Computer Science & Engineering Department
I. I. T. Kharagpur

Software Engineering: CS20006

Assignment – 3: Inheritance Hierarchy, Design, Analysis & Testing *Marks: 100*

Assign Date: *15th February, 2021* Submit Date: *23:55, 4th March, 2021*

We need to develop a rudimentary railway reservation / booking system (somewhat like IRCTC Train Ticket Booking, but extremely scaled down in features). We present various stages of this development process leading finally to the specific tasks of the assignment.

1 Specification

This is the outline specification that has been acquired from the client.

1.1 Requirement Statement

The entities involved in the booking system design include:

- Station: Every Station is identified by its name. Booking is done between any two Stations.
- Railways: It is the Indian railways. It has a collection of Stations with pairwise distance between Stations known a priori. Naturally, there can be only one Railways, called IndianRailways, in the system.
- Date: Any valid date in dd/MMM/yy format.
- BookingClass: There are several BookingClasses for travel (as in Indian Railways fare classes explained). Each BookingClass has the following attributes:
  - *Name*: Name of the BookingClass
  - *Fare Load Factor*: The factor by which the fare for travel by this BookingClass would be loaded over the base fare. This may change from time to time.
  - *Seat / Berth*: Whether the BookingClass provides sleeping berths or just seats. This will not change in future.
  - *AC / Non-AC*: Whether BookingClass is air-conditioned or otherwise. This will not change in future.
  - *# of Tiers*: How many tiers exist in the coach for this BookingClass. This will not change in future.
  - *Luxury / Ordinary*: Whether this BookingClass is considered luxurious by the Government. This may change from time to time.

  New booking classes may be added in future.
- Booking: A Booking is requested with the following information:
  - *fromStation*: Station from which the travel starts for the Booking. This is given by the name of the Station
  - *toStation*: Station at which the travel ends for the Booking. This is given by the name of the Station
  - *date*: Date of travel for the Booking
  - *bookingClass*: BookingClass for the Booking
  - *passenger*: Details of the passenger including name, aadhaar number, date of birth, gender, mobile number, and category of the passenger. *This is for future extension and optional for now.*

  On request of a Booking, the same is processed and fare is computed based on the business logic given in Section 1.3. The Booking is then confirmed with *PNR* and other details on the output. *PNR* is serially allocated starting with 1.
- Passenger: A Passenger may have the following details:
  - *name*: Name of the passenger

- *aadhaar #*: Aadhaar Number to be used as a unique ID
- *dateOfBirth*: Date of birth to be used for verification of age
- *gender*: Gender of the passenger: *male* or *female*
- *mobile #*: Mobile number (optional)
- *category*: One of *General*, *Ladies*, *Senior Citizen*, *Divyaang*, *Tatkal*, *Premium Tatkal*

1.2 Assumptions

The following **assumptions** are made for the design:

- IndianRailways has a given set of Stations with distances known a priori. The list of Stations and distances between them are given as Master Data in Section 1.4. No new station can be added to the IndianRailways and distance between pair of stations do not change.
- A Booking, as requested, is always available - between any pair of Stations, on any Date, and for any BookingClass
- No passenger information is considered for the Booking

1.3 Business Logic

The fare between a pair of stations for a booking class is determined through the following steps:

- *Base Fare Rate*: The base fare for every KM of travel = Rs. 0.5. This may change from time to time.
- *Base Fare*: The base fare between two stations is computed by multiplying the distance between the stations with the base fare for every KM of travel. The base fare applies to the *Sleeper* booking class.
- *Loaded Fare*: For booking classes other the *Sleeper*, the fare is loaded by a factor with respect to the *Sleeper* booking class fare as shown in the *Booking Class Matrix* (Section 1.4.2). The load factor may change from time to time.
- *AC Surcharge*: Further, for air-conditioned classes, AC surcharge of Rs. 50 will be charged on the *loaded fare*. This may change from time to time.
- *Luxury Tax*: Finally, there is a 25% luxury tax to be imposed for all luxury class bookings on the fare computed with surcharge. This may change from time to time. The luxury classification as well as taxation rate may change from time to time.
- Final fare is rounded to the nearest integer.
- *Date* has no effect on the fare.
- *Passenger* has no effect on the fare as it is being ignored for now.

