CSM20 Final Group Project

This is the Design Document for the CSM20 Final Group Project for the right group. The goal of the assignment was to create an Airline Reservations System in C++.

Github repository: `https://github.com/radiofreq10/csm20final`

Design Strategies

We all worked together to create a design plan for the application. Strategies included simplifying tasks to specific classes, optimizing code size by relying on containers, and focusing on the requirements of the assignment. We started by outlining what classes we would need, including PassengerData.h, and FlightData.h classes. We then discussed separating the data from the interface by creating two classes: AirlineInterface.h would handle the program output and interface, and FlightManager.h would handle the creation, storage, and calculation of the data. We tried to focus on how an actual Airline Reservations System might work.

Notes on Compiler: When we initially compiled the application on Visual Studio we noticed a significant delay to display the main menu. This delay was not present when compiling with Xcode's LLVM compiler or GCC's g++ compiler. We quickly ruled out this being a machine specific issue and managed to track it down and found it was related to Visual Studio defaulting the solution configuration to debug mode causing extra debugging information to be generated and the optimizations to be turned off. Switching this option to release fixed this speed issue since the extra debug information was not included and the optimizations were turned on.

Credit

- Nicholas Clayton
  - FlightManager Class
  - SeatingQueue Class
- Benjamin Dahl
  - Airline Interface Class
- Jeff Grammer
  - FlightData Class
- Robbie Jordan
  - Input Files
- Alex Latouf

    - Airline Interface Class
- Beto Lopez
    - Graph Class and Flightmap Clas
- Jake Lyon
    - Passenger Data Class

Project Architecture

Input Files

We decided to use CSV files because comma delimited files are relatively easy to parse. It is also easy to create and modify CSV files in programs such as Excel. The columns are as shown below.

PassengerData.txt				
Reservation Number	First Name	Last Name	Flight Number	Seat Class Number

FlightData.txt				
Flight Number	Origin City	Destination City	Departure Time	Arrival Time

AirlineInterface.h

AirlineInterface.h is the main interface class for the program. Its responsibilities are to hold the FlightManager data member, and provide input/output interface for the program. displayMenu() is designed to be used in main inside a while loop. It returns false when the user wishes to quit.

The AirlineInterface main menu.

```
Welcome to CSM20 Airlines. Choose an option.
1. Display all passengers.
2. Display passengers going to a city.
3. Display passengers leaving a city.
4. Display passengers on a flight.
5. Display wait list for a flight.
6. Display global wait list.
7. Display all flights.
8. Display all flights including passengers.
9. Search for a passenger.
0. Search for a flight.
D. Delete a Passenger.
```

```
Q. Quit.
Enter your selection:
```

AirlineInterface.h also contains a couple nested classes including SearchForPassenger class, and SearchForFlight class. These classes are utility classes that have the operator() overloaded. They are used as a function object for the AVL Tree traversal functions.

Notes: It is important to recognize that the output of displaying passengers on a flight shows the original seatclass of the passenger's reservation rather than the seat class they are currently in. The list of passengers on a flight is in order of the seatmap. The first 3 seats are for Pilot Club, the next 5 seats are First Class, the next 5 seats are Business Class, and the remaining 27 seats are economy. If there was not room for a passenger in their seating class they were bounced into a lower seat rather than putting them on a waitlist. The bounce was in order of reservation number. Consideration was made to be sure that no more than 10 bounced seats occur on a given flight.

