Software Engineering CS20006 -- Theory Assignment O4
Nakul Aggarwal 19CS10044
31 March 2021

ANALYSIS AND DESIGN

HIGH LEVEL DESIGN

Design Principles

- *Flexible And Extensible Design*
    - The design should be flexible. It should be easy to change the changeable parameters like *load factors, base factor, reservation factors etc* for *booking classes*, *concession factors* for various *booking categories*, *eligibility criteria* for the *booking categories* and others.
    - The design should be extensible. It should be easy to add new behaviour to classes.
- *Minimal Design And Maximal Code Reusability*
    - Only the required models and behaviours should be coded. No extra classes should be used.
    - *Less code, less error principle should be followed.*
    - *Reusability of code* should be maximized by used *inheritance* and *templates*.
- *Reliable Design*
    - Every functionality should be working as given in the specifications.
    - *Erroneous input conditions* should be handled with exceptions. System should never be allowed to go into an inconsistent state.
    - All possible errors must be appropriately thrown and caught handled.
    - *Encapsulation* should be maximized.
    - Parameters should be appropriately defaulted wherever possible.
- *Testable Design*
    - Every class should support the *output streaming operator* for checking the intermittent output if needed.
    - Every class should be easily testable.
    - *Equality and inequality relational operators* should be overloaded wherever required to enable easier testing.

Classes

- *Station* -- Every *Station* is identified by its name. Booking is done between any two *Station*s.

- *Railways* -- It has a collection of *Station*s with pairwise distance between *Station*s known a priori (for the default arguements). It has to be implemented as a *singleton class*.

- *Date* -- Any valid date in *DD/MM/YYYY* format is allowed. No same day booking is allowed. Hence date of booking must be later than the date of reservation (current date in the system). Booking for upto one year in advance is allowed.

- *BookingClass* -- There are several *BookingClass*es for travel. 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.
  - *Number 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.
  - *Reservation Charge*: The reservation or booking charge as levied for the BookingClass. This may change from time to time.

- *BookingCategory* -- There are several BookingCategory for travel. Each BookingCategory has the following attributes:
  - *Name*: Name of the BookingCategory
  - *Eligibility*: Eligibility criteria / conditions for the BookingCategory – typically dependent on the Passenger

  Some *BookingCategory*s allow for *Concessional* fare based on the BookingClass and the eligibility of the Passenger. Some BookingCategorys allow for priority (Tatkal) booking on a higher Tatkal fare depending on the BookingClass of travel. New booking categories may be added in future.

- *Passenger* -- A *Passenger* may have the following details:

- *name*: Name of the passenger comprising (input as three separate strings):
  - firstName: Optional if lastName is present
  - middleName: Optional
  - lastName: Optional if firstName is present
- *dateOfBirth*: Date of birth to be used for verification of age and decisions about eligibility for a *BookingCategory*
- *gender*: Gender of the passenger: male or female – to be used for verification of identity and decisions about eligibility for a BookingCategory
- *aadhaar #:* 12-digit Aadhaar Number to be used as a unique ID and input as a string
- *mobile* #: 10-digit Mobile number (optional) and input as a string
- *disability type*: Type of disability (optional). This is used to check eligibility for Divyaang BookingCategory booking.
- *disabilityID* #: Number of the divyangjan ID (optional) and input as a string.

Modelling Sub Types

- *Inlcusion parametric polymorphism* for *Gender, BookingClass, BookingCategory, Divyaang* and *Booking* is to be used.
- *Ad-hoc polymorphism* should be used for *Concessions and Exceptions.*
- *Date, Station, Railways* and *Passenger* classes have no sub types and hence will have no hierarchy.

Interfaces

- Every class should have proper *constructor* and *destructor*. They should be made *private/protected* wherever required.
- If the construction of an object of a class has possibility of exception due to erroneous inputs, the same should be checked in a separate static function before invoking the constructor. No exception should be thrown from a constructor or a destructor.
- *Copy assignment operators* should be *overloaded*, if required, and blocked otherwise.
- Provide output streaming operator for every class to help output process as well as debugging.
- Define *public get methods* for every class to get the private *static/non-static data*, as required.
- *Station* can have *GetName()*, *BookingClass* can have *GetLoadFactor(), GetReservationCharge(), IsSitting()* etc, *BookingCategory* can have *GetName()* etc; and so on public methods.

- Make methods *const* wherever possible.
- Pass parameters *by reference* for *user-defined* types and *by value* for built-in types to the functions.
- Constructors might be kept private and instead *static member functions* can be used in the interface to throw any *exceptions* if needed. Therefore, *CreateDate(), CreateStation(), CreatePassenger()* etc static methods may be used in the interface.
- Methods implementing some specific functionalities related to a class like *IsElligible()* method in *BookingCategory sub-classes, ComputeFare()* in *Booking sub-classes* and *GetAge()* in *Passenger* (among others) may be kept in the interface.

Static Constants

- All the master data given in the specifications should be included in the *static const data members* of appropriate classes.
- Other *static const data members* can be used as *helpers*, like *sDaysInMonths* data member in *Date* to identify the erroneous date formats.
- Classes like *BookingClass*, *BookingCategory*, *Concessions* and *Divyaang* are built entirely from *static data*.

Errors And Exceptions

- All the erroneous conditions as given in the specifications should be handled.
- Classes like *BookingClass*, *BookingCategory*, *Concession* and *Divyaang* are constructed entirely from *static data* and hence can be assumed to be free of errors.
- Every error must be properly handled and meaningfully reported.
- If there are more than one validation failures, the system should attempt to report as many of them as possible in a single run.
- All validations and reporting should be based on exceptional design clearly separating the normal flow from the exception flow.
- An appropriate hierarchy of exception classes may be designed for the error management.
- In no case, the system may be allowed to go to an inconsistent state and / or crash.

Guidelines

- *Global variables* or *functions* are not used (other than the *friend functions*).
- No *constant* value should be written within the code. All constants should be put in the application as static.

- Pass parameters by value for *built in types* and *by const reference* for *user defined types*.
- Every polymorphic hierarchy must provide a virtual destructor in the base class.
- Constructors and destructors should never *throw* an exception.
- *Virtual functions* should not be called in the constructors of base classes.
- *C++ style casting* should be preferred.
- Encapsulation should be maximized.
- Indentation of the code must be proper and standard.
- Variable names should be indicative of their semantics.

Let us now extend these details to formulate a Low Level Design for the Booking Application.

LOW LEVEL DESIGN

Station

Non Static Data Members

- *Station* class has a *non-static const private data member* -- *Station::name_*
- This stores the name of an instance of a *Station* class
- It is of *string* data type. *string* is a class declared in <*string*> header file
- It is of *const* type because the name of a *Station* cannot be changed
- It is kept *private* for maximizing encapsulation
- Any string-value that contains at least one character other than *whitespace* is allowed for *Station::name_*

Constructors

- *Station* has a *private constructor* that takes a *string* as an argument and constructs a new instance of *Station* class.
- *Signature:* `Station::Station(const std::__cxx11::string &)`
- *string* object is *passed by reference* to avoid copying overheads. It is passed as a *const* reference so as to not permit the change of the *actual parameter.*
- Constructors do not have a return type and neither can they be *const* methods.
- The constructor is declared in private scope because *we need to control the values that are passed to the constructor as argument*.