1.3.1 Example

For a booking from **Delhi** to **Mumbai**:

By *AC3Tier*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = 1447km * Rs. 0.5 / km =
- Loaded fare for *AC3Tier* = Rs. 723.50 * 1.75 = Rs. 1266.125
- After adding the AC surcharge, we get Rs. 1266.125 + Rs. 50 = Rs. 1316.125 = Rs. 1316/= (rounded)

By *ACFirstClass*:

- Distance from Delhi to Mumbai = 1447km
- Base fare = 1447km * Rs. 0.5 / km =
- Loaded fare for *ACFirstClass* = Rs. 723.50 * 3.0 = Rs. 2170.50
- After adding the AC surcharge, we get Rs. 2170.50 + Rs. 50 = Rs. 2220.50
- Finally, we levy the luxury tax to get Rs. 2220.50 * 1.25 = Rs. 2775.625 = Rs. 2776/= (rounded)

1.4 Master Data

1.4.1 Stations

IndianRailways has *five* stations, namely: *Mumbai*, *Delhi*, *Bangalore*, *Kolkata*, and *Chennai*. The distances between the stations are given below:

Station Distance Matrix

From Station	**To Station**				
	Mumbai	*Delhi*	*Bangalore*	*Kolkata*	*Chennai*
	Distance in KM				
Mumbai	X	1447			
Mumbai			981		
Mumbai				2014	
Mumbai					1338
Delhi		X	2150		
Delhi				1472	
Delhi					2180
Bangalore			X	1871	
Bangalore					350
Kolkata				X	1659

Distance between a pair of stations is symmetric

1.4.2 Booking Classes

IndianRailways has *seven* booking classes as follows - shown with their respective attributes:

Booking Class Matrix

Booking Class	*Name*	*Fare Load Factor*	*Seat / Berth*	*AC*	*# of Tiers*	*Luxury / Ordinary*	*Remarks*
ACFirstClass (1A)	**AC First Class**	3.00	Berth	Yes	2	Luxury	AC 2 berth coupe
AC2Tier (2A)	**AC 2 Tier**	2.00	Berth	Yes	2	Ordinary	AC 2 berth inside, 2 berth on side
FirstClass (FC)	**First Class**	2.00	Berth	No	2	Luxury	Non-AC 2 berth coupe
AC3Tier (3A)	**AC 3 Tier**	1.75	Berth	Yes	3	Ordinary	AC 3 berth inside, 2 berth on side
ACChairCar (CC)	**AC Chair Car**	1.25	Seat	Yes	0	Ordinary	AC chairs
Sleeper (SL)	**Sleeper**	1.00	Berth	No	3	Ordinary	Non-AC 3 berth inside, 2 berth on side
SecondSitting (2S)	**Second Sitting**	0.50	Seat	No	0	Ordinary	Bench seating

- *New booking classes may be added in future*
- *Fare load factors may change from time to time*
- *Luxury / Ordinary categorization may change according to tax rules*
- *Seat / Berth & AC / non-AC classification, and # of tiers will not change in future*

2 Analysis of Specification

We first analyze the specifications to identify the classes and hierarchy for the design. We also try to extract possible constraints on the design.

- Station and Date are simple data classes.
- Class Railways should be a singleton and should contain the master data of stations and distances. The singleton should be constant as no station can be added and distances cannot be changed.

- Different Booking Classes should be a polymorphic hierarchy rooted at BookingClasses which may be an abstract base class. Instead of making it a flat hierarchy, it would be good to make it a multi-level hierarchy. This would need identification of abstract base sub-classes that are aligned with one or more properties of the Booking Classes.

  If multiple properties are used in organizing the hierarchy, then the model would need multiple inheritance. However, we do not want to use multiple inheritance for the associated complications and inefficiency. Rather, we would use single inheritance on the strongest property and use the rest as HAS-A with polymorphic value based on the leaf class.

  Naturally, there can be two candidates for this as *Fare Load Factor*, *# of Tiers*, and *Luxury / Ordinary* are more like pure attributes and clearly not useful for hierarchy:

  - *AC or Non-AC*: Air-condition leads to comfort level, and is not fundamental to travel. So this is a weak candidate.
  - *Seat or Berth*: This is fundamental property for a rail travel. So this is a strong candidate.

  So we may introduce several intermediate abstract base classes on the strong property and its closest associated attribute, viz. the number of tiers.

