OOP RESERVATIONS PROJECT

OVERVIEW

SUBMISSION REQUIREMENTS

a. Must be 100% your own work.
If you submit a code that you did not write on your own, you may fail the course.
If you share your code with others and they submit it as their own, you may fail the course.
I encourage you to discuss concepts and approaches with others but do not share code.

b. Must be submitted as standard operating system .zip file with the name *firstname_lastname.zip.*
See Submission Format for more information.

c. Must run from your *makefile* without errors (**compiler errors will receive a 0 grade**).
Comment out code which does not compile for possible partial credit.

d. Must run all files without *using namespace std* (review namespace examples under Data Structures)

e. Must include all specified files coded using separate compilation as explained in class.

f. Must be neat, well indented and easy to read (like my class examples).

GRADING PROCEDURE

I will download your submission from Blackboard.
I will know who whose project it is by the .zip file name.
I will successfully unzip your file using the default OSX extraction utility.
I will expect that your *makefile* has been edited to automatically run one of the *Phase_#.cpp* files.
I will open your folder in a console and type *make*.
I will expect the project to compile and run without errors.

If any of those expectations are not met, I will immediately stop what I am doing and issue a grade of 0.

If the above expectations work as specified, I will inspect your project further and issue a grade based upon the *phase* of the project that you have completed.

At a minimum your project must automatically run and compile with an unedited version of *Phase_1.cpp*.

SUBMISSION FORMAT

Zip file as specified. WinRAR, 7-zip or other third-party compression formats will not be graded.

Makefile
Submit an edited *makefile* which runs the *phase* file you wish to have graded.

Windows
Select/Highlight all files in your project. *Right click*, *Send To*, *Compressed (zipped) Folder*.

OSX
Select/Highlight all files in your project. *Right click*, *Compress X Items*.

Blackboard
1. Rename the zip file *firstname_lastname.zip* such as *John_Smith.zip.*
2. Open the *Project* folder in Course Documents.
2. Click *Project Assignment* Link.
3. Drag your named .zip file into the *upload* box. Click *Submit*.

REQUIRED FILES FOR SUBMISSION

makefile	(edit to automatically run the *Phase_#.cpp* you choose)
MyArray.h	(modified version of Template Homework, *MyArray.cpp* not required)
Airline.h	
Airline.cpp	
Airport.h	
Airport.cpp	
Flight.h	
Flight.cpp	
Passenger.h	
Passenger.cpp	
Person.h	(abstract class, *Person.cpp* not required)
Pilot.h	
Pilot.cpp	
Phase_1.cpp	(do not edit, used to test project skeleton)
Phase_2.cpp	(do not edit, used to test default constructors and output)
Phase_3.cpp	(do not edit, used to test accessors and mutators)
Phase_4.cpp	(do not edit, used to test *MyArray* data members and related functions)
Phase_5.cpp	(must edit, used to build a project interface)

DOWNLOADABLE FILES

makefile.	(edit to automatically run the *Phase_#.cpp* you choose)
Phase_1.cpp	(do not edit, used to test project skeleton)
Phase_2.cpp	(do not edit, used to test default constructors and output)
Phase_3.cpp	(do not edit, used to test accessors and mutators)
Phase_4.cpp	(do not edit, used to test *MyArray* data members and related functions)
Phase_5.cpp	(must edit, used to build a project interface)

CONSOLE/MAKEFILE COMMANDS

OSX	Windows	
make	mingw32-make	run the makefile
make clean	mingw32-make clean	run the clean portion of the makefile
clear	cls	clear screen
./prog	prog.exe	execute the program named prog.exe
rm	del	delete in the clean section of the makefile
cd	cd	change directory
ls	dir	view folder directory (files and folders)

CONCEPT: FORWARD DECLARATIONS

For this project *Forward Declarations* of classes are required since classes are co-dependent upon each other. A practical example of this co-dependency would be if a Doctor object contains a list of Patient objects, and each Patient object contains a list of Doctor objects. In this situation, do we declare the Doctor or the Patient first? The solution is to use *Forward Declarations* in each file as follows:

```
// Doctor.h file
#include “Patient.h”                  // include Patient.h
class Patient;                        // forward declaration of the Patient class
class Doctor {
   Patient *patient_list;             // array of Patient objects
 };
// end of Doctor.h file
```

```
// Patient.h file
#include “Doctor.h”                   // include Doctor.h
class Doctor;                         // forward declaration of Doctor class
class Patient {
   Doctor *doctor_list;               // array of Doctor objects
};
// end of Patient.h file
```

CONCEPT: AGGREGATIONS AND INHERITANCE

This project will associate classes via Aggregations and Inheritance.

