**CS M20 Final Project**

December 15, 2015



**Team Members**

Max Brain

Ryan Harshberger

Andrew Laux

Matthew Robinson

Jeff Stanton

Bailey Vandehei

Eric Wolfe

**Purpose**

The purpose of this design document is to detail the design for an Airline Reservation System based on the project requirements of Professor Chetlen. The program shall take three external elements to provide the user an airline reservation interface. The three elements are: passenger information, list of flights, and possible flight routes and distances. The user may look up complete details on specific flights, locations, passengers, times, and distance traveled. Manual input for passenger and flights are disallowed however the user may remove passengers, without modifying the original passenger input.

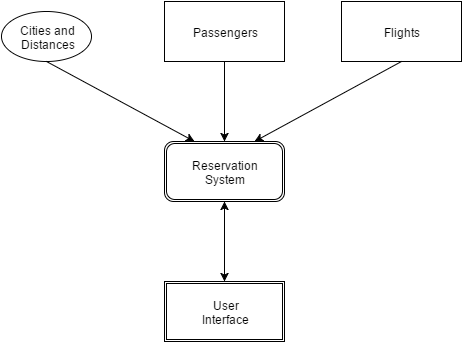
**Design Philosophy**

This program is meant to be interfaced with a user. We assume the user to be an employee of the airline. The airline employee should be accessing specific data based on reservation and flight numbers. The user options that directly ask for reservation number are most optimized.

**Scope**

This design focuses on the interface with the user and computation times. There will be explanations for containers and methodology of functions.

**Overview**



**Initial Resource Data**

1. Passenger Input File

2. Flight Input File

**User Interface**

**Style: Drill Down Menu**

A drill down menu is a good fit for this program. Reading the user requirements as the programmer seemed overwhelming; we decided to group the options in categories. A lot of user requirements overlapped by “functionality”. For example, searching for a passenger has 4 different options to search. This also allows the main menu titles to be much shorter to read. New users should understand fully which each menu accomplishes. One negative aspect is that people who are fully accustomed to a single level menu could navigate all of the options faster. But on that note, we assume the user should not need to move from sub-menu to sub-menu in fast time.

**Time Reference:** At the start of the program the user is required to specify a time reference. This time will be used by the program to generate information about which flights have and have not.

Yet departed. User must input a valid time for the program to continue.

**Input:** a time of day represented as HH:MM (military), ‘e’ or ‘Q’

**Output:** none

**Main Menu:** The main menu of the program has six options listed zero through five. Options one through five take the user to submenus which contain specific tasks to be completed. Option zero quits the program. The options are listed as follows: 1. Display a list of Passengers 2. Display a list of Flights 3. Search for a Passenger 4. Search for a Flight 5. Delete a Passenger from a Flight

0. [Q]uit. Input is taken as a char datatype. If a valid input is not given the program will prompt the user again.

**Input:** 0 – 5, ‘Q’ or ‘q’

**Output:** displays a submenu or quits program.

**Display a List of Passengers:** Item one of the main menu. Displays a submenu of options for displaying passengers and the option to go back to the main menu. The options are as follows: 1. List all passengers and data 2. List passengers by destination 3. List passengers by city of departure 4. List passengers by flight 5. List overbooked passengers 6. List passengers waiting for a flight 0. [B]ack. Input is taken as a char datatype. If a valid input is not given the program will prompt the user again. ‘0’, ‘B’ or ‘b’ returns user to main menu.

**Input:** 0 – 6, ‘B’ or ‘b’

**Output:** takes user through the selected operation

**List All Passengers:** Item one of the Display a List of Passengers submenu submenu. Brings up information for every passenger in the system. Displays passenger’s reservation number, first and last names, membership, and flight number in that order.

**Input:** none

**Output:** data for all passengers

**List Passengers by Destination:** Item two of Display a List of Passengers submenu. Lists passengers aboard all flights bound for a specified destination. Cities are represented as single letters. Program will prompt user again if an invalid city is inputted.

**Input:** a city represented as a single letter. ( not case sensitive )

**Output:** data on specified passengers

**List Passengers by City of Departure:** Item three of Display a List of Passengers submenu. List passengers about flights departing from specified city. Cities are represented as single letters. Program will prompt user again if an invalid city is inputted.

**Input:** a city represented as a single letter. ( not case sensitive )

**Output:** data on specified passengers

**List Passengers by Flight:** Item four of Display a List of Passengers submenu. Allows user list passengers by flight. User must input a valid flight number or the program will prompt the user for another number.