  Further we may note that every concrete booking class has all fixed properties and there should be no need to construct more than one object for any of them. So there may be a singleton constant object for each which, kind of, will stand for its polymorphic type.

  The hierarchy should be extensible in future as new booking classes are added.

- Booking may be treated as a simple concrete class with the parameters mentioned in the specification. We may keep Passenger as a null-able default for future extension.

  Booking may be also be modeled as a polymorphic base class as with the introduction of Passenger in future it is likely to lead to a booking hierarchy.

- Class Passenger may be an empty abstract base class. Since we are not going to use it, we would not need to make objects for the same. However, it would be good to have it as a polymorphic base for future extension, especially since the specification talks of various categories of passengers.

3 High Level Design

Based on the analysis, now we carry out the High Level Design (HLD) below for Classes, Interfaces, Constants, Statics, Exceptions, and overall design considerations.

3.1 Design Principles

The following design principles may be adhered to in the HLD:

- *Flexible & Extensible Design*
  - The design should be flexible. That is, it should be easy to change the changeable parameters (like base rate, load factor etc.) easily from the Application space. This should should not need re-building of the library of classes.
  - The design should be extensible. That is, it should be easy to add new behaviour (classes) wherever indicated in the specification (like Booking Classes, Booking, Passenger, etc.). This should not require a re-coding of the existing applications.
- *Minimal Design*
  - Only the stated models and behaviour should be coded. No extra class or method should be coded.
  - *Less code, less error* principle to be followed.
- *Reliable Design*
  - Reliability should be a priority. Everything should work as designed and coded.
  - Data members, methods and objects should be made constant wherever possible.
  - Parameters should be appropriately defaulted wherever possible
- *Testable Design*

- Every class should support the output streaming operator for checking intermittent output if needed.
- Every class should be tested with an appropriate test application for its unit functionality (Section 6.1).
- Test Applications (Section 6.2) and regression test suites should be designed for testing the application on (at least) the common scenarios of use.

3.2 Classes

- Class Station HAS-A name.
- Class Railways is a singleton called IndianRailways. It has a collection of the Stations and their mutual distances. IndianRailways is a constant object.
- Class Date is discussed in the lecture modules.
- Class BookingClasses HAS-A loadFactor. Remaining attributes may be encoded on the methods in the hierarchy classes.
- Class Booking HAS-A fromStation, toStation date, and bookingClass from the booking request where every station name, date and booking class are assumed to have been given correctly. Further it HAS-A fare computed and PNR allocated. Optionally, it may HAS-A bookingStatus (which would be true for this assignment always) and bookingMessage (which may be "BOOKING SUCCEEDED" for this assignment always).

  Booking should support Passenger as a null-able parameter for future extension.
- You may add any class, any data member to a class, or any hierarchy as you need for implementation. Justify your design choice for them.

3.3 Interfaces

- Constructors / Destructors: Proper constructor and destructor for every class
- Copy Functions: Provide user-defined Copy Constructor and / or Copy Assignment Operator for a class if used in the design (should not be needed). Otherwise, block them.
- Provide output streaming operator for every class to help output process as well as debugging
- Class Station to have GetName() for accessing its name and GetDistance(.) to get distance to another station.
- Class Railways to have GetDistance(., .) to get distance between a pair of stations. It should also have proper interface for making it a singleton IndianRailways
- Class BookingClasses to have GetLoadFactor(), GetName(), IsSitting(), IsAC(), GetNumberOfTiers(), and IsLuxury() to get access to various BookingClasses properties. Depending on the polymorphic hierarchy, these methods may be non-polymorphic and / or polymorphic (and in some case `pure`) in BookingClasses and its various derived classes. Consider making them `const` methods.
- Class Booking to have ComputeFare() to implement the fare computation logic. Should it be `virtual` (polymorphic) for future extensions?
- You may add any interface to a class (or `private` / `protected` methods) as you need for implementation. Justify your design choice for them.

3.4 Constants

The following should be static constants in appropriate classes:

- *Load Factors* of various BookingClasses
- *Base Fare Rate*: Rs. 0.50 / km
- *AC Surcharge*: Rs. 50.00
- *Luxury Tax*: 25% on booking amount

3.5 Statics

- Class Date to have month and day names.
- Class Railways to have sStations (list of stations) and sDistStations (distance between stations).
- Class BookingClasses to have load factors.
- Class Booking to have sBaseFarePerKM, sBookings (list of bookings done), sBookingPNRSerial (next available PNR), sACSurcharge, and sLuxuryTaxPercent
- You may add any `static` to a class as you need for implementation.