  A *Station* object cannot have an *empty* name. An *exception* must be thrown when an erroneous/invalid value is passed to be set as the name of the *Station* object. *Constructors* however should never throw exceptions because they might lead to *inconsistent object states*. So the constructor must be declared in the *private* access-specifier and another *static member function* should be used in the public interface to take filter the error-free values that are passed to the private constructor (*discussed later*).

- *Station* has a *public copy constructor* that takes a *Station* object as argument and constructs a new *Station* object with the same state.
- *Signature:* `Station::Station(const Station &)`
- *Station* object must be *passed as a const reference* to avoid an infinite loop stuck in the copy constructor. If the parameter is passed by value, the passed object will have to be copied to the local arguements and will in turn call the copy constructor; leading to an indefinite sequence of calls to the copy constructor.
- The *source Station* instance is passed as a *const* reference to ensure that the state of the *actual parameter* stays intact.
- Constructors do not have a return type and neither can they be *const* methods.

Operator Overloading

- The *copy-assignment operator '='* for *Station* class is blocked (declared in *private* access-specifier)
- This is done because of the *constness* of *Station::name_* data member which should not allow the value of *Station::name_* to be re-assigned/modified.

- The *equality check relational operator '=='* is overloaded to compare two instances of *Station* class for equality
- *Signature:* `bool Station::operator==(const Station &) const`
- *Return type:* '==' operator can only return one of the two values -- *true* or *false*. Therefore, the return type *bool* is appropriate
- *Arguements:* The operator method has one argument. The *Station* instance on the RHS of the operator '==' is passed as the argument to the respective operator method.
  It is passed *by reference* to avoid calling the *copy constructor* (and hence saving both memory and CPU cycles). It is passed as a *const* reference because equality checking should not change the state (value of *Station::name_*) of the passed *Station* object (*actual parameter*).
- *Constant Method:* The instance on the LHS of the operator '==' is the *Station* object as a member of which the operator method is called. In order to ensure the *constness* (because equality checking should not change its state -- value of *Station::name_*) of the *Station* object on the LHS of the operator, *const* keyword is used on the extreme right end of the signature.
- *Return Value:* The operator method returns *true* if both the *Station* objects have the same name (value of *Station::name_* data member) and *false* otherwise.

- The *inequality check relational operator '!='* is overloaded to compare two instances of *Station* class for inequality
- *Signature:* `bool Station::operator!=(const Station &) const`
- *Return type:* '!=' operator can only return one of the two values -- *true* or *false*. Therefore, the return type *bool* is appropriate
- *Arguements:* The operator method has one argument. The *Station* instance on the RHS of the operator '!=' is passed as the argument to the respective operator method.
  It is passed *by reference* to avoid calling the *copy constructor* (and hence saving both memory and CPU cycles). It is passed as a *const* reference because inequality checking should not change the state (value of *Station::name_*) of the passed *Station* object (*actual parameter*).

- *Constant Method:* The instance on the LHS of the operator '!=' is the *Station* object as a member of which the operator method is called. In order to ensure the *constness* (because inequality checking should not change its state -- value of *Station::name_*) of the *Station* object on the LHS of the operator, *const* keyword is used on the extreme right end of the signature.
- *Return Value:* The operator method returns *false* if both the *Station* objects have the same name (value of *Station::name_* data member) and *true* otherwise.

- The *output streaming operator '<<'* is overloading to print the name of the *Station* object on the console.
- *Signature:*
```
std::ostream &operator<<(std::ostream &, const Station &)
```
- *Friendship:* This method is not a member function but a *friend* function that is declared in the global scope. *Friend functions* can private (and protected) data members of the class in which they are declared as a *friend.*
  Friendship is realized by the keyword *friend* which is precedes the signature of the function when it is declared inside a class.
- *Arguments:* This operator function has two parameters. The first one is of output stream object *std::ostream* type. The standard objects like *std::cout* or *std::cerr* are of this type. The LHS entity in the expression "*cout << x*" (that is, *cout*) is passed as the first parameter. This parameter is passed as a *non-const reference*; *non-const* because the state of the output-stream should be changed by the function when it inserts *formatted output* to that stream. The parameter is *passed by reference* because it is imperitive to output the content/message on the same output-stream on which it is intended to by the expression "*cout << x*" in the caller function.
  The second arguement is of user-defined type *Station* which is actually the RHS entity in the same expression. It is *passed by reference* to avoid copying overheads. It is passed as a *const* reference to ensure that the state of the *actual parameter* is not changed by this function.
- *Return Type:* The return type is *std::ostream*. The same stream that is passed as an arguement is returned to enable *chained output streaming* (like "*cout << x << y << z*"). To ensure that the same stream is returned, the *return is by reference*. It is a ret*urn by non-const-reference* because the state of the returned output stream object might have to change in the caller function in case there is another instruction for formatted output chained with the former one. Return-by-reference is possible here because the returned output stream object is not a local object but rather the same object that was passed-by-reference as the first parameter.

Static Member Functions

- *Station* has a *static public member function* that acts as an interface between the private constructor (the one other than the *copy constructor*) and the scope outside of the class. It handles *erroneous values* of the arguements and throws exceptions accordingly.
- *Signature:*
  `static Station Station::CreateStation(const std::__cxx11::string &)`
- *Arguments:* *string* object is *passed by reference* to avoid copying overheads. It is passed as a *const* reference so as to not permit the change of the *actual parameter.*
- *Exceptions:* The method is responsible for ensuring that the constructor of *Station* class is called only if the value of the arguement is not of undesired format.
  (*Undesired Format:* The name either has no characters, or has only *whitespace* characters. In other words, it must have at least one character other than *whitespace.*)
  Therefore the passed *string* object is checked if it matches this criteria. If yes, the *Station* constructor is called and the newly-constructed object is returned. Otherwise, a *Bad_Station* exception is thrown.
- *Return Type:* The method needs to return a *Station* object (if it does not throw an exception) and hence the return type. The return is *by value* because the constructed object is a *local variable*.

Non-Static Member Functions

- *Station* has a *non-static public member function GetName* that returns the value of the attribute *Station::name_*.
- *Signature:* `std::__cxx11::string Station::GetName() const`
- *Constant Method:* The *const* keyword on the extreme right of the signature ensures that the state of the object as a member of which this method was called stays intact.
- *Inlining:* It is made an *inline* function.
- *Return type:* The data type of the data member *Station::name_* is *string* and hence the return type.

- *Station* has a *non-static public member function GetDistance* that returns the distance of a *Station* object from the *Station* that is passed as argument.
- *Signature:* `unsigned int Station::GetDistance(const Station &) const`
- *Arguements:* The *Station* object from which the distance is to be returned is passed as the only parameter. It is *passed by reference* to avoid copying overhead. It is passed as a *const* reference to ensure that the state of the *actual parameter* stays intact.