AirlineInterface.h
-data : FlightManager
+displayMenu() : boolean -displayAllPassengers() : boolean -displayPassengersTo(city : char) : boolean -displayPassengersFrom(city : char) : boolean -displayPassengersOnFlt(flight : size_t) : boolean -displayWaitFlight(flight : size_t) : boolean -displayWaitAll() : boolean -displayAllFlights() : boolean -displayAllFlightsPassengers() : boolean -searchPassenger(passenger : string) : boolean -searchFlight(flight : size_t) : boolean -getCharInput(field : string) : char -getNumericInput(field : string) : size_t -enterToContinue() : void

SearchForPassenger
-displayCount : int -firstNameKey : string -lastNameKey : string -reservationNumkey : size_t -flightNumKey : size_t -removePassenger : bool

-firstPass : bool -fmPtr : FlightManager* -markedForRemoval : PassengerData* = nullptr
+SearchForPassenger() +~SearchForPassenger() +setFirstNameKey(key : string) : void +setLastNameKey(key : string) : void +setReservationNumKey(key : size_t) : void +setFightNumKey(key : size_t) : void +setRemovePassengerKey(key : bool) : void +setfmPtr(key : FlightManager*) : void +clearKeys() : void +operator()(passenger : PassengerData) : void +removeMarkedPassenger() : bool

SearchForFlight
-displayCount : int -toCity : char -fromCity : char -flightNum : size_t -displayPassengers : bool -firstPass : bool
+SearchForFlight() +setToCity(key : char) : void +setFromCity(key : char) : void +setFlightNum(key : size_t) : void +setDisplayPassengers(key : bool) : void +clearKeys() : void +operator()(passenger : PassengerData) : void

FlightManager.h

FlightManager.h is the main data class. It is responsible for holding AVL trees of PassengerData objects and FlightData objects. AVL trees were chosen as the primary data container for ease of sorting and searching based on key values. It's constructor reads the input files and fills the trees. The constructor also calculates the distances between airports and calculates the seatmaps for each flight and the waitlist. FlightManager.h also provides traversal functions to the AirlineInterface.h class for the private data members. addPassenger() adds a passenger to passengerList and the seating vector on their corresponding flight. addFlight() adds a flight to flightList and calls calculateMileage using

the flight as the parameter. removePassenger() removes a passenger from all lists and their flight's corresponding seating vector. calculateMileage() calculates the mileage of a flight using the flightMap graph. populateMap() sets the vertices and edge length of flightMap.

FlightManager.h
-passengerList : AVLTree<PassengerData> -flightList : AVLTree<FlightData> -waitList : AVLTree<PassengerData> -flightMap : FlightMap -seatQueue : SeatingQueue
+FlightManager() +traversePassenger() : void +traverseFlight() : void +traverseWait() : void +addPassenger(passenger : const PassengerData&) : bool +addFlight(flight : FlightData&) : void +removePassenger(aPassenger : const PassengerData&) : PassengerData -calculateMileage(flight : FlightData&) : void -readFlightsFromFile(inputStream : ifstream&) : void -readPassengersFromFile(inputStream : ifstream&) : void -populateMap() : void

nextFromWaitList

nextFromWaitList is a private utility class within FlightManager. It is called as a function object parameter for an AVL tree traversal. It will determine the next highest membership passenger waitlisted for a given flight. This passenger is stored in dynamic memory pointed to by nextPassenger, and is returned by get(). The found() function returns a bool indicating whether a passenger for the specified flight was found in the AVL tree. get() returns *nextPassengers. found() returns nextPassenger != nullptr.

nextFromWaitlist
-flightNum : size_t -nextPassenger : PassengerData*
+nextFromWaitlist(flightNumber : const size_t&) +operator()(aPassenger : const PassengerData&) : void +get() : PassengerData +found() : bool

SeatingQueue.h

The SeatingQueue.h class is a functional class that extracts the PassengerData vector from a FlightData object, and enqueues the PassengerData objects in a priority queue checking for membership class. The underlying structure is a std::priority_queue; chosen for ease of sorting by clearly defined priority levels. Membership classes range from 1 to 4, with 1 being the highest class and 4 the lowest. empty() returns seatQueue.empty(). isFull() returns seatQueue.size() >= MAX_SEATS. clear() empties seatQueue and sets flightPtr = nullptr. finalizeSeating() inserts passengers from seatQueue into the seating vector. Handles bouncing lower class passengers and adding excess passengers to the waitlist.