3.6 Errors & Exceptions

- All Booking requests are taken to be correct. That is, the Staions as mentioned - do exist, the Date is valid (in future), and no invalid BookingClass is requested
- There is no error in input, processing, or output.
- No error or exception handling to be incorporated in the design for this assignment. However, structure the code flow well so that they can be incorporated later with minimal changes (adhering to the need of flexibility).

4 Low-Level Design

Based on the High Level Design (HLD), we now perform the Low Level Design (LLD). LLD makes use of the specific constructs and idioms of C++.

4.1 Design Principles

The following design principles may be adhered to in the LLD:

- *Encapsulation*
  - Maximize encapsulation for every class
  - Use `private` access specifier for all data members that are not needed by derived classes, if any. Use `protected` otherwise.
  - Use `public` access specifier for interface methods and static constants and *friend* functions only.
- *STL Containers*
  - Use STL containers (like `vector`, `map`, `hashmap`, `list`, etc.) and their iterators. Do not use arrays
  - Use iterators for STL containers. Do not use bare `for` loops.
- *Pointers & References*
  - Minimize the use of pointers. Use pointers only if you need null-able entities
  - If you use pointer for dynamically allocated objects (should be minimized), remember to `delete` at an appropriate position.
  - Use `const` reference wherever possible.

4.2 Design of Classes, Data Members & Methods

This is left as an exercise in the assignment. Design based on the HLD and the principles and document well.

5 Implementation

After completing the LLD, we perform the coding (implementation). In this we adhered to a set of basic guidelines and code organization.

5.1 Basic Coding Guidelines

An indicative set of guidelines are listed in Section A. You may add more on your own.

5.2 Code Organization

Ideally, the definition of every class (or hierarchy) should be put in a corresponding `.h` file with the `static` definitions and method implementations in the respective `.cpp`. The application should be in `Application.cpp` file. However, for simplicity, it would be acceptable if all the codes are put in the `Application.cpp` file with the application.

6 Test Plan

We also need to prepare a test plan to test the implementation at different stages of development so that better quality and productivity can be ensured. Variety of test processes are common. We shall follow two of these in the current assignment.

6.1 Unit Tests

This is typically the basic test process which is engaged during development (however, it may be useful for future testing and debugging as well). In this, we test every class as it is implemented. We test all non-static & static member functions and friend functions. For a class hierarchy, the unit test is done typically at both concrete classes and the overall hierarchy levels specifically checking the polymorphic methods.

For the purpose of understanding, in Section B we illustrate the test plan and test function for a few unit cases for the `Fraction` class we have developed in Assignment 2.

6.2 Application Test

After the units have been tested, we integrate them into the application and test various scenarios for the application. A sample test application was provided for the `Fraction` class in Assignment 2. However, since it was just a single class application, the application code looked pretty much like the unit test application code with the exception of the comparison with golden data.

Like the units, we again need to enumerate scenarios for the application in the test plan and write the application test.

In addition, a sample test application for booking is given in Section C with the expected output in Section C.1. Your codes should pass this test application too.

7 Tasks

The following tasks are to be completed for the assignment:

1. **Design**: Complete the HLD and the LLD. Document the salient points from your design in `Design.txt`. Follow the quality guidelines and design principles outlined above.
2. **Implementation**: Implement the LLD in C++ following the basic coding guidelines (Section A).
3. **Test Planning**: Write a unit test and application test plan in `Testplan.txt` covering all scenarios. Note that no wrong input or erroneous data situation is to be handled. For example, a Date specified will always be valid. So plan tests based only on correct input data.
4. **Testing**: Implement unit test and application test codes and perform testing. For application testing, test with the application given in Section C as well as the application developed by you from the testplan.
5. **Bundle and Submissions**: Name and bundle your files as given in Section 8 and submit to Moodle.

8 Submission of Files

The following files must be submitted as a single ZIP file:

1. `Documents.zip`
   (a) `Design.txt`: The design document stating the design details (especially LLD) with principles and guidelines followed

(b) `Testplan.txt`: The testplan document stating scenarios for unit tests (with golden output if needed) and the scenarios of the test application.