Within the scope of this project class aggregations will not require the big three since each object must have independent lifetimes and each object should not be cloned. This includes the majority of class types including *Airline*, *Airport*, *Flight*, *Person* and *Pilot*. Practically speaking *Flights* would be a logical candidate for cloning, but this is not a requirement for this project.

Furthermore, some use of inheritance/polymorphism will be required with the *Passenger* and *Pilot* classes since they are both of type *Person*.

CLASS ASSOCIATIONS

This project implements part of a simple flight reservation system using a variety of classes associated via aggregation and inheritance.

The is a simple list of classes and their data member associations (aggregation or inheritance) with other classes. Keep these associations in mind when considering dependencies and function requirements.

Airline

Airport
MyArray Flight (arrivals)*
MyArray Flight (departures)*

Person
MyArray Flight (flights)*

Passenger *(inherits from Person)*

Pilot *(inherits from Person)*

Flight
Airline
Airport (source)
Airport (destination)
Pilot
MyArray Person (passengers)*

PROJECT PHASES

This is the probably the most complex program you have attempted to develop thus far. For this reason, the project has been divided into five implementation *phases*. Five *Phase_#.cpp* files (1 to 5) have been provided to test your project at each *phase* of development.

Your project will run with only one *Phase_#.cpp* file at a time which is decided by editing your makefile.

Phase_1.cpp through *Phase_4.cpp* files must not be edited in your project submission to receive credit. *Phase_5.cpp* must be edited to match the client interface specifications of the *fifth phase*.

You should read this entire document (especially class specifications) before starting the project.

Watch the video to help you understand how to proceed and what is required.

Phase 1: Implement the project skeleton

1. Make sure your Template Homework MyArray container compiles as is. If not fix it first.
   Copy the template *MyArray* class code (nothing else) into a *MyArray.h* file.
2. Create empty .h and .cpp files for all of the files listed above.
3. Only add the following required code for each file (review the class descriptions):
   a. *include* statements
   b. *header guards* (#ifndef, def, #endif code)
   c. *forward declarations* if required
   d. Basic class code including:
      1. All data members except for *MyArray* objects
      2. Default constructor declarations in .h files, definitions in .cpp files (no other functions)
4. Duplicate the downloaded *makefile* for easy copy/paste editing later on.
5. It may help to remove any references to *Flight* in the *makefile* until you get all other files working.
6. Repetitively compile and debug until the skeleton of the program is functioning.

Do not advance until Phase 1 compiles perfectly.

Phase 2: Constructors and Output

1. Modify the *makefile* to run *Phase_2.cpp* instead of *Phase_1.cpp*.
2. Comment out and uncomment parts of *Phase_2.cpp* as you work through this *phase*.
3. Complete all constructor functions with exception to *MyArray* object content for these classes:
   Airline, *Airport*, *Person*, *Pilot*, *Passenger*, *Flight*
4. Complete *getName* accessor function declarations and definitions for the following classes:
   Airline, *Airport*, *Pilot*, *Passenger*
5. Complete the *overloaded* << *operator* functions for the following classes:
   Airport, *Passenger*, *Pilot*, *Flight*

Do not advance until Phase 2 compiles perfectly.

Phase 3: Accessors and Mutators

1. Modify the *makefile* to run *Phase_3.cpp* instead of *Phase_2.cpp*.
2. Add the remaining specified *set* and *get* (not add/remove) functions for the following classes:
   Airline, *Airport*, *Flight*, *Passenger*, *Person*, *Pilot*
3. For the moment avoid any code which requires add/remove functions for *MyArray objects*.

Do not advance until Phase 3 compiles perfectly.

Phase 4: MyArray Objects and Functions

1. Modify the *makefile* to run *Phase_4.cpp* instead of *Phase_3.cpp*.
2. Add *MyArray* objects as specified to the following classes:
   Airport, *Person*, *Flight*
3. Add *addArrival*, *removeArrival*, *addDeparture*, *removeDeparture* functions to class *Airport*.
   Update the *Flight* multi-parameter constructor to work with these new functions.
4. Add Pilot *addFlight* and *removeFlight* functions.
   Add Passenger *addFlight* and *removeFlight* functions.
   Add Flight *addPassenger* and *removePassenger* functions
   Add Flight *getPassenger*, *listPassengers* and *getNumPassengers* functions
   Note that all of these functions and similar and are dependent upon each other.
   You need to update all of these functions progressively and simultaneously while testing for compiler errors.
5. Add any remaining class functions (check all files and class specifications).