**Input:** a flight number

**Output:** data on specified passengers

**List Overbooked Passengers:** Item five of Display List of Passengers submenu. Displays passenger who could not board a flight which has already departed. This is calculated using user’s time reference entered at the start of the program.

**Input:** none

**Output:** data on specified passengers

**List Passengers Waiting for a Flight:** Item six of Display List of Passengers submenu. Displays passengers who are waiting to board a flight which has not yet departed. This is calculated using user’s time reference entered at the start of the program.

**Input:** none

**Output:** data on specified passengers

**Display a List of Flights:** Item two of the main menu. Displays a submenu of options which display flights. Flights are displayed in order of flight number from least to greatest. Options for displaying flights are as follows: 1. Display all flights 2. Display all flights with passengers.

Program will continue to prompt for input until a valid operation is selected. ‘0’, ‘B’ or ‘b’ returns user to main menu.

**Input:** 0-2, ‘B’ or ‘b’

**Output:** takes user through selected operation

**Display All Flights:** Item one of Display a List of Flights submenu. Displays all flights in order of flight number from least to greatest. Data displayed includes departure time, arrival time, and flight mileage miles.

**Input:** none

**Output:** data on specified flights

**Display All Flights with Passengers:** Item two of Display a List of Flights submenu. Displays all flights in order of flight number from least to greatest along with the passengers of each flight and all their data.

**Input:** none

**Output:** data on specified flight and passengers

**Search for A Passenger:** Item three of the main menu. Displays submenu of options for searching for a passengers. The options are as follows: 1. Search for a passenger by reservation number 2. Search for a passenger by name. Program will continue to prompt for input until a valid operation is selected. ‘0’, ‘B’ or ‘b’ returns user to main menu.

**Input:** 0 – 2, ‘B’ or ‘b’

**Output:** takes user through selected operation

**Search for a Passenger By Reservation Number:** Item one of Search for a Passenger submenu. Allows the user to search for a passenger by his/her reservation number. If no passenger is found to have a matching reservation number the program will in form the user and return to Search for a Passenger submenu.

**Input:** a reservation number (positive integer)

**Output:** the results of the search query

**Search for a Passenger By Full Name:** Item number two of Search for a Passenger submenu.

Allows the user to search for passengers with a matching first and last name. Returns any passenger or passengers that match or informs the user that no matches were found. Returns to submenu upon returning results.

**Input:** a first name ( not case sensitive )

**Input:** a last name ( not case sensitive )

**Output:** result of search query

**Search for a Passenger with First Name:** Item two of Search for a Passenger submenu. Allows the user to search for a passenger using only a first name. Returns any passenger or passengers that match or informs the user that no matches were found. Returns to submenu upon returning results.

**Input:** a first name ( not case sensitive )

**Output:** result of search query

**Search for a Passenger with Last Name:** Item two of Search for a Passenger submenu. Allows the user to search for a passenger using only a last name. Returns any passenger or passengers that match or informs the user that no matches were found. Returns to submenu upon returning results.

**Input:** a last name ( not case sensitive )

**Output:** result of search query

**Search for a Flight:** Item four of the main menu. Displays a submenu of options for searching the database of flights. The options for searching are as follows: 1. Search flight by city to city

2. Search flight by flight number 3. Toggle displaying passengers for a Flight. The submenu requires that a valid option be selected for the program to continue. If an invalid selection is made by the user the program will prompt the user again. Flights shown as results of these search options can be made to display their passengers as well by toggling the displaying passengers for a flight option.

**Input:** 0 - 3, ‘B’ or ‘b’

**Output:** takes the user through selected operation

**Search Flight by City to City:** Item one of Search for a Flight submenu. Allows the user to search for a flight by city of departure and destination city. The user must input a valid destination and city of departure in order for the operation to return a result. The program will return the user back to Search for a flight submenu regardless of results or invalid input.

**Input:** a city of departure represented as a single letter. ( not case sensitive )

**Input:** a city of arrival represented as a single letter. ( not case sensitive )

**Output:** results of search

**Search Flight by Flight Number:** Item two of Search for a Flight submenu. Allows the user to search for a flight by entering flight number. Flight numbers are unique to each flight, so if a match is found then only one result will be displayed. The program returns to the submenu whether a result if found or not.

**Input:** flight number (positive integer)

**Output:** results of search

**Toggle Displaying Passengers for a Flight:** Item three of Search for a Flight submenu. When this option is toggled on searching for a flight will also display all passengers aboard any matching flight. Set to off by default. Toggling switch simply redirects user to the same submenu but with on/off indicator changed to reflect the toggle’s status.