2. `Source.zip`:

   (a) Source (`.cpp`) and header (`.h`) files for implementation.

   (b) Source (`.cpp`) and header (`.h`) files for test application.

   (c) README file that describes the contents of every file in the `Source.zip`. Also, mention the compiler (with version, and compiler options, if any) that you have used.

3. `Outputs.zip`

   (a) Output from the given test application (Section C)

   (b) Output from the your test application developed from the test plan

- The output file can be generated by redirecting the output to a text file or by copy-paste from the console in a text file.
- There is no need to include the `a.out` file.

Every file (with the exception of program output) must have your name and roll number.

9 Marks

The marks are distributed as follows:

Design		**[20]**
	Breakup	
Non-static & static data members	*[4]*	
Non-static[1] & static member functions signatures	*[4]*	
friend function signatures	*[2]*	
Design of BookingClasses Hierarchy	*[10]*	
Implementation		**[25]**
	Breakup	
Non-static & static member functions	*[15]*	
Static data members	*[5]*	
friend function	*[5]*	
Test Planning		**[20]**
	Breakup	
Unit Test Scenarios & Goldens (Completeness of scenarios)	*[15]*	
Application Test Scenarios	*[5]*	
Testing		**[15]**
	Breakup	
Unit Test Application (adherence to test plan)	*[7]*	
Own Test Application (adherence to test plan)		
Output	*[5]*	
On given Test Application (Section C)		
Output	*[3]*	
Quality of Design & Implementation		**[20]**
	Breakup	
Adherence to Design Protocols		
Singletons	*[3]*	
const-ness	*[3]*	
Coding Guidelines	*[5]*	
Extensibility & Flexibity	*[4]*	
Code Comments	*[5]*	

1 Non-static include non-polymorphic as well as polymorphic member functions

A Coding Guidelines

It is advised to follow the guidelines below while coding:

- Use CamelCase for naming variables, classes, types and functions
- Every name should be indicative of its semantics
- Start every variable with a lower case letter
- Start every function and class with an upper case letter
- Use a trailing underscore (_) for every non-static data member
- Use a leading 's' for every static data member
- Do not use any global variable or function (except `main()`, and `friend`s)
- No constant value should be written within the code - should be put in the application as static
- Prefer to pass parameters by value for build-in type and by const reference for UDT
- Every polymorphic hierarchy must provide a `virtual` destructor in the base class
- Prefer C++ style casting (like `static_cast<int>(x)` over C Style casting (like `(int)`)
- The project should compile without any compiler warning
- Indent code properly
- Comment the code liberally and meaningfully
- Adopt more guidelines as you prefer. Try to document them

B Unit Testing Fraction Class

As an example of unit test, let us consider the `Fraction` class we have developed in Assignment 2. We illustrate the test for its one overloaded constructor (`Fraction(int = 1, int = 1)`) and `operator+` only. For this we enumerate the different possible cases to test in a unit test plan.

B.1 Unit Test Plan for `Fraction`

We elucidate the unit test plan for constructor and add operator.

B.1.1 Test Scenarios for Construction of Objects

We consider the `Fraction(int = 1, int = 1)` constructor. The scenarios (including normalization, sign handling, and default) are:

- Normalization
  1. Improper fraction in reduced form
  2. Improper fraction in irreduced form
  3. Proper fraction in reduced form
  4. Proper fraction in irreduced form
  5. Fraction 0 with arbitrary denominator
- Sign handling
  1. Fraction with negative numerator
  2. Fraction with negative denominator
  3. Fraction with negative numerator & denominator
- Default parameters
  1. Fraction with only numerator
  2. Fraction with no parameter

B.1.2 Test Scenarios for Addition Operator

We consider the overloaded add operator `friend Fraction operator+(const Fraction&, const Fraction&)`. The scenarios are (considering the given constructor):

1. Add two fractions
2. Add a fraction with an integer
3. Add an integer with a fraction

Rest of the `Fraction` class can be tested by preparing a similar plan.

B.2 Unit Test Implementation for `Fraction`

For unit testing, we write a static function in the class that has this test code. In the application, we use the 'golden output' for every test and compare for equality. If the expected output is not obtained, a message on test error is printed.