Do not advance until Phase 4 compiles perfectly.

Phase 5: Client Interface

The client interface is a program that uses all classes of your project to build a flight reservation system.

1. Modify the *makefile* to run *Phase_5.cpp* instead of *Phase_4.cpp*.
2. Modify *Phase_5.cpp* to create a flight reservation client interface.
   See the Client Interface description at the end of this document (last two pages).

CLASS SPECIFICATIONS

Class Airline

Data member:
name (string): name of the airline

Functions:
a. *default constructor*
b. *one parameter constructor*
c. *name mutator*
d. *name accessor*

Class Airport

Data member:
name (string): name of the airport
symbol (string): 3 letter symbol, for example John F Kennedy is JFK
arrivals (MyArray): MyArray of arriving flight object pointers
departures (MyArray): MyArray of departing flight object pointers

Functions:
a. *default constructor*
b. *two parameter constructor*
c. *name mutator and accessor*
d. *symbol mutator and accessor*
e. *addArrival*
   1. Call whenever a flight *destination* is set to this airport
   2. Check if a flight is in *arrivals*
   3. If so, add flight to *arrivals*
f. *addDeparture*
   1. Call whenever a flight *source* is set to this airport
   2. Check if a flight is in *departures*
   3. If so, add flight to *departures*
g. *removeArrival*
   1. Call whenever a flight d*estination* is no longer this airport
   2. Check if a flight is in the *arrivals*
   3. If so, remove flight from *arrivals*
h. *removeDeparture*
   1. Call whenever a flight s*ource* is no longer this airport
   2. Check if a flight is in *departures*
   3. If so, remove flight from *departures*
i. *closeAirport*
   1. All *flights* in *arrivals* and *departures* should have *source*/*destination* set to nullptr (use flight functions)
j. *overloaded << operator*
   1. Print *name*, *symbol* and list *arrivals* and *departures*, if no *arrivals* or *departures* print *none*.

Class Person

This is an abstract class which does not require a .cpp file, only a .h file.

Data member:
name (string): name of the airline

Functions:
a. *default constructor – inline definition*
b. *one parameter constructor – inline definition*
c. *name mutator – pure virtual function (defined in derived classes)*
d. *name accessor – pure virtual function*
e. *addFlight – pure virtual function*
f. *removeFlight – pure virtual function*

Class Pilot

This class is derived from class Person.

Data member:
flights (MyArray): MyArray of flight object pointers

Functions:
a. *default constructor*
b. *one parameter constructor*
c. *addFlight*
   1. Check if a flight is in *flights*
   2. If not, add flight to *flights*
   3. Set *flight pilot* to this pilot (use flight functions)
d. *removeFlight*
   1. Check if a *flight* is in *flights*
   2. If so, remove *flight* from *flights*
   3. Set *flight pilot* to nullptr (use flight functions)
e. *overloaded << operator*
   1. Print *name* and list *flights*

Class Passenger

This class is derived from class Person.

Data member:
flights (MyArray): MyArray of flight object pointers

Functions:
a. *default constructor*
b. *one parameter constructor*
c. *addFlight*
  1. Check if a *flight* is in *flights*
  2. If not, add *flight* to *flights*
  3. Add *passenger* to *flight passenger*s (use flight functions)
d. *removeFlight*
  1. Check if a *flight* is in *flights*
  2. If so, remove *flight* from *flights*
  3. Remove *passenger* from *flight* (use flight functions)
e. *cancelFlights*
  1. Remove *passenger* from every *flight* in *flights* (use flight functions)
f. *overloaded << operator*
  1. Print passenger *name* and list the *flights*

Class Flight

Data member:
number (int): the number of the flight
airline (Airline): the flight airline
source (Airport): the airport the flight will depart from
destination (Airport): the airport the flight will arrive at
pilot (Pilot): the pilot of the flight
passengers (MyArray): MyArray of Person object pointers

Functions:
a. *default constructor*
b. *multi parameter constructor*
  1. Initialize *number*, *airline*, *source*, *destination* and *pilot*
  2. Add this *flight* to the *source* airport *departures* (do this after *Airport* class is complete)
  3. Add this *flight* to the *destination* airport *arrivals* (do this after *Airport* class is complete)
  4. Add this *flight* to the *pilot flights* (do this after *Pilot* class is complete)
c. *getNumber – return flight number*
d. *getAirline* – return *airline* pointer
e. *getSource* – return *source* pointer
f. *getDestination* – return *destination* pointer
g. *getPilot*
  1. If *pilot* is nullptr return “No Pilot”
  2. otherwise, return *pilot name* as string (use pilot functions)