- *Constant Method:* The *const* keyword on the extreme right of the signature ensures that the state of the object as a member of which this method was called stays intact.
- *Return type:* Distance will always be non-negative and in this design it is rounded off to the nearest integer, so this return data type is appropriate.
- *Sub-Routine:* This method uses the *non-static public member function Railways::GetDistance* of *Railways* class that searches in the *map* (*Railways::distStations_*) attribute of the *singleton Railways instance* for the distance between the two terminal stations. *Station::GetDistance* returns the value returned by this method.
- *Exceptions: Railways::GetDistance* throws a *Bad_Railways_Distance exception* if the map does not contain the distance between the two terminal stations. This exception, in this case, will be *catch*-ed in *Station::GetDistance* and *re-thrown*.

Destructor

- *Station* has a *public destructor*.
- *Signature:* `Station::~Station()`
- Destructors neither have any arguments, nor can they be *const* methods, nor can they have any return type.

Railways

Non Static Data Members

- *Railways* class has a *non-static const private data member* -- *Railways::stations_*
  It is a *const vector* of *Station* objects. This *contains* all the *Station* objects that are a part of the *Railways* network.
  In the design it is rightly assumed that no new *Station* can be added to a *Railways* and neither can the existing ones be removed. Therefore, it is apt to implement the *collection* of all the *Station* objects constituting a particular *Railways* instance as a *const* vector
  ```
  const std::vector<Station> Railways::stations_
  ```
  (NOTE : vector is a standard template provided by the header file <vector>. Here the template arguement is chosen to be Station.)

- The second one is a *const map* which is again a standard template provided by the header file *<map>*. It is actually a standard container made up of *(key, value)* pairs, which can be retrieved based on a key, in logarithmic time. This map is used to store the distance (in kilo-meters) against a pair of stations.
  ```
  const std::map<std::pair<std::__cxx11::string,std::__cxx11::string>,
  unsigned int> Railways::distStations_
  ```

  About key: The data structure/type for the *key* of a map is passed as its first template parameter. The key chosen here is a *pair* which is another standard template class. The *pair* container is a simple container defined in *<utility>* header consisting of two data elements or objects.
  Here a *homogenous pair* of data type *string* is used (both the *template parameters* of *pair* are *string*). A pair of strings depicts a pair of names of the two terminal stations.

  About value: The data structure/type for the *value* of a map is passed as its second template parameter. In this case, the *value* corresponding to a *key* in the map signifies the distance (in kilometers) between the stations that have their names as the first and the second element of the *key* that is implemented by a *pair* container. The distances are taken as rounded off integers and therefore the data type of the second template parameter is chosen to be of built-in type *unsigned int* (distance is always non-negative).

  For the design it was rightly assumed that for a *Railways* instance, neither can any new stations be added, nor can the existing ones be removed, nor can the distance between any of the existing stations change. Under this assumption, it is apt to make *Railways::distStations_* a *const* data member.

Singleton Class

- The *Railways* class is implemented as a *singleton* class. That is, at most one instance of the *Railways* class can be constructed.
  This is realised as a *Meyer's implementation of a singleton*.
- Therefore, the constructor is kept *private* and a public member function *Railways::SpecialRailways* is used in the interface to *get* the *singleton instance of the Railways class*.

Constructor

- The class has a *private constructor* that takes two arguements -- a *vector* and a *map*.
- *Signature:*
  ```
  Railways::Railways (const std::vector<Station> &, const
  std::map<std::pair<std::__cxx11::string, std::__cxx11::string>,
  unsigned int> &)
  ```
- A constructor does not have a return type and neither can it be a *const* method.
- The first parameter is used to initialize the data member *Railways::stations_* in the initializer list (because this data member is a *const*).
  It is *passed as a reference* to avoid expensive copying overhead. It is passed as a *const* reference to ensure that the value of the *actual parameter* stays intact.
- The second parameter is used to initialize the data member *Railways::distStations_* in the initializer list (because this data member is a 'const').
  It is *passed as a reference* to avoid expensive copying overhead. It is passed as a *const* reference to ensure that the value of the *actual parameter* stays intact.
- It is important to keep the constructor private otherwise an instance of the *Railways* class can be constructed anywhere, any number of times thus violating its *singleton* property.

Static Member Functions

- The class has a *public static member function* that acts as an interface between the private constructor and the scope outside this class.
- *Signature:*
  ```
  static const Railways &Railways::SpecialRailways(const
  std::vector<Station> & = {
  Station::CreateStation("Mumbai"),
  Station::CreateStation("Delhi"),
  Station::CreateStation("Bangalore"),
  Station::CreateStation("Kolkata"),
  Station::CreateStation("Chennai")},
  const std::map<std::pair<std::__cxx11::string,
  std::__cxx11::string>, unsigned int> & =
  ```

```
{{{"Delhi", "Mumbai"}, 1447},
{{"Bangalore", "Mumbai"}, 981},
{{"Kolkata", "Mumbai"}, 2014},
{{"Chennai", "Mumbai"}, 1338},
{{"Bangalore", "Delhi"}, 2150},
{{"Delhi", "Kolkata"}, 1472},
{{"Chennai", "Delhi"}, 2180},
{{"Bangalore", "Kolkata"}, 1871},
{{"Bangalore", "Chennai"}, 350},
{{"Chennai", "Kolkata"}, 1659}})
```

- This method has the same arguements as the constructor, except that here default values of the arguements are also given. When *Railways::SpecialRailways* is called for the first time, the singleton object is constructed and every next call to *Railways::SpecialRailways* results with the same object being returned.
  Providing flexibility to construct the singleton instance based on some custom attributes is important because the *master data* given in the *specifications* realizes a *miniature railways network* with only 5 stations, which is not a very realistic situation.
- A *static Railways object* is constructed in this method. This is a singleton because static storage duration for a function local means that only one instance of that local exists in the program. This singleton instance is returned everytime a "*Railways::SpecialRailways()*" call is made. The *return is by reference* because *Railways* does not have a *copy constructor*; moreover it is a singleton class. It is returned as a *const* reference because the singleton *Railways* object that is returned to the caller function must be treated as a constant object.
  Note that return by reference is possible here because the returned object is not a *local non-static object* but rather a *local static object* which is not allocated on the stack frame.
- The default parameters are selected based on the *Master Data* given in the *specifications*. The default arguements realize a small network of Indian Railways with 5 stations. The first parameter (*const std::vector<Station> &*) stores the *Station* objects constituting the *Railways* network and the second parameter (*const std::map<std::pair<std::__cxx11::string, std::__cxx11::string>, unsigned int> &*) stores the pairwise distance between all the stations in the form of a *map*.
- The first and second parameters are *passed as a reference* to avoid expensive copying overheads. They are passed as a *const* reference to ensure that the value of the *actual arguments* stays intact.
- These two parameters are ultimately used in constructing the *singleton object* of the class.
- *Exceptions:* This member function is not only responsible to *control the number of calls to the constructor* but is also responsible in *validating the passed/default*

arguments. The values of the data members *Railways::stations_* and *Railways::distStations_* have to follow some conditions. If the passed/default values (that might eventually be used to call the constructor) do not satisfy at least one of them, a suitable *exception* is thrown.