B.2.1 `Fraction` Class Code

Here is the relevant parts of the class including the static test function signature

```
#ifndef __FRACTION_HXX// Control inclusion of header files
#define __FRACTION_HXX

/************ C++ Headers ***********************************/
#include <iostream>// Defines istream & ostream for IO
using namespace std;
```

```
/************ CLASS Declaration ******************************/
class Fraction {

public:

// CONSTRUCTORS
// ------------
Fraction(int = 1, int = 1);// Uses default parameters. Overloads to
// Fraction(int, int);
// Fraction(int);
// Fraction();

// BINARY ARITHMETIC OPERATORS USING FRIEND FUNCTIONS
// --------------------------------------------------
friend Fraction operator+(const Fraction&, const Fraction&);

// Other member functions, static functions, friend functions

// ...

// STATIC UNIT TEST FUNCTION
// -------------------------
static void UnitTestFraction(); // Test application for Fraction

// Data members

// ...
#endif // __FRACTION_HXX
```

B.2.2 `Fraction` Class Unit Test Application Code

```
// To unit test class Fraction
void Fraction::UnitTestFraction() {
    // Check difference cases of fraction construction
    Fraction f1(5, 3);   // Improper fraction in reduced form
    Fraction f2(15, 9);  // Improper fraction in irreduced form
    Fraction f3(3, 5);   // Proper fraction in reduced form
    Fraction f4(9, 15);  // Proper fraction in irreduced form
    Fraction f5(0, 2);   // Fraction 0 with arbitrary denominator
    Fraction f6(-2, 3);  // Fraction with negative numerator
    Fraction f7(2, -3);  // Fraction with negative denominator
    Fraction f8(-2, -3); // Fraction with negative numerator & denominator
    Fraction f9(5);      // Fraction with only numerator
    Fraction f10;        // Fraction with no parameter

    // Check if every object is constructed in the desired way
    if (f1.iNumerator_ != 5 || f1.uiDenominator_ != 3)    // Check members
        cout << "Fraction Consturction Error on Fraction(5, 3)" << endl;

    if (f2.iNumerator_ != 5 || f2.uiDenominator_ != 3) // Check members & reduction
        cout << "Fraction Consturction Error on Fraction(15, 9)" << endl;

    if (f3.iNumerator_ != 3 || f3.uiDenominator_ != 5) // Check members
        cout << "Fraction Consturction Error on Fraction(3, 5)" << endl;

    if (f4.iNumerator_ != 3 || f4.uiDenominator_ != 5) // Check members & reduce
        cout << "Fraction Consturction Error on Fraction(9, 15)" << endl;

    if (f5.iNumerator_ != 0 || f5.uiDenominator_ != 1) // Check members with denominator = 1
        cout << "Fraction Consturction Error on Fraction(0, 2)" << endl;
```

```
    if (f6.iNumerator_ != -2 || f6.uiDenominator_ != 3) // Check members
        cout << "Fraction Consturction Error on Fraction(-2, 3)" << endl;

    if (f7.iNumerator_ != -2 || f7.uiDenominator_ != 3) // Check members & sign flip
        cout << "Fraction Consturction Error on Fraction(2, -3)" << endl;

    if (f8.iNumerator_ != 2 || f8.uiDenominator_ != 3) // Check members & sign flip
        cout << "Fraction Consturction Error on Fraction(-2, -3)" << endl;

    if (f9.iNumerator_ != 5 || f9.uiDenominator_ != 1) // Check default on second parameter
        cout << "Fraction Consturction Error on Fraction(5)" << endl;

    if (f10.iNumerator_ != 1 || f10.uiDenominator_ != 1) // Check default on both parameters
        cout << "Fraction Consturction Error on Fraction" << endl;

    // Check addition of two fractions
    Fraction f11 = f1 + f3; // Add two fractions
    Fraction f12 = f1 + 1;  // Add a fraction with an integer
    Fraction f13 = 1 + f3;  // Add an integer with a fraction

    if (f11.iNumerator_ != 34 || f11.uiDenominator_ != 15) // Check members on add
        cout << "Fraction Addition Error on Fraction(5, 3) + Fraction(3, 5)" << endl;

    if (f12.iNumerator_ != 8 || f12.uiDenominator_ != 3) // Check members on add
        cout << "Fraction Addition Error on Fraction(5, 3) + 1" << endl;

    if (f13.iNumerator_ != 8 || f13.uiDenominator_ != 5) // Check members on add
        cout << "Fraction Addition Error on 1 + Fraction(3, 5)" << endl;

    return;
}
```