**Input:** none

**Output:** changes the toggle status

**Delete a Passenger from a Flight:** Item five of the main menu. Displays a list of options for deleting a passenger from a flight. Deleting a passenger mean that any passengers waiting to board will be bumped into the open seat. This menu contains the following options for deleting passengers: 1. Delete a Passenger using Reservation Number 2. Delete a Passenger using a Flight Number. A valid selection must be made. If user makes an invalid selection the program will prompt the user for another input.

**Input:** 0 - 3, ‘B’ or ‘b’

**Output:** takes user through selected operation

**Delete a Passenger Using Reservation Number:**  Item one of the Delete a Passenger from a Flight submenu. Allows user to delete a passenger from a flight using a known reservation number which is unique to every passenger. In the case that a passenger is located with a matching reservation number they will be deleted from the flight and their seat will be occupied by another passenger who is waiting for a seat. Attempting to remove the same passenger a second time will result in an error message. If a matching reservation number is not found the program will inform the user and return to Delete a Passenger submenu.

**Input:** a reservation number ( positive integer )

**Output:** the result of the delete operation

**Delete a Passenger Using Flight Number:** Item two of the Delete a Passenger from Flight submenu.

Allows the user to search for a passenger to delete using a known flight number. Upon entering a valid flight number the program will display a list of passengers aboard the matching flight. Given the list, the user can perform a deletion using the first and last name of a passenger aboard the flight. If no matches are found for the flight number, first , or last names- the program will return to the Delete a Passenger from Flight submenu.

**Input:** flight number (positive integer )

**Input:** first name ( not case sensitive )

**Input:** last name (not case sensitive )

**Output:** results of the delete operation

**External Data Input**

**Flights**

Filename: Flights.txt

*flightNumber*

*departureTime*

*arrivalTime*

*fromCity*

*toCity*

Example:

1

0600

0630

L

F

**Passengers**

Filename: Passenger.txt

*firstName*<19 spaces - length of First Name>

*lastName*<19 spaces - length of Last Name>

*Membership*<19 spaces - length of Membership>

*reservationNumber*<4 spaces - length of RN>

*flightNumber*<4 spaces - length of RN>

Example:

Margaret Trundle Pilot Club 1 11

Sharron Mattson First Class 2 16

**Cities**

The user does not have access to change flight routes, cities (names), nor distances. These are hard coded within the program.

**Memory Containers**

**Intensive Containers**

1. Passengers

Container

std::map

Info

Passenger class has one unique data member, their reservation number. The passenger objects are stored in a map which is an associative container. If we allow the data to be mapped by reservation number, any searches made by reservation number will have the average and worst case efficiency of O(log n) when using the find function for map containers. All modifications done to the passenger container will be done by association with the reservation number. The downside is searching for passenger by name always has an efficiency of O(n) in order to iterate through the whole map container to check each passenger’s name. An alternative is to use a multi-map and use the passenger’s name as the key. However this would not be the most efficient way for the user to access and modify data. The user would always be presented with the choice of which passenger to select, if there are other passengers with the same name, instead of inputting a reservation number and not to worry about a second prompt.

2. Flights

Container

std::map

Info

The flight class has a single unique data member, the flight number. The average and worst case efficiency of searching by flight number is O(log n) because the flights are also stored in a map container. Like passenger class, finding a flight by reservation number is the most efficient option for the user. If searching by flight time/location/departure/destination, the user must go through a list of flights to choose which to modify or look at.

**Small Containers**

1. Cities

Container

std::set

Info

The program request lists 14 cities stored in a set. Assuming a drastic amount of cities are not added by the program request, the efficiency of finding a single city is O(1). A set has fast access time in comparison to other STL containers because the value of the objects a set contains is itself the identifier. This is not essential for the current program but can prove useful for future renditions. This container should really only be used to check if a city exists.

**Calculating Mileage**

Each flight needs to know the route it is going to take. Especially how much gas it will consume. We can calculate the amount of miles a plane has to travel.

FlightLocation

Containers

std::map

std::vector

Info

The FlightLocation class holds a data member containing the character representing the city’s “name,” and also contains a vector of adjacent flight locations and the distances to those locations.

FlightMap

Container

std::map

Info

The FlightMap class holds a map from a character of a city location to a FlightLocation which is used to hardcode the flight locations and their distances into the program. This is used to prevent the client from altering the flight map. The flight map also contains a function which uses Dijkstra's algorithm to get the minimum distance between two locations.