- *Bad_Railways_NotEnoughStations: Railways* needs to have at least *two* stations in its network.
- *Bad_Railways_DuplicateStations:* Same *Station* object is present more than once in the vector that is the *first argument*
- *Bad_Railways_DistBwSameStationsDefined:* There exists a *key* (of *pair<string, string>* type) in the *map* as the *second argument* in which both the first and the second element have the same values
- *Bad_Railways_NoDefinition:* There exists a pair of names of two distinct *Station* objects in the first argument, that is not present as a *key* in the *map* of the *second argument* (neither of the two ordered pairs is present)
- *Bad_Railways_RepeatedDefinition:* There exists a pair of names of two distinct *Station* objects in the first argument, for *both* the ordered pairs of which a *key* is present in the *map* of the *second argument*. (The definition is considered symmetric - so only one direction should given.)

Non-Static Member Functions

- The class has a *public non-static member function* that takes two *Station* objects as input and returns the distance between them (in kilometers).
- *Signature:* `unsigned int Railways::GetDistance(const Station &, const Station &) const`
- *Arguements:* It has two parameters, both of which are of the user defined type *Station*. Both of them are *passed by reference* to avoid any copying overheads. Besides, they are passed as *const* reference so that the state of the *actual parameters* stays intact. The method is a *mathematically-symmetric* function.
- *Return type:* The distance between any two stations is realized as an *unsigned int* in *Railways::distStations_* (second template arguement). Hence, the return type is *unsigned int*.
- This is a *const* method. This is necessary because the only way the singleton object of *Railways* class can be obtained is by the call to the function *Railways::SpecialRailways*, that returns a *const reference* to the singleton object. On a *const* instance of a class, only *const* member functions can be called.
- *Exceptions:* If none of the two ordered pairs corresponding to the names (*Station::name_*) of the two *Station* objects passed as arguments is present as a *key* in the *map Railways::distStations_* of the singleton *Railways* instance, a *Bad_Railways_Distance exception* is thrown.

Operator Overloading

- The *copy-assignment operator '='* for *Railways* class is blocked (declared in *private* access-specifier)
- This is done because of the *constness* of the singleton instance of *Railways* class.

- The *output streaming operator '<<'* is overloading to print all the details (all *Station* objects and *pairwise distances* between all of them) of the *Railways* object on the console.
- *Signature:*
  `std::ostream &operator<<(std::ostream &, const Railways &)`
- *Friendship:* This method is not a member function but a *friend* function that is declared in the global scope. *Friend functions* can private (and protected) data members of the class in which they are declared as a *friend.*
  Friendship is realized by the keyword *friend* which is precedes the signature of the function when it is declared inside a class.
- *Arguments:* This operator function has two parameters. The first one is of output stream object *std::ostream* type. The standard objects like *std::cout* or *std::cerr* are of this type. The LHS entity in the expression "*cout << x*" (that is, *cout*) is passed as the first parameter. This parameter is passed as a *non-const reference*; *non-const* because the state of the output-stream should be changed by the function when it inserts *formatted output* to that stream. The parameter is *passed by reference* because it is imperitive to output the content/message on the same output-stream on which it is intended to by the expression "*cout << x*" in the caller function.
  The second argument is of *user defined type Railways*. It is *passed by reference* because there does not exist any copy constructor for this class (moreover *Railways* is a singleton class). The parameter is passed as a *const* reference because the only way the singleton object of *Railways* class can be obtained is by the call to the function *Railways::SpecialRailways*, that returns a *const reference* to the singleton object. *const* reference cannot be converted into a *non-const* reference.
- *Return Type:* The return type is *std::ostream*. The same stream that is passed as an arguement is returned to enable *chained output streaming* (like "*cout << x << y << z*"). To ensure that the same stream is returned, the *return is by reference*. It is a ret*urn by non-const-reference* because the state of the returned output stream object might have to change in the caller function in case there is another instruction for formatted output chained with the former one. Return-by-reference is possible here because the returned output stream object is not a local object but rather the same object that was passed-by-reference as the first parameter.

Destructor

- *Railways* has a *private* destructor
- *Signature:* `Railways::~Railways()`
- Destructors do not have any arguments or return type and neither can they be *const* methods.
- *Railways* class is a *singleton* class and the only instance of this class is of *static* type. When a variable is declared as static, space for it gets allocated for the lifetime of the program. Therefore, the singleton *Railways* instance is not *destructed* until the program gets terminated.
- It is a good idea to make a destructor of a *singleton* class private because then the client/application code won't call the destructor by accident. Calling the destructor would cause the singleton to fail for all applications in the project as the instance would become invalid.

Date

Non Static Data Members

- *Date* class has three *non-static private data members* -- *Date::date_, Date::month_, Date::year_*.
- These data members respectively store the date/day, month and year corresponding to any date on the calendar.
- All three of them are of built-in *unsigned int* type. This choice of data type is apt because none of them can be negative.

Static Data Members

- *Date* class has a *private static const vector* of size 12 that stores the number of days in each month of a year (the number of days in the month of *February* are arbitrarily chosen as 28, the case of *leap years* will be handled separately).
- The template arguement is chosen to be of *unsigned int* built-in type because the number of days in a month is strictly positive.
- Certainly, it must be a *const* data member because the number of days in a month cannot be changed.
- This static data member comes in very handy in validating the passed arguments for semantic accuracy.
  ```
  static const std::vector<std::size_t> Date::sDaysInMonths
  ```

- *Date* class has a *private static const* data member that stores the minimum value of year in which any *Date* can be constructed.
  ```
  static const unsigned int Date::sMinValidYear
  ```
- *Date* class has a *private static const* data member that stores the maximum value of year in which any *Date* can be constructed.
  ```
  static const unsigned int Date::sMaxValidYear
  ```
- Both of them are of *unsigned int* data type.

- *Date* class has a *public static const vector* that stores the names of the 12 months of the year.
- The template argument is chosen to be *string*.
  ```
  static const std::vector<std::__cxx11::string> Date::sMonthNames
  ```
- *Date* class has a *public static const vector* that stores the names of the 7 days of a week.
- The template argument is chosen to be *string*.
  ```
  static const std::vector<std::__cxx11::string> Date::sDayNames
  ```

Constructors

- *Date* has a *private constructor* that takes a three arguments of *unsigned int* type and constructs a new instance of *Date* class with date, month, year respectively equal to the arguments.
- *Signature:* `Date::Date(unsigned int, unsigned int, unsigned int)`
- Constructors do not have a return type and neither can they be *const* methods.
- The constructor is declared in private scope because *we need to control the values that are passed to the constructor as arguments*.
  Any triplet of unsigned integers will have to be first checked for a number of possible errors that can make the triplet invalid as far as the construction of a *Date* is concerned. *Constructors* however should never throw exceptions because they might lead to *inconsistent object states*. So the constructor must be declared in the *private* access-specifier and another *static member function* should be used in the public interface to take filter the error-free values that are passed to the private constructor (*discussed later*).

- *Date* has a *public copy constructor* that takes a *Date* object as argument and constructs a new *Date* object with the same state.
- *Signature:* `Date::Date(const Date &)`
- *Date* object must be *passed as a const reference* to avoid an infinite loop stuck in the copy constructor. If the parameter is passed by value, the passed object will have to be copied to the local arguements and will in turn call the copy constructor; leading to an indefinite sequence of calls to the copy constructor.
- The *source Date* instance is passed as a *const* reference to ensure that the state of the *actual parameter* stays intact.
- Constructors do not have a return type and neither can they be *const* methods.