h. *getPassenger* -
   1. given an *index*, return a *Person* object reference to a *passenger* in *passengers*
i. *setAirline* - *Airline* object as parameter
j. *setSource* - *Airport* object as parameter
k. *setDestination* - *Airport* object as parameter
l. *setPilot*
   1. *Pilot* object as parameter
   2. If replacing a *pilot*, remove this *flight* from old *pilot flights*
   3. Set *pilot* to new *pilot* and add *flight* to new *pilot flights*
m. *nullPilot* - set *pilot* to nullptr
n. *nullSource* - set *source* to nullptr
o. *nullDestination* - set *destination* to nullptr
p. *addPassenger*
   1. Accept a *Person* object by parameter.
   2. Check if *passenger* is in *passengers*.
   3. If not, add *passenger* to *passengers* and add *flight* to *passenger flights*
q. *removePassenger*
   1. Accept a *Person* object by parameter.
   2. Check if *passenger* is in *passengers*.
   3. If so, remove *passenger* from *passengers* and remove *flight* from *passenger flights*
r. *listPassengers* - print a list of the passengers
s. *getNumPassengers* - return the number of passengers
t. *cancel*
   1. Remove *flight* from *source* airport *departures*, set *source* to nullptr
   2. Remove *flight* from *destination* airport *arrivals*, set *destination* to nullptr
   3. Remove *flight* from *pilot flights*, set *pilot* to nullptr
   4. Remove *flight* from each *passenger flights* in *passengers*
r. *overloaded* << *operator*
   1. Print *number*, *airline*, *source*, *destination*, *pilot* and list *passengers*.

CLIENT INTERFACE

Modify the *Phase_5.cpp* file to implement a flight reservation client interface as specified below. The provided file implements a few global containers and a *read* function with some data to get you started

Watch the end of project video for a walk through of the client interface.

EXPECTATIONS/ CONCERNS

The description for the client interface is somewhat vague to provide some flexibility in implementation.

When modifying objects it is extremely important to update all associated objects. For example if you delete a *Flight* you have to update associated *MyArray* objects such as source object *departures*, destination object *arrivals*, pilot object *flights* and passenger object *flights*. Always consider how all associations will be impacted when modifying data.

Above all, it is important that the interface you code is easy to read, easy to understand and bug free. It is significantly better to code less with no bugs then to code more with tons of bugs.

CONTAINERS (provided In original *Phase_5.cpp* file)

Airlines
Airports
Passengers
Pilots
Flights

MENUS

MAIN MENU:

1 Airports	(opens Airports submenu)
2 Flights	(opens Flights submenu)
3 Passengers	(opens Passengers submenu)
4 End program	(end the program)

AIRPORTS SUBMENU

1 List Airports	(list *Airports* by index)
2 Create Airport	(create a new *Airport* name and symbol)
3 Delete Airport	(list *Airports*, select by index to delete)
4 Main Menu	(return to *Main Menu*)

FLIGHTS SUBMENU

1 List Flights	(list *Flights* by index)
2 Add Flight	(create a new *Flight*, select all data members by index)
3 Delete Flight	(list *Flights*, select by index to delete)
4 Select Flight	(list *Flights* and select by index to open *Flight Submenu*)
5 Main Menu	(return to *Main Menu*)

FLIGHT SUBMENU

1 Change Pilot	(list pilots, select by index to modify pilot)
2 Change Departure Airport	(list *Airports*, select by index to modify *source airport*)
3 Change Arrival Airport	(list *Airports*, select by index to modify *destination airport*)
4 Add Passenger	(list *Passengers*, select by index to add *Passenger*)
5 Remove Passenger	(list *Passengers*, remove by index to remove *Passenger*)
6 Flights Menu	(return to *Flights Submenu*)

PASSENGER SUBMENU

1 List Passengers	(lists all *Passengers* in database)
2 Create Passenger	(create a new *Passenger* and add it to the *Passengers* container)
3 Delete Passenger	(lists *Passengers*, select by index to delete)
4 Main Menu	(return to *Main Menu*)

FUNCTIONS

These are a number of recommended functions you should implement to support menu actions in the *Phase_5.cpp* file. The purpose of these functions should be relatively obvious if you examine the menus. Many of these functions access the global containers to provide a list of options for the user to select from.

listAirlines
listAirports
listFlights
selectPilot
selectSourceAirport
selectDestinationAirport
addPassenger
removePassenger
changeFlight (can use this to run the Flight Submenu or code it in Main)
createAirport
deleteAirport
createPassenger
deletePassenger