Example of calculating mileage

FlightMap map; // constructor creates Flight locations

int distance = map.getMinDistance(‘B’, ‘A’);

**File Names**

1. FinalProject.cpp
2. Flight.h
3. Flight.cpp
4. FlightLocation.h
5. FlightLocation.cpp
6. FlightMap.h
7. FlightMap.cpp
8. Passenger.h
9. Passenger.cpp
10. ReservationSystem.h
11. ReservationSystem.cpp

**Classes**

**Passenger**

The Passenger class is a simple class which is made up of string firstName, string lastName, string Membership, size\_t reservation number, and int flight number. The reservation number will be a unique value given to each passenger object to identify it. The class is a way to hold this information for each passenger and uses get functions to access the data. The class also contains a function to output this data in a specialized format.

Constructors

Passenger ( );

info

- default constructor

- Not the intended use of the class, ints set to -1 to allow for error checking

- sets strings to null, and reservationNumber, flightNumber to -1.

Passenger (string afirstName, string alastName, string aMembership, size\_t areservationNumber, int aflightNumber);

info

- initialize constructor

- sets the values in the class to the ones intended in the class.

Modifier Functions

void setFirstName(string afirstName);

input

- afirstName first name

info

- sets the first name of the Passenger

void setLastName(string alastName);

input

- alastName last name

info

- sets the last name of the Passenger

void setMembership(string aMembership);

input

- aMembership membership of passenger

info

- sets the Membership of the Passenger

void setReservationNumber(size\_t areservationNumber)

input

- areservationNumber unique reservation number

info

- sets the ReservationNumber of the Passenger

void setflightNumber(int aflightNumber)

input

- aflightNumber flight number found in Flight class

info

- sets the flightNumber of the Passenger

Accessor Functions

string getFirstName ( ) const;

returns the FirstName of the Passenger

string getLastName ( ) const;

returns the LastName of the Passenger

string getMembership ( ) const;

returns the Membership of the Passenger

size\_t getReservationNumber ( ) const;

returns the ReservationNumber of the Passenger

int getFlightNumber ( ) const;

returns the FlightNumber of the Passenger

// IO stream

friend ostream &operator << (ostream &, const Passenger &);

**Flight**

A Flight is the abstract idea of a plane traveling from one location to another following airline specific details. The flights contain departure city, destination city, departure and arrival time, and flight number. The flight number will be a unique value given to each flight to identify it.

Each Flight consists of deques for each membership status, which holds passenger reservation numbers to the actual passengers on the flight. There is a map which holds total passengers, including those waiting to board.

Reasoning for Flight containing reservation number

In Flight, we store passengers by reservation number. When we obtain the manifest from the flight, we must then traverse the map of passengers. There is an efficiency of O(log n), but it could be better if the manifest was of passenger pointers. However, the decision against this was made if a future rendition of this program alters the passenger container. If the user requires an option to delete a passenger from data, it is possible all of the pointers in all Flight objects are pointing to invalid locations. A safer way to do it is with a manifest of iterators. However, if an iterator is pointing to an erased passenger, there would be memory problems. The safest way to handle this situation is to directly search the map for passenger reservation numbers.

Constructors

Flight ( );

info

-sets the time of the flight. The rest of the variables are not initialized.

Flight (const size\_t& flightNumber, const size\_t& departTime, const size\_t& arriveTime, const size\_t& mileage, const string& fromCity, const string& toCity);

input

- flightNumber

- departureTime in format hhmm

- arriveTime in format hhmm

- mileage shortest distance calculated by FlightMap

- fromCity a string representing the departure city

- toCity a string representing the arrival city

info

- sets the variables in the respective class to the ones in the

- initialization list. Then sets the time.

Modifier Functions

void addFlightNumber(const size\_t& newFlightNumber);

input

- newFlightNumber

info

- sets newFlightNumber to the FlightNumber.

void addOrigin(const string& from);

input

- from a string representing a departure city

info

- sets origin to from.

void addDestination(const string& to);

input

- to a string representing an arrival city

info

- sets Desetination to to.

void addDepartTime(const size\_t& departureTime);

input

- departureTime in form hhmm

info

- sets departureTime to departureTime.

void addArriveTime(const size\_t& arrivalTime);

input

- arrivalTime in form hhmm

input

- sets arrivalTime to arriveTime

void addMileage(const size\_t& mileage);

input

- mileage distance calculated by FlightMap

info

- sets the mileage from the graph class. DOes this in the addflight function.

void addPassenger(const size\_t& reservation), const string& membership);

input

- reservation reservationNumber of a passenger

- membership of the passenger

info

- if 10 passengers have been moved already been moved, new passenger is - put on the waitlist

- adds the passenger to their respective class and places them into the manifest.