Operator Overloading

- The *copy-assignment operator '='* is overloaded for *Date* class.
- *Signature:* `Date &Date::operator=(const Date &)`
- *Return type:* The *Date* instance as a member of which this operator method is called is *returned by reference* to enable *changed assignment operations* (like *"x = y = z"*. Therefore the return type is *Date*. It is returned by *reference* to avoid copy overheads.
- *Arguments:* The argument is the *source Date object* that is on the RHS of the operator in the caller function. It is passed *by reference* to avoid copy overheads. It is passed as a *const* reference to ensure that the state of the actual argument stays intact.

- The *equality check relational operator '=='* is overloaded to compare two instances of *Date* class for equality

- *Signature:* `bool Date::operator==(const Date &) const`
- *Return type:* '==' operator can only return one of the two values -- *true* or *false*. Therefore, the return type *bool* is appropriate
- *Arguements:* The operator method has one argument. The *Date* instance on the RHS of the operator '==' is passed as the argument to the respective operator method.
  It is passed *by reference* to avoid calling the *copy constructor* (and hence saving both memory and CPU cycles). It is passed as a *const* reference because equality checking should not change the state of the passed *Passenger* object (*actual parameter*).
- *Constant Method:* The instance on the LHS of the operator '==' is the *Date* object as a member of which the operator method is called. In order to ensure the *constness* (because equality checking should not change its state -- value of *Date ::date_*, *Date::month_* or *Date::year_*) of the *Date* object on the LHS of the operator, *const* keyword is used on the extreme right end of the signature.
- *Return Value:* The operator method returns *true* if both the *Date* objects have the same date, month and year and *false* otherwise.

- The *inequality check relational operator '!='* is overloaded to compare two instances of *Date* class for inequality
- *Signature:* `bool Date::operator!=(const Date &) const`
- *Return type:* '!=' operator can only return one of the two values -- *true* or *false*. Therefore, the return type *bool* is appropriate
- *Arguements:* The operator method has one argument. The *Date* instance on the RHS of the operator '!=' is passed as the argument to the respective operator method.
  It is passed *by reference* to avoid calling the *copy constructor* (and hence saving both memory and CPU cycles). It is passed as a *const* reference because inequality checking should not change the state of the passed *Date* object (*actual parameter*).
- *Constant Method:* The instance on the LHS of the operator '!=' is the *Date* object as a member of which the operator method is called. In order to ensure the *constness* (because inequality checking should not change its state) of the *Date* object on the LHS of the operator, *const* keyword is used on the extreme right end of the signature.
- *Return Value:* The operator method returns *false* if both the *Date* objects have the same date, month and year and *true* otherwise.

- The *output streaming operator '<<'* is overloading to print the date represented by a *Date* object in the format *"DD/MM/YYYY"* where all placeholders are digits.

- *Signature:*
  `std::ostream &operator<<(std::ostream &out, const Date &)`
- *Friendship:* This method is not a member function but a *friend* function that is declared in the global scope. *Friend functions* can private (and protected) data members of the class in which they are declared as a *friend.*
  Friendship is realized by the keyword *friend* which is precedes the signature of the function when it is declared inside a class.
- *Arguments:* This operator function has two parameters. The first one is of output stream object *std::ostream* type. The standard objects like *std::cout* or *std::cerr* are of this type. The LHS entity in the expression "*cout << x"* (that is, *cout*) is passed as the first parameter. This parameter is passed as a *non-const reference*; *non-const* because the state of the output-stream should be changed by the function when it inserts *formatted output* to that stream. The parameter is *passed by reference* because it is imperitive to output the content/message on the same output-stream on which it is intended to by the expression "*cout << x*" in the caller function.
  The second arguement is of user-defined type *Date* which is actually the RHS entity in the same expression. It is *passed by reference* to avoid copying overheads. It is passed as a *const* reference to ensure that the state of the *actual parameter* is not changed by this function.
- *Return Type:* The return type is *std::ostream*. The same stream that is passed as an arguement is returned to enable *chained output streaming* (like "*cout << x << y << z*"). To ensure that the same stream is returned, the *return is by reference*. It is a ret*urn by non-const-reference* because the state of the returned output stream object might have to change in the caller function in case there is another instruction for formatted output chained with the former one. Return-by-reference is possible here because the returned output stream object is not a local object but rather the same object that was passed-by-reference as the first parameter.

Static Member Functions

- *Date* has a *static public member function* that acts as an interface between the private constructor (the one other than the *copy constructor*) and the scope outside of the class. It handles *erroneous values* of the arguements and throws exceptions accordingly.
- *Signature:*
  `static Date Date::CreateDate(const std::__cxx11::string &)`
- *Arguments:* *string* object is *passed by reference* to avoid copying overheads. It is passed as a *const* reference so as to not permit the change of the *actual parameter.*
- *Exceptions:* The method is responsible for ensuring that to the private constructor of *Date* class only those arguments are passed that *actually*

represent a date on the calendar. The *only* allowed format for the *string representation* of a date is "*DD/MM/YYYY*" (where all of the placeholders are digits and actually represent a valid date). The string parameter has to be checked for the following errors and suitable exceptions are to be thrown.

- *Bad_Date_Format:* The pattern *"[0-9]{2}/[0-9]{2}/[0-9]{4}"* must be matched by the string passed as argument. That means, it must be of "*DD/MM/YYYY*" format where *D, M* and *Y* are digits (non-negative integers). Otherwise, *Bad_Date_Format exception* is thrown.
- *Bad_Date_Year:* The first test is passed. The unsigned integer formed by the last 4 characters *YYYY* in the string must not be less than *Date::sMinValidYear* and must not be more than *Date::sMaxValidYear*. Otherwise, *Bad_Date_Year* exception is thrown.
- *Bad_Date_Month:* The first two test are passed. The unsigned integer formed by the fourth and fifth characters in the string (in the same order) *MM* must not be less than 1 and must not be more than 12. Otherwise, *Bad_Date_Month exception* is thrown.
- *Bad_Date_Day:* The first three tests are passed.
    - The unsigned integer formed by the first two characters in the string (in the same order) *DD* must not be less than 1 and must not be more than 31. Otherwise, *Bad_Date_Day exception* is thrown.
    - If the *month* represented by this date in string format is *February* (MM = 02) then the value of *DD* must not be more than 29. Otherwise, *Bad_Date_Day exception* is thrown.
    - If the *month* represented by this date in string format is *February* (MM = 02) and the year represented is *not* a *leap year* (YYYY is not a multiple of 4) then the value of *DD* must not be more than 28. Otherwise, *Bad_Date_Day exception* is thrown.
    - The value of *DD* must not be more than the number of days in the month *MM* (use *Date::sDaysInMonths*). Otherwise, *Bad_Date_Day exception* is thrown.

If no exception is thrown and the *string* passes all the checks, the *private Date constructor* is called (with the three arguments extracted from the string) and the newly-constructed object is returned.