SeatingQueue.h
-MAX_PILOT_CLASS : static const size_t = 3 -MAX_FIRST_CLASS : static const size_t = 5 -MAX_BUSI_CLASS : static const size_t = 5 -MAX_ECON_CLASS : static const size_t = 27 -MAX_BOUNCED : static const size_t = 10 -MAX_SEATS : static const size_t = 40 -PILOT_CLASS : static const size_t = 1 -FIRST_CLASS : static const size_t = 2 -BUSI_CLASS : static const size_t = 3 -seatQueue : std::priority_queue<PassengerData, std::vector<PassengerData>, compareMembership> -waitListPtr : AVLTress<PassengerData>* -flightPtr : FlightData* = nullptr
-queueCurrentPassengers() : void +SeatingQueue(waitList : AVLTree<PassengerData>&) +add(newPassenger : const PassengerData&) : bool +setFlight(aFlight : FlightData&) : void +empty() : bool +isFull() : bool +clear() : void +finalizeSeating() : void

compareMembership

compareMembership is a private utility class of SeatingQueue.h. It is used as a function object for use in an std::priority_queue to compare the membership values of the enqueued PassengerData objects. Low membership values have higher priority than higher values.

compareMembership
+operator()(passenger1 : const PassengerData&, passenger2 : const PassengerData&) : bool

PassengerData.h

PassengerData.h class is a data class that holds all the data fields associated with a Passenger.

The default constructor is written such that it will either validate the input via set functions, or set the data members to default values.

All set functions check the data being passed to them and throw PrecondViolatedExcep for invalid data entry. SetFirstName() and SetLastName() use a helper function, nameCheck(string), to validate name data.

All get functions return constant data. No validation is conducted, assumes data has been checked via set functions.

NameCheck(string) is the only helper function. It checks each character in the string using the function isalpha(char);

Overloaded comparison operators check the reservation numbers first. If both reservation numbers are set to default values, the rest of the data members are compared instead. If all data is default, the operator returns false.

Copy constructor and overloaded assignment operator originally made private to prevent implicit copies. Copies turned out to be extremely valuable in the implementation of other classes that use PassengerData, thus the copy constructor is back in.

Overloaded stream extraction operator (>>) inserts data in the order of reservation number, first name, last name, flight number, and membership class. It is assumed the class inputting the data knows this. Only one (string) buffer is used; an int buffer was considered, but the idea was dropped as getline() will not accept an int argument. Instead, the String function stol(string) was used to input numerical data. All input validation is done through the set functions.

Overloaded stream insertion operator (<<) outputs data in the order of Reservation number, first name, last name, membership class, and flight number. Here, the number used to represent membership is used in a switch/case to output a string stating the passenger's seating class.

PassengerData.h

-firstName : string
-lastName : string
-seatClass : size_t
-reservationNumber : size_t
-flightNum : size_t
-nameCheck(name : string) : bool

+PassengerData(reservation : size_t, first : string, last : string, seatClass : size_t, flight : size_t)
+setFirstName(first : string) : void
+setLastName(last : string) : void
+setMembership(seatClass : membership) : void
+setReservationNum(reservation : size_t) : void
+setFlightNum(flight : size_t) : void
+getFirstName() : string
+getLastName() : string
+getMembership() : size_t
+getReservationNum() : size_t
+getFlightNum() : size_t
+operator ==
+operator >
+operator

graph.h

For the *graph* class we went with a weighted, undirected and connected graph, since **1.** all cities had to be connected, **2.** there is a distance between cities, and **3.** it's possible for a flight to go both directions between cities. Furthermore, the *graph* class was implemented as an adjacency list. One of the advantages of an adjacency list implementation over an adjacency matrix is that an adjacency list is more efficient in finding all vertices adjacent to a given vertex. Which means that Dijkstra's algorithm will run more efficiently. The reason why is because Dijkstra's algorithm uses breath-first search to traverse all vertices in the graph, and BFS function by first visiting all vertices adjacent to a given vertex(which is the more efficient operation performed by an adjacency list). Since the majority of operations demanded from this class in this this particular application will come from Dijkstra's algorithm, implementing the graph as an adjacency list was the more efficient choice.