- return true if passengers (seated + waiting) <= 50

- return false when waitlist >= 10 (can only displace 10 passengers)

void removePassenger(const size\_t& reservation);

input

- reservation unique number of a passenger

info

- Assumes the flight has not taken off yet.

- Returns true if passenger is on flight.

- Returns false if passenger is not on flight.

Accessor Functions

size\_t getFlightNumber ( ) const;

info

- returns flightNumber as a size\_t

tm getDepartTime ( ) const;

info

- returns DepartTime as struct tm

tm getArriveTime ( ) const;

info

- returns ArriveTime as struct tm

size\_t getMileage ( ) const;

info

- returns Mileage as size\_t

string getOrigin ( ) const;

info

- returns Origin as a string

string getDestination ( ) const;

info

- returns Destination as a string

vector<size\_t> getManifest ( ) const;

info

- returns a list of all passengers by reservation number aboard flight

void findPassenger(const size\_t& reservation) const;

**FlightLocation**

This class is an abstract idea of a city a plane can fly over. Planes can only fly to adjacent cities from their current location. This class contains the possible cities, their adjacent cities, and the distances between each other. FlightMap relies on the functions of this class.

Constructors

FlightLocation();

info

- sets data member identifier to default value, UNKNOWN

FlightLocation(char identifer);

input

- identifier char to represent city

info

- sets the identifier to a char value of the city

Public functions

FlightLocation& addAdjacentLocation(FlightLocation& location, int distance);

input

- location a FlightLocation object

- distance distance between two adjacent loactions

info

- adds adjacency between two locations

- returns the FlightLocation added for chaining

Accessor Functions

std::vector<char>::iterator getAdjacencyIterator();

info

- returns iterator for iterating through adjacent locations

std::vector<char>::iterator getIteratorEnd();

info

- returns iterator at the end for iterating through adjacent locations

int getDistanceToLocation(char location);

input

- location a char that represents a city

info

- returns distance between two locations

char getIdentifier() const;

info

- returns char represent a city

Assignment Operator

FlightLocation& operator=(const FlightLocation& other);

Equivalence Operators

bool operator==(const FlightLocation& other);

bool operator==(const char& other);

input

- other a char representing a city

info

- returns true if the FlightLocation object’s identifier equals other

bool operator!=(const FlightLocation& other);

bool operator!=(const char& other);

input

- other a char representing a city

info

- returns true if the FlightLocation object’s identifier does not equal - other

**FlightMap**

Abstractly, this is a singleton class. Only one instance of this object should be made in which the constructor creates FlightLocations and stores them in a map. The client is then able to use getMinDistance to find the minimum distance between two cities.

Constructor

FlightMap ( );

Public functions

int getMinDistance ( char startIdentifier, char endIdentifier );

input

- startIdentifier a char representing the departure city

- endIdentifier a char representing the arrival city

info

- returns an int of the shortest mileage between the two cities

**ReservationSystem**

The interface for the user. This class contains the major data containers including list of passengers and list of flights. The client and user are not allowed to modify any data except open and use the menu provided.

Constructor

ReservationSystem ( tm baseTime );

input

- baseTime the time frame the client would like to view the program

Menu function

bool runMenu ( );

return

- bool whether the user wants to continue running the menu

**Accessor Algorithms**

**Search for a Passenger**

* by Name (Last) (Last, First)

vector<size\_t> reservationNumbers;

loop (map iterator through passengers)

Passenger pas = it->second();

if (searchName == pas.getName())

names.push\_back(pas.getReservationNumber());

return names;

* by Reservation Number

return passengers[reservation number];

Searching by name requires the entire list of passengers to be searched. There will always be an efficiency of O(n). It does not stop at the first customer with that name. Searching by reservation number gives us the best case efficiency of O(1) and average and worst case efficiency of O(log n).

**Search for a Flight**

* by Flight departure/destination/time (data):

vector<int> flight\_numbers;

loop (map iterator through flights)

Flight temp\_flight= it->second();

if (searchData == temp\_flight.getData())

flights.push\_back(flight\_numbers);

* by Flight Number

return flights[flight number];

There are too many variables to search flights by. Of those variables, there is only one unique data member. Just like the passenger class, searching by flight number gives best case efficiency of O(1) and average and worst case efficiency of O(log n).

**Search for Passengers Arriving/Departing from a City**

loop ( all flights )

if ( flight city == search city )

vector = flights.getManifest()

print manifest

The pseudo code shows the inefficiency of searching by city. However, knowing the entire list of passengers who are arriving at a city is not a frequently repeated task of a user. We sacrifice the efficiency for searching by flight number.