**MAJOR PRACTICAL – HOTEL RESERVATION SYSTEM**

**Introduction**

This project will focus on the development of an online hotel reservation system for guests. When the guest accesses the online system, he/she can choose to login as an existing customer or create a new account. When creating a new account, the system will store the user’s login details so that the user does not have to create a new account every time they access the program. Once authenticated, the guest has two options: search for rooms, or cancel a previous booking.

If the guest chooses to search for a specific type of room to book, a list of available room numbers will be displayed. If no rooms are available, the system will alert the guest and the guest has an option to search again for another room type. The guest selects the room number they want to book, and a summary of the booking will be generated with the user’s information already retrieved and a unique booking ID generated. The guest will be directed to the payments portal where they can choose to pay by credit/debit card, PayPal or BPAY. Each payment method will have different methods of calculating the cost due to the surcharges. An invoice of their payment will be displayed, and the guest has an option to make another booking. If yes, they get directed to the booking screen; otherwise, the guest is redirected back to the main menu (logged out of account).

If the guest chooses to cancel a booking, the guest enters in their booking ID and the system alerts whether it was a successful cancellation, along with the refund value. That specific room will then be made vacant to other guests.

**Design Description**

*Assessment Concepts – Brief outline*

**Memory allocation from the stack and heap**

* Arrays:
  + Creating list of rooms in hotel – heap
  + List of booked rooms – heap
* Strings: (mainly from the stack)
  + Details of guests
  + room type and status
* Objects
  + Hotel, Guest, Login, Room, Payments

**User Input and Output**

* I/O of different data types:
  + Output list of available rooms for guests – string array
  + Guests input type of room to book – string
  + Guests select menu options and room number – int
  + Output of booking and payment summary – string, int, float

**Object-oriented programming and design**

* Inheritance:
  + Different room types (single, double, grand) are subclasses of Rooms, where each will have a fixed but different capacity.
  + There will also be a Payments base class with subclasses credit/debt payment, PayPal and Bpay
* Polymorphism
  + The Payment class will contain a virtual setAmount() function, which, in the base class, will return the total cost of the room. However, the derived classes PayPal and Bpay will return different amounts as these classes charge an additional fee
* Abstract classes
  + Rooms will be an abstract class with pure virtual method setTotalCost() as each room type will have different total costs due to the room type and the addition of extras, and the cost of a generic ‘room’ cannot be calculated unless the room type is specified

*Class Diagrams*

Table

Description automatically generated

Diagram

Description automatically generatedDiagram

Description automatically generated

*User Interaction*

Users will use the command-line to navigate different options. Menu interfaces will number the different options the user can select, and users must enter the number corresponding to their preference. Users will be asked when to enter their input, and the type of input that is required (e.g. yes/no, number, etc.). After an input is accepted, any output will always be displayed on a new line. An example interface is shown below:

Welcome!

Select one of the following options:

1. Login

2. Register

Your option (1 or 2):

*Code Style*

**Indentation**

Logical blocks of code, especially those involving nested loops, will be indented with a ‘tab’ (equivalent to 4 spaces).

**Comments**

Comments will mainly be written to summarise the operation of a block of code and placed on the line before the code block is executed. Similarly, comments will describe the operation of certain functions belonging to a class and what variable type those functions may return. Side comments may also be provided for parts of code that contain complex features to assist in understanding how the code is executed.

**Naming Conventions**

All variables, including function names, will be written in lower camel case, where a variable name containing multiple words will be written without spaces, the first word written in lowercase, and the proceeding words capitalised (eg. setAmount, getStatus, etc.).

Class names, on the other hand, will start with a capital letter. Each different class will have a different header and .cpp file, which will be named after that specific class. For classes containing inheritance relationships, the base and the derived classes will be placed into one header file named after the base class. This helps to limit the number of different header files and cpp files that need to be created and helps to visualise the inheritance hierarchy.

The main file where the whole, polished code is to be executed will be placed under a main.cpp, and any relevant functions not relating to the created classes will be placed in functions.cpp.