C Test Application for Booking

```
// Test application for booking
void BookingApplication() {

    // Bookings by different booking classes

    // <BookingClasses>::Type() returns the constant object of the respective type
    Booking b1(Station("Mumbai"), Station("Delhi"), Date(15, 2, 2021), ACFirstClass::Type());
    Booking b2(Station("Kolkata"), Station("Delhi"), Date(5, 3, 2021), AC2Tier::Type());
    Booking b3(Station("Mumbai"), Station("Kolkata"), Date(17, 3, 2021), FirstClass::Type());
    Booking b4(Station("Mumbai"), Station("Delhi"), Date(23, 3, 2021), AC3Tier::Type());
    Booking b5(Station("Chennai"), Station("Delhi"), Date(25, 4, 2021), ACChairCar::Type());
    Booking b6(Station("Chennai"), Station("Kolkata"), Date(7, 5, 2021), Sleeper::Type());
    Booking b7(Station("Mumbai"), Station("Delhi"), Date(19, 5, 2021), SecondSitting::Type());
    Booking b8(Station("Delhi"), Station("Mumbai"), Date(22, 5, 2021), SecondSitting::Type());

    // Output the bookings done where sBookings is the collection of bookings done
    vector<Booking*>::iterator it;
    for (it = Booking::sBookings.begin(); it < Booking::sBookings.end(); ++it) {
        cout << *(*it);
    }

    return;
}

int main() {

    BookingApplication();

    return 0;
}
```

Your implementation of classes needs to compile with the above application and output details of every booking done. A sample output could look as follows. It is not necessary to match every line of the output. But the same information should be available in your output.

C.1 Test Output

```
BOOKING SUCCEEDED:
PNR Number = 1
From Station = Mumbai
To Station = Delhi
Travel Date = 15/Feb/2021
Travel Class = AC First Class
 : Mode: Sleeping
 : Comfort: AC
 : Bunks: 2
 : Luxury: Yes
Fare = 2776

BOOKING SUCCEEDED:
PNR Number = 2
From Station = Kolkata
To Station = Delhi
Travel Date = 5/Mar/2021
Travel Class = AC 2 Tier
 : Mode: Sleeping
 : Comfort: AC
 : Bunks: 2
 : Luxury: No
```

```
Fare = 1522

BOOKING SUCCEEDED:
PNR Number = 3
From Station = Mumbai
To Station = Kolkata
Travel Date = 17/Mar/2021
Travel Class = First Class
 : Mode: Sleeping
 : Comfort: Non-AC
 : Bunks: 2
 : Luxury: Yes
Fare = 2518

BOOKING SUCCEEDED:
PNR Number = 4
From Station = Mumbai
To Station = Delhi
Travel Date = 23/Mar/2021
Travel Class = AC 3 Tier
 : Mode: Sleeping
 : Comfort: AC
 : Bunks: 3
 : Luxury: No
Fare = 1316

BOOKING SUCCEEDED:
PNR Number = 5
From Station = Chennai
To Station = Delhi
Travel Date = 25/Apr/2021
Travel Class = AC Chair Car
 : Mode: Sitting
 : Comfort: AC
 : Bunks: 0
 : Luxury: No
Fare = 1413

BOOKING SUCCEEDED:
PNR Number = 6
From Station = Chennai
To Station = Kolkata
Travel Date = 7/May/2021
Travel Class = Sleeper
 : Mode: Sleeping
 : Comfort: Non-AC
 : Bunks: 3
 : Luxury: No
Fare = 830

BOOKING SUCCEEDED:
PNR Number = 7
From Station = Mumbai
To Station = Delhi
Travel Date = 19/May/2021
Travel Class = Second Sitting
 : Mode: Sitting
 : Comfort: Non-AC
 : Bunks: 0
 : Luxury: No
Fare = 362
```

```
BOOKING SUCCEEDED:
PNR Number = 8
From Station = Delhi
To Station = Mumbai
Travel Date = 22/May/2021
Travel Class = Second Sitting
 : Mode: Sitting
 : Comfort: Non-AC
 : Bunks: 0
 : Luxury: No
Fare = 362
```

The above test application is given as a sample. In addition, you should write your own unit and application tests.