- *Return Type:* The method needs to return a *Date* object (if it does not throw an exception) and hence the return type. The return is *by value* because the constructed object is a *local variable*.

- There is another *overloaded* member function that takes instead of a string, three arguments of *unsigned int built-in data type*, representative of *day, month* and *year*.

- *Signature:* `static Date Date::CreateDate(unsigned int = 1U, unsigned int = 1U, unsigned int = Date::sMinValidYear)`
- *Exceptions:* The *error/exception handling* is exactly the same as in the former member function, the only difference is that here the *date format* need not be checked and hence *Bad_Date_Format* exception will never be thrown. Besides, the values of *day, month, year* are already available as *unsigned integers* and no extraction has to be done.
  Except this, the various scenarios of errors that are checked and the corresponding exceptions that are thrown are exactly the same.
- *Return Type:* The method needs to return a *Date* object (if it does not throw an exception) and hence the return type. The return is *by value* because the constructed object is a *local variable*.

- *Date* has a *public static data member* that returns the *current/present date*.
- *Signature:* `static Date Date::GetTodaysDate()`
- *Return type:* The present / today's date is returned as an instance of *Date* class and hence the return type. Return is *by value*.

Non-Static Member Functions

- *Date* has a *public non-static data member* that returns *true* if the *Date* object represents a date that falls in a leap year and *false* otherwise.
- *Signature:* `bool Date::IsLeapYear() const`
- *Return type:* The method can only return one of the two values -- *true* or *false*. Therefore, the return type *bool* is appropriate.
- *Constant Method: const* keyword on the extreme right of the signature ensures that the state of the object as a member of which this method was called stays intact.
- *Inlining:* This member function is implemented as an *inline function*.

- *Date* has a *public non-static data member* that returns *true* if it occurs after the date passed as the argument, and *false* otherwise.
- *Signature:* `bool Date::IsAfter(const Date &) const`
- *Arguements:* The *Date* parameter is *passed by reference* to avoid copying overheads. It is passed as a *const* reference so that the state of the *actual parameter* stays intact.
- *Constant Method: const* keyword on the extreme right of the signature ensures that the state of the object as a member of which this method was called stays intact.
- *Return type:* The method can only return one of the two values -- *true* or *false*. Therefore, the return type *bool* is appropriate.

- *Date* class has a *public non-static data member* that returns the difference between two *Date* objects in *years*. Note that it should return a *positive difference* if the date passed as the argument occurs before the date as a member of which this method is called; and *negative/zero* otherwise.
- *Signature:* `int Date::GetDifferenceInYears(const Date &) const`
- *Arguements:* The *Date* parameter is *passed by reference* to avoid copying overheads. It is passed as a *const* reference so that the state of the *actual parameter* stays intact.
- *Constant Method: const* keyword on the extreme right of the signature ensures that the state of the object as a member of which this method was called stays intact.
- *Return type:* The difference is computed not only considering the difference in the years of the two dates but also considering the differences in months and days. Therefore, the precise difference between the dates in years should have a fractional part. The *rounded off* difference is returned and hence the return type. Note that the difference can also be negative and therefore the return type is *int*.

- *Date* class has a *public non-static data member* that returns the difference between two *Date* objects in *days*. Note that it should return a *positive difference* if the date passed as the argument occurs before the date as a member of which this method is called; and *negative/zero* otherwise.
- *Signature:* `int Date::GetDifferenceInDays(const Date &) const`
- *Arguements:* The *Date* parameter is *passed by reference* to avoid copying overheads. It is passed as a *const* reference so that the state of the *actual parameter* stays intact.
- *Constant Method: const* keyword on the extreme right of the signature ensures that the state of the object as a member of which this method was called stays intact.
- *Return type:* Since in the application we nowhere are dealing with the aspects related to *time* and *time difference*, the difference between two *Date*s in days will always be an integer. Note that the difference can also be negative and therefore the return type is *int*.

Destructor

- *Date* has a *public destructor*.
- *Signature:* `Date::~Date()`
- Destructors neither have any arguments, nor can they be *const* methods, nor can they have any return type.

Passenger

Non Static Data Members

Data Member	*Data Type*	*Purpose*
*Passenger::firstName_*	*const string*	stores the first name
*Passenger::middleName_*	*const string*	stores the middle name
*Passenger::lastName_*	*const string*	stores the last name
*Passenger::dateOfBirth_*	*const Date*	stores the date of birth
*Passenger::gender_*	*const Gender &*	stores the gender
*Passenger::adhaarNumber_*	*const string*	stores the adhaar number
*Passenger::mobileNumber_*	*string*	stores the mobile number
*Passenger::disabilityType_*	*const Divyaang* const*	stores the disability type
*Passenger::disabilityID_*	*const string*	stores the disability ID

All the above *non-static data members* (with the exception of *Passenger::mobileNumber_*) are *const* data members. It is obvious that for any *Passenger*, his/her first name, middle name, last name, date of birth, gender, adhaar card number, disability type and disability ID cannot change.
However it might happen that some *optional data members* (like *Passenger::middleName_* and *Passenger::disabilityID_*) are not *set* at the time of *instantiation* but need to be provided for an already constructed *Passenger* object. This functionality is *not* implemented in the design for simplicity.

*Passenger::gender_* data member is a *reference* because the polymorphic hierarchy of *Gender* rooted at *Gender* has a *derived template class* which realizes *singleton class(es)*. Since the *singleton instance* of any sub type of *Gender* (*Gender::Male* or *Gender::Female*) is available only through a *public static member function* that returns a const-reference, the data type of the *Passenger::gender_* data member has to be a *const reference*. `const Gender &Passenger::gender_`
*Passenger::disabilityType_* data member is a *pointer* because it is optional and hence should be *nullable*. In fact, it is a *const* pointer because every *static sub-type* of *Divyaang* class is implemented as a *singleton class*. Since the *singleton instance* of any sub-type is available through *public static member function* that returns a

const-reference, its address has to be a pointer to a *const* object.

```
const Divyaang *const Passenger::disabilityType_
```

All of these *non-static data members* are kept *private* to maximize encapsulation.

Constructors

- *Passenger* has a *private constructor* that takes multiple arguments and constructs a new instance of *Passenger* class with those arguments assigned to the appropriate data members.
- *Signature:*
```
Passenger::Passenger(const Date &, const Gender &, const
std::__cxx11::string &, const std::__cxx11::string &, const
std::__cxx11::string &, const std::__cxx11::string &, const
std::__cxx11::string &,const Divyaang*,const std::__cxx11::string &)
```
- Constructors do not have a return type and neither can they be *const* methods.
- The constructor is declared in private scope because *we need to control the values that are passed to the constructor as arguments*.
  Any groups of arguements will have to be first checked for a number of possible errors that can make the arguments invalid as far as the construction of a *Passenger* object is concerned. *Constructors* however should never throw exceptions because they might lead to *inconsistent object states*. So the constructor must be declared in the *private* access-specifier and another *static member function* should be used in the public interface to filter the error-free values that are passed to the private constructor (*discussed later*).
- *Arguments:* The constructor has 9 arguments, as many as the *non-static* data members. These arguments are used to initialize the values of the data members *Passenger::dateOfBirth_, Passenger::gender_, Passenger::adhaarNumber_, Passenger::firstName_, Passenger::middleName_, Passenger::lastName_, Passenger::mobileNumber_,Passenger::disabilityType_, Passenger::disabilityID_* in the same order. All the arguments (except *"const Divyaang *"*) are instances of a *class* and are hence *passed by reference* to avoid copying overheads. Besides they are passed as *const* references to ensure that the values/states of the *actual parameters* stay intact.
  Second last argument is passed as a *const* pointer because it points to the *singleton* instance of a class of *DivyaangTypes* template, which is only available by a *static member function* that returns it as a *const reference*.
  (More on *Divyaang* later)

