `ResManager` class

You will create a new `ResManager` class that creates and manages all the reservations for the hotel. The class will contain the following data members:

3.9.1. the hotel whose reservations are managed, stored as a `Hotel` pointer

3.9.2. a collection of the reservations for the hotel, stored as a statically allocated, primitive array of `Reservation` pointers; as always when we work with primitive arrays, we must track the current number of elements in the array

3.9.3. a collection of event recorders (the observers) that will be notified every time a new reservation is created; this collection will be a STL `vector` of `Recorder` pointers

The class will contain the following member functions:

3.9.4. a default constructor that takes a parameter for the hotel, and initializes all the data members

3.9.5. a destructor that deallocates the required dynamically allocated objects

3.9.6. a `setHotel(Hotel*)` member function that sets the hotel data member to the given parameter

3.9.7. an `addReservation(string name, int yr, int mth, int day, int stay, ReqRoomType req)` member function that does the following:

- (a) validate the `stay` parameter so that it is a positive number less than 31; if the stay parameter is not valid, the reservation cannot be created
- (b) dynamically allocate a new `Date` object using the given day, month, and year parameters, and store the returned pointer in a temporary variable
- (c) loop over the hotel's rooms collection, and find the first room that fits all of the following criteria:
  - (i) the room must be of the type specified in the `req` parameter
  - (ii) the room must be available for every day starting at the arrival date, for the duration of the stay indicated in the `stay` parameter
    NOTE: A room is available if there is no reservation for that same room that overlaps with this one, for any given day.
- (d) if there is no room available for the duration of the stay, the reservation cannot be created
- (e) find the hotel guest with the name matching the `name` parameter; if the guest is not found, the reservation cannot be created
- (f) dynamically allocate a new `Reservation` object with the found guest, the available room, the arrival date, and the `stay` parameter
- (g) add the new reservation to the reservation manager's collection, in its correct place, so that the collection remains sorted in ascending order
  - (i) you must **shift** the elements in the array towards the back of the array to make room for the new element in its correct place; **do not** add to the end of the array and sort, as this is both inefficient and unnecessary
  - (ii) you must use the `Reservation` class's `lessThan()` function to perform the comparison
- (h) compute the number of points that are accrued for the available room on a daily basis
- (i) add to the guest's loyalty points the product of the daily points and the stay (number of days)
- (j) notify all the event recorders of the new reservation

3.9.8. a `subscribe(Recorder*)` member function that adds the given recorder to the event recorder collection

3.9.9. a `print()` member function that prints to the screen all the reservations in the collection

3.9.10. a `printRecords()` member function that loops over the collection of event recorders, and prints to the screen all the records stored by each event recorder

3.9.11. a private `notify(Reservation*)` member function that loops over the collection of event recorders, and notifies each one that the given reservation has been newly created, using their `update()` member function

3.9.12. other private helper member functions that are required for good design

3.10. **Implement the `Control` class**

You will implement the `Control` class as follows:

3.10.1. the data members will be as shown in the partial UML class diagram

3.10.2. the default constructor will do the following:

- (a) dynamically allocate a reservation manager
- (b) dynamically allocate a hotel with a specific name and the reservation manager as parameters
- (c) set the reservation manager's hotel data member to the newly created hotel
- (d) dynamically allocate three event recorders, as follows:
  - (i) create one event recorder of each type: `StayRecorder`, `GuestRecorder`, `UpgradeRecorder`, giving each one a name that reflects its type
  - (ii) subscribe each event recorder as an observer to the reservation manager

3.10.3. the destructor will clean up the necessary memory

3.10.4. the `launch()` member function will do the following:

- (a) call the initialization function that has been provided in the `a4-posted.tar` file; you must use the provided function, *without modification*, to initialize the data in your program
- (b) use the `View` object to display the main menu and read the user's selection, until the user exits
- (c) if required by the user:
  - (i) print out all the reservations stored by the reservation manager
  - (ii) print out all the rooms in the hotel
  - (iii) print out all the guests in the hotel
  - (iv) print out all the records stored by the event recorders, by calling a member function on the reservation manager

3.11. **Write the `main()` function**

Your `main()` function must declare a `Control` object and call its `launch()` function. The entire program control flow must be implemented in the `Control` object as described in the previous instruction, and the `main()` function must do nothing else.

4. Constraints

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

4.1. The code must be written in C++98. It must compile and execute in the default course VM, without any libraries or packages or any software that's not already provided in the VM.

4.2. Your program must comply with the principle of least privilege, and it must follow correct encapsulation principles, including the separation of control, view, entity, and collection object functionality.

4.3. Your program must not use any classes, containers, or algorithms from the C++ standard template library (STL), unless explicitly permitted in the instructions.

4.4. Your program must follow basic OO programming conventions, including the following:

4.4.1. Do not use any global variables or any global functions other than `main()`.

4.4.2. Do not use `struct`s. You must use classes instead.

4.4.3. Objects must always be passed by reference, never by value.

4.4.4. Functions must return data using parameters, not using return values, except for getter functions.

4.4.5. Existing functions must be reused everywhere possible.

4.4.6. All basic error checking must be performed.

4.4.7. All dynamically allocated memory must be explicitly deallocated.

4.5. All classes must be thoroughly documented in every class definition, as indicated in the course material, section 1.3. **DO NOT** place inline comments in your function implementations.

5. Submission

5.1. You will submit in *cuLearn*, before the due date and time, the following:

5.1.1. A UML class diagram (as a PDF file), drawn by you using a drawing package of your choice, that corresponds to the entire program design.

5.1.2. One `tar` or `zip` file that includes:

(a) all source and header files

(b) a Makefile

(c) a README text file that includes:

(i) a preamble (program author, purpose, list of source and header files)

(ii) compilation and launching instructions

NOTE: Do not include object files, executables, hidden files, or duplicate files or directories in your submission.

5.2. Late submissions will be subject to the late penalty described in the course outline. No exceptions will be made, including for technical and/or connectivity issues. Do not wait until the last day to submit.

5.3. Only files uploaded into *cuLearn* will be graded. Submissions that contain incorrect, corrupt, or missing files will receive a grade of zero. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading

6.1. **Marking components:**

- 20 marks: UML class diagram
- 10 marks: correct implementation of `Date`, `Guest`, and `Room` classes
- 16 marks: correct implementation of `Recorder` classes
- 10 marks: correct implementation of `Hotel` class
- 8 marks: correct implementation of `Reservation` class
- 26 marks: correct implementation of `ResManager` class
- 10 marks: correct implementation of `Control` class

6.2. **Execution requirements:**

6.2.1. all marking components must be called and execute successfully in order to earn marks

6.2.2. all data handled must be printed to the screen for marking components to earn marks

6.3. **Deductions:**

6.3.1. Packaging errors:

(a) 10 marks for missing Makefile

(b) 5 marks for missing README

(c) up to 10 marks for failure to correctly separate code into header and source files

(d) up to 10 marks for bad style or missing documentation

6.3.2. Major design and programming errors:

(a) 50% of a marking component that uses global variables or `struct`s

(b) 50% of a marking component that consistently fails to use correct design principles

(c) 50% of a marking component that uses prohibited library classes or functions

(d) up to 100% of a marking component where Constraints listed are not followed

(e) up to 10 marks for bad style

(f) up to 10 marks for memory leaks and errors reported by `valgrind`

6.3.3. Execution errors: 100% of any marking component that cannot be tested because it doesn't compile or execute in the provided course VM, or the feature is not used in the code, or data cannot be printed to the screen, or insufficient datafill is provided.