**Testing Plan**

*Unit Testing*

Every member function of a class will be tested in a cpp file labelled [ClassName]-main.cpp. The Makefile will consist of different targets corresponding to the class to be tested, with each target labelled after the name of the class. The cpp files for each class will be tested by running the Makefile command *make* [classname]. Every constructor and destructor will be tested, either by displaying the values of the member variables, or by displaying “[class] constructor called” or “[class] destructor called” to ensure that the constructors and destructors have been invoked by the program.

*Integration Testing*

The first few stages of development will involve the construction of separate classes and testing their methods in isolation to the other classes. When integrating all classes together to form a working program, the integration testing will be placed into a main-test-1.cpp file, which will be built gradually and sequentially.

Once the main-test file has been completed, any repetitive blocks of code will be grouped into functions or multiple if-else statements can be replaced with more efficient loops. In this refactoring stage, the new functions will be placed into a functions.cpp and these functions are then tested with their parameters and their return value in another main-test-2.cpp to ensure that the code still works as expected with the new functions.

*Input/Output Testing*

When the user is required to select options from a menu to perform their desired operation, the only input that can be accepted by the program is an integer from the range displayed on the screen. Any other value, or any other input that is not an integer, should not allow the user to proceed and warn the user of their invalid input, allowing them to keep entering their input until it is accepted. Some test cases to consider for this are negative numbers, numbers outside the range, special characters, spaces.

For parts of the program accepting string inputs, the program must consider case-sensitivity. In such cases, the program will only accept inputs spelled as specified, which can be written in lowercase or capitalising the first letter. Other inputs would not be accepted and should warn the user when this happens. The program would keep asking for input until the input is in an accepted form.

Some classes of the program will have specific methods that accept other types of user input, such as the Card class derived from Payment. This class would require the user to input a credit card number, which always contain 16 digits. The program must validate the user’s input in two parts: if the input has length 16, and if it is all in digits.

Despite the type of input, the method of validation is generally the same: identify acceptable input, check user’s input against the expected inputs, alert and continue asking user if input is invalid.

*Automated Testing*

A Makefile will be constructed, containing multiple targets corresponding to different classes as their methods are individually tested. Each of these targets would have [ClassName].cpp [ClassName]-main.cpp as their dependencies so if the code is changed in any of these files, the new build is run and executed. For functions that take in a user’s input, a small collection of tests will be made by creating input and output files to compare whether the output of that function matches the expected output. A test file labelled *test[functionname]* will be run using a bash script.

*Regression Testing*

**Weekly Schedule**

*MILESTONES:*

* Finalised design document – by end of Week 9
* Complete creating all header and cpp files for each class – by the end of Week 9
* Complete testing of all class methods – by the end of Week 9/middle of Week 10
* Integrate all classes – end of week 10
* Complete testing of all classes – end of week 10
* Refactoring of code – before assessment Week 11

*Week 8*

* Rough outline of project
* Create SVN directory
* Start working on the design document
  + Brief description of project
  + Link parts of project to the assessment concepts
  + Create class diagrams
  + Plan out weekly schedule

*Break Week 1*

* Finalise testing plan
* Program the Login class (incl. .h and .cpp files)
  + Unit testing
* Program the Guest class (incl. .h and .cpp files)
  + Unit testing

*Break Week 2*

* Program the Room class
  + Unit testing
* Program the Hotel class – half done

*Week 9*

* Program the Payment class
* Finish programming the Hotel class
* Integrate the guest and hotel class for testing
* Link all classes together in a main.cpp file for testing
* Submit code to SVN

*Week 10*

* Check testing of each class and their member functions
* Check testing of integration of classes
* Refactor code, if needed
* Prepare for week 10 practical marking
  + Identify 2 features to present – check Makefile
  + Submit code to SVN
  + Identify 2 test situations
* Finalise schedule for week 11
* Compare work against rubric

If time permits, could extend project to include a login from an Admin, which would involve extra functionality specific to the Admin only, such as checking in and checking out guests, viewing details of booked rooms. This would involve the use of polymorphism as the Guest and Admin would use different login methods (there would be different locations from which the program will retrieve and store logins from different user types).

*Week 11*

* Polish code, improve readability
* Check comments, code structure
* Revise project document to ensure the concepts have been met in the program
* Compare program against rubric again