Of course, the main duty relegated to the *graph* class and our primary reason for choosing this class, was to represent our flight map. The vertices represent cities and the edges represent the distance(in miles) cost between the cities. for this particular flight map, it was possible to travel either direction between cities.

graph class is base class for *flightMap* class

Graph.h
-numVertices : int -numEdges : int -adjList : map<ItemType, map<ItemType, int>> -iter : map<ItemType, int&gt::const_iterator -iter2 : map<ItemType, map<ItemType, int>>::const_iterator
+getNumVertices() const : int +getNumEdges() const : int +getVertex(vertex : ItemType) : ItemType +getWeight(startVertex : ItemType, endVertex: ItemType) : int +add(startVertex : ItemType, endVertex : Itemtype, weight : int) : bool +remove(startVertex : ItemType, endVertex : ItemType) : bool +isEmpty() : bool +breadthFirstTraversal(startVertex : Itemtype, visit(vertex : Itemtype) : void) : void

flightMap.h & Dijkstra's algorithm

flightMap class was created to implement Dijkstra's algorithm. The goal was to maintain the data independence of the *graph* class and have the *flightMap* class inherit from *graph* class to expand on that needed functionality. We found that Dijkstra's algorithm was the best option for finding the shortest path between two cities (vertices), since the algorithm will find the shortest path between the source vertex and all other vertices. Furthermore, since only the shortest distance between the origin vertex and the destination vertex is desired, the algorithm was modified to terminate as soon as it found the destination vertex.

flightMap class inherits from *graph* class

flightMap.h
+shortestPath(startVertex : ItemType, endVertex : ItemType) : size_t

FlightData.h

`FlightData.h` handles the all the data that is associated with a flight.

FlightData.h
-MAX_PASSENGERS : static const size_t = 40 -flightNumber : size_t -departTime : size_t

-arriveTime : size_t
-bounced : size_t = 0
-mileage : size_t
-toCity : char
-fromCity : char
-seatMap : vector

+FlightData(fn : size_t = 0, miles : size_t = 0, departTime : size_t = 0, arriveTime : size_t = 0, toc : char = '0', frc : char = '0')
+operator==(right : const FlightData&) : bool
+operator>(right : const FlightData&) : bool const
+operator +operator +setFlightNumber(fn : size_t) : void
+setMileage(miles : size_t) : void
+setDepartTime(time : size_t) : void
+setArriveTime(time : size_t) : void
+setToCity(city : char) : void
+setFromCity(city : char) : void
+setSeatMap(sm : const vector&) : void
+getMaxPassengers() : size_t
+getFlightNumber() : size_t
+getMileage() : size_t
+departureTime() : size_t
+arrivalTime() : size_t
+fullSeats() : size_t
+freeSeats() : size_t
+getBounceCount() : size_t
+getToCity() : char
+getFromCity() : char
+getSeatMap() : vector
+incBounceCount() : void
+decBounceCount() : void
+addPassenger(pd : const PassengerData&) : bool
+findPassenger(pd : const PassengerData&) : bool
+removePassenger(pd : const PassengerData&) : PassengerData

`static const size_t MAX_PASSENGERS`

The maximum number of passengers that the flight can hold, currently, this is set to `40`.

`size_t flightNumber`

Unique identifier for the flight. It is used to compare flights to determine if they are the same, when searching for a flight for example. It is the key for the `FlightData` object. `void`

`setFlightNumber(size_t)` and `size_t getFlightNumber()` are used to set and get the flight number.