- *Passenger* has a *public copy constructor* that takes a *Passenger* object as argument and constructs a new *Passenger* object with the same state (exactly same values for all the *non-static* data members)

- *Signature:* `Passenger::Passenger(const Passenger &)`
- *Passenger* object must be *passed as a const reference* to avoid an infinite loop stuck in the copy constructor. If the parameter is passed by value, the passed object will have to be copied to the local arguements and will in turn call the copy constructor; leading to an indefinite sequence of calls to the copy constructor.
- The *source Passenger* instance is passed as a *const* reference to ensure that the state of the *actual parameter* stays intact.
- Constructors do not have a return type and neither can they be *const* methods.

Operator Overloading

- The *copy-assignment operator* '=' for *Passenger* class is blocked (declared in *private* access-specifier)
- This is done because of the *constness* of almost all the *non-static attributes* of a *Passenger* object. In this case, it should not be possible to change any of those data members.

- The *output streaming operator* '<<' is overloading to print the values of all the attributes (only those that are specified, i.e, non-empty) of a *Passenger* object in a neat format.
- *Signature:*
  `std::ostream &operator<<(std::ostream &, const Passenger &)`
- *Friendship:* This method is not a member function but a *friend* function that is declared in the global scope. *Friend functions* can private (and protected) data members of the class in which they are declared as a *friend.*
  Friendship is realized by the keyword *friend* which is precedes the signature of the function when it is declared inside a class.
- *Arguments:* This operator function has two parameters. The first one is of output stream object *std::ostream* type. The standard objects like *std::cout* or *std::cerr* are of this type. The LHS entity in the expression "*cout << x"* (that is, *cout*) is passed as the first parameter. This parameter is passed as a *non-const reference*; *non-const* because the state of the output-stream should be changed by the function when it inserts *formatted output* to that stream. The parameter is *passed by reference* because it is imperitive to output the content/message on the same output-stream on which it is intended to by the expression "*cout << x*" in the caller function.
  The second arguement is of user-defined type *Passenger* which is actually the RHS entity in the same expression. It is *passed by reference* to avoid copying overheads. It is passed as a *const* reference to ensure that the state of the *actual parameter* is not changed by this function.

- *Return Type:* The return type is *std::ostream*. The same stream that is passed as an arguement is returned to enable *chained output streaming* (like "*cout* << *x* << *y* << *z*"). To ensure that the same stream is returned, the *return is by reference*. It is a return *by non-const-reference* because the state of the returned output stream object might have to change in the caller function in case there is another instruction for formatted output chained with the former one. Return-by-reference is possible here because the returned output stream object is not a local object but rather the same object that was passed-by-reference as the first parameter.

Static Member Functions

- *Passenger* has a *static public member function* that acts as an interface between the private constructor (the one other than the *copy constructor*) and the scope outside of the class. It handles *erroneous values* of the arguements and throws exceptions accordingly.
- *Signature:*

```
static Passenger Passenger::CreatePassenger(const Date &, const
std::__cxx11::string &, const std::__cxx11::string &, const
std::__cxx11::string & = "", const std::__cxx11::string & = "",
const std::__cxx11::string & = "", const std::__cxx11::string & =
"", const Divyaang * = NULL, const std::__cxx11::string & = "")
```

- *Arguments:* The constructor has 9 arguments, as many as the *non-static* data members. These arguments are used to initialize the values of the data members *Passenger::dateOfBirth_, Passenger::gender_, Passenger::adhaarNumber_, Passenger::firstName_, Passenger::middleName_, Passenger::lastName_, Passenger::mobileNumber_,Passenger::disabilityType_, Passenger::disabilityID_* in the same order. All the arguments (except *"const Divyaang *"*) are instances of a *class* and are hence *passed by reference* to avoid copying overheads. Besides they are passed as *const* references to ensure that the values/states of the *actual parameters* stay intact.
  Second last argument is passed as a *const* pointer because it points to the *singleton* instance of a class of *DivyaangTypes* template, which is only available by a *static member function* that returns it as a *const reference*.
- *Default parameters:* The arguments that are used to initialize the optional (or possibly optional) attributes of a *Passenger* are kept together on the right extreme so that *default values* can be defined for all of them. In the design, an empty string (string of length zero) is treated as an *unspecified* value for *string* data types and a *NULL pointer* is treated as an *unspecified* value for pointer types. Hence, the default values are chosen to be *empty strings* and *NULL* accordingly.

Consequently, for the *optional* attributes of any *Passenger* object either a valid value of proper format is stored or an empty string (*NULL* pointer in case of *Passenger::disabilityType_*) is stored.

- *Exceptions:* The method is responsible for ensuring that the private constructor of *Passenger* class is called with the arguments only when each one of these arguments is valid. The arguments are checked for accuracy through the following tests that return a suitable *exception* if the test fails.
    - *Bad_Passenger_Name:* If neither the first name nor the last name are specified (both are empty strings), throw *Bad_Passenger_Name exception*. Otherwise, proceed.
    - *Bad_Passenger_AdhaarNumber:* If either the length of the adhaar number is not equal to 12 or the adhaar number consists of at least one non-digit, throw *Bad_Passenger_AdhaarNumber exception.* Otherwise, proceed.
    - *Bad_Passenger_MobileNumber:* If mobile number is not specified (empty string), then proceed. Else, if either the length of the mobile number is not equal to 10 or the mobile number consists of at least one non-digit, throw *Bad_Passenger_MobileNumber exception.* Otherwise, proceed.
    - *Bad_Passenger_DateOfBirth:* If the date of birth is in future (occurs after the current date), throw *Bad_Passenger_DateOfBirth exception.* Otherwise, proceed.
    - *Bad_Passenger_Gender:* If the gender is neither *male* nor *female*, throw *Bad_Passenger_Gender exception.* Otherwise, proceed.
    - *Bad_Passenger_DisabilityType:* If disability type is not specified (*NULL* pointer), then proceed. Else, if the disability type is none of the *Blind, Orthopaedically Handicapped, Cancer Patient, TB Patient*, throw *Bad_Passenger_DisabilityType exception.* Otherwise, proceed.

  If no exception is thrown and all the checks are passed, the *private Passenger constructor* is called with these arguements and the newly-constructed object is returned.
- *Return Type:* The method needs to return a *Passenger* object (if it does not throw an exception) and hence the return type. The return is *by value* because the constructed object is a *local variable*.