`size_t departTime` `size_t arriveTime`

The departure and arrival time of the flight, respectively. They are formatted simply as HHMM, where H represents the hour and M represents the minute. The time is stored in 24 hour time and the overloaded stream insertion operation is used to format this time into a human readable format. Storing the time like this creates and easy to use and efficient way to move the data considering the times are static. `void setDepartTime(size_t)` and `void setArriveTime(size_t)` are used to set the departure and arrival times to the `size_t` argument that they accept. `size_t departureTime()` and `size_t arrivalTime()` are used to get the departure and arrival time of the flight.

`size_t bounced`

The number of passengers that have been popped off the flight to make room for passengers of higher priority. It can be used to ensure that there have not been too many passenger bounced from the flight. `void incBounceCount()` and `void decBounceCount()` are used to increment or decrement this value when a passenger is bounced off the flight or added back to the flight. `size_t getBounceCount()` will return the number of passengers that have been bumped from the flight.

`size_t mileage`

The total mileage of the flight between the cities. `void setMileage(size_t)` is used to set the mileage of the flight to the `size_t` parameter that it accepts and `size_t getMileage()` is used to get the mileage of the flight.

`char toCity` `char fromCity`

The characters that represents the city that the flight is going to and the city that the flight is coming from, respectively. `void setToCity(char)` and `void setFromCity(char)` are used to set these cities to the `char` parameter that they accept. `char getToCity()` and `char getFromCity()` return these characters representing the cities.

`size_t fullSeats()` `size_t freeSeats()`

Return the number of seats that are currently occupied on the flight and the number of seats that are currently available on the flight, respectively. `size_t fullSeats()` returns the size of `vector<PassengerData> seatMap` by using the `size()` function and `size_t freeSeats()` returns `static const size_t MAX_PASSENGERS` minus the size of `vector<PassengerData> seatMap`.

`vector<PassengerData> seatMap`

A `vector<PassengerData>` that contains all the passengers currently on the flight. Member

functions `bool addPassenger(const PassengerData&)` and `bool findPassenger(const PassengerData&)` are used to add a passenger to the flight or determine if a passenger is on the flight. They return `bool` based on whether or not the operation was successful or not. When adding a passenger, it will return `false` if the flight is already full and when searching for a passenger of the flight, it will return `false` is the passenger is not on the flight. `PassengerData removePassenger(const PassengerData&)` will remove the specified passenger from the flight, and if the passenger is not on the flight, it will return an empty passenger. These functions accept a `PassengerData` object by constant reference so the objects do not have to be copied when passed as parameters. `vector<PassengerData> getSeatMap()` and `void setSeatMap(const vector<PassengerData>&)` are used to get and set `vector<PassengerData> seatMap`. They are used when performing seating checks to make sure that there are not too many passengers of any one seating priority and to allow for passengers of higher priority to be added at the expense of the economy class passenger.

`bool operator==(const FlightData&) const` `bool operator>(const FlightData&) const` `bool operator<(const FlightData&) const`

Operator overloads used to determine where a flight should be in a list and whether or not flights are the same, to do comparisons between the flight. They check based on the `size_t flightNumber` of the left and right hand operands, this is effectively the key of the `FlightData` object.

`friend ostream& operator<<(ostream&, FlightData&);`

Used to format and output the data of the flight. The data is aligned, leading zeros are added to make the columns clean, the time is formatted to make is human readable, and the columns are set to a fixed space to keep them looking clean. The time is formatted by simply dividing the `size_t` by 100 to get the hours and modulosing by 100 to get the minutes. These minutes and hours are simply then separated by a colon. The data is formatted as follows:

`flightNumber` fixed to 4, right align	`fromCity` fixed to 12, left align	`toCity` fixed to 12, left align	Formatted `departTime` fixed to 5, left align	Formatted `arriveTime` fixed to 5, left align	`mileage` fixed to 4, left align

In addition, there are two blank spaces before the data begins and there are two blank spaces separating each of the columns.