Non-Static Member Functions

- *Passenger* has a *public non-static member function* that returns the disability type (value of *Passenger::disabilityType_*) of the *Passenger* object as a member of which the method is called.
- *Signature:*

```
const Divyaang * Passenger::GetDisabilityType() const
```

- *Return type:* The data type of *Passenger::disabilityType_* data member is *const Divyaang ** and hence the return type.
- *Constant Method:* The *const* keyword on the right extreme of the signature ensures that the state of the object as a member of which the method is called stays intact.
- *Inlining:* The method is implemented as an *inline* function.

- *Passenger* has a *public non-static member function* that returns the gender (value of *Passenger::gender_*) of the *Passenger* object as a member of which the method is called.
- *Signature:* `const Gender &Passenger::GetGender() const`
- *Return type:* The data type of *Passenger::gender_* data member is *const Gender &* and hence the return type.
- *Constant Method:* The *const* keyword on the right extreme of the signature ensures that the state of the object as a member of which the method is called stays intact.
- *Inlining:* The method is implemented as an *inline* function.

- *Passenger* has a *public non-static member function* that returns the age of the *Passenger* object as a member of which the method is called.
- *Signature:* `unsigned int Passenger::GetAge() const`
- *Return type:* The age is computed not only considering the difference in the years of the present and the *d.o.b. (date of birth)* dates but also considering the differences in months and days. Therefore, the precise difference between the dates in years should have a fractional part. The difference is finally *rounded off* before returning and and hence the return type.
- *Constant Method:* The *const* keyword on the right extreme of the signature ensures that the state of the object as a member of which the method is called stays intact.
- *Inlining:* The method is implemented as an *inline* function.

Destructor

- *Passenger* has a *public destructor*.
- *Signature:* `Passenger::~Passenger()`
- Destructors neither have any arguments, nor can they be *const* methods, nor can they have any return type.

BookingClass

Inclusion-Parametric Polymorphic Design

- *BookingClass* hierarchy is rooted at an *abstract base class* -- *BookingClass*
- A *template class* *BookingClassTypes* is derived from the abstract base class
- The template of the derived classes is designed in such a way that every instance of the template is a *singleton class*
- As is clear from the *abstractness* of the base class, this hierarchy is a polymorphic hierarchy that allows *dynamic dispatch* of calls to *polymorphic methods*.

Design Of Abstract Base Class -- Tag Types

- The booking application should have *8 booking classes* (*ACFirstClass*, *ExecutiveChairCar*, *AC2Tier*, *FirstClass*, *AC3Tier*, *ACChairCar*, *Sleeper*, *SecondSitting*) and corresponding to each of the booking classes a *user-defined data type* is defined in the *private* access-specifier as a *placeholder* to tag each of the specialized booking class type.

  struct BookingClass::ACFirstClassType
  struct BookingClass::ExecChairCarType
  struct BookingClass::AC2TierType
  struct BookingClass::FirstClassType
  struct BookingClass::AC3TierType
  struct BookingClass::ACChairCarType
  struct BookingClass::SleeperType
  struct BookingClass::SecondSittingType
- In the *public* access-specifier, the *target sub-types* of the *BookingClass* (each one of which is instantiated from the derived template class using the appropriate *tag type* from above as the template argument) are *typedef*-ed so that they can be accessed by using *BookingClass* as the qualifier. For example --

  typedef BookingClassTypes<ACFirstClassType> ACFirstClass

Design Of Abstract Base Class -- Constructor

- *BookingClass* has a *protected* constructor.
- *Signature:* `BookingClass::BookingClass()`
- *BookingClass* is an abstract class and hence cannot be instantiated to construct a *stand-alone* object (though it will be instantiated when a derived class is instantiated). Therefore there is no need of the constructor outside of the hierarchy and hence it can be avoided to be kept as *public*.

  It cannot be kept *private* because it needs to be accessible to the derived class(es) when they are instantiated. Therefore *protected* is chosen as the apt access-specifier.

- Constructors do not have a return type and neither can they be *const* methods.

Design Of Abstract Base Class -- Destructor

- *BookingClass* has a *protected* destructor.
- *Signature:* `BookingClass::~BookingClass()`
- Before construction of any instance of a derived class, there must be an instantiation of the base class. Every derived class has a base class part in its *object layout* and therefore call to the destructor of a derived class is always followed by the call to the destructor of the base class. For this to happen, the destructor of the base class must actually be accessible from the derived class. Therefore, destructor in a base class should not be *private*.
  If the destructor of any base class, which has singleton derived classes (or as in this case a template of singleton classes) is made *public*, it will become transparent to the client-side. If in the application, the base class destructor is called on the singleton instance of the derived class, the base class part of the object will get destroyed hence failing the singleton for all applications in the project.
  Therefore *protected* is chosen as the apt access-specifier.
- The destructor is *virtual (polymorphic).* It is important that in the base class of a *polymorphic hierarchy*, the destructors are also declared as polymorphic. Polymorphic destructors in the base classes enable *dynamic dispatch* of destructors and prevent *object slicing.*
- Destructors do not have a return type and neither can they be *const* methods. They also do not have any arguments.

Design Of Abstract Base Class -- Operator Overloading

- The *copy-assignment operator* ‘=’ in the *derived template class BookingClassTypes* is blocked (declared in *private* access-specifier).
- This is done because of the *const*-ness of the *singleton instance* of any *sub-type* of *BookingClass*.

- *BookingClass* has an overloaded *output streaming operator* that prints all the details of the *singleton* object of any instance of the derived template class on the console.
- *Signature:*
  `std::ostream &operator<<(std::ostream &, const BookingClass &)`
- *Friendship:* This method is not a member function but a *friend* function that is declared in the global scope. *Friend functions* can access private (and protected) members of the class in which they are declared as a *friend.*

Friendship is realized by the keyword *friend* which is precedes the signature of the function when it is declared inside a class.

- *Arguments:* This operator function has two parameters. The first one is of output stream object *std::ostream* type. The standard objects like *std::cout* or *std::cerr* are of this type. The LHS entity in the expression “*cout << x”* (that is, *cout*) is passed as the first parameter. This parameter is passed as a *non-const reference*; *non-const* because the state of the output-stream should be changed by the function when it inserts *formatted output* to that stream. The parameter is *passed by reference* because it is imperitive to output the content/message on the same output-stream on which it is intended to by the expression "*cout << x*" in the caller function.

  The second arguement is of user-defined type *BookingClass* which is actually the RHS entity in the same expression. It is *passed by reference* because it is an *abstract class* and pass-by-value will mean constructing a local instance of *BookingClass* which is not possible. The parameter is passed as a *const* reference because the only way the singleton object of an instance of *BookingClassTypes* can be obtained is by the call to the template method *BookingClassTypes<T>::Type*, that returns a *const reference* to the singleton object. *const* reference cannot be upcasted to a *non-const* reference of its base class.

  (*NOTE:* Design of *BookingClassTypes* is discussed later)
- *Return Type:* The return type is *std::ostream*. The same stream that is passed as an arguement is returned to enable *chained output streaming* (like "*cout << x << y << z*"). To ensure that the same stream is returned, the *return is by reference*. It is a ret*urn by non-const-reference* because the state of the returned output stream object might have to change in the caller function in case there is another instruction for formatted output chained with the former one. Return-by-reference is possible here because the returned output stream object is not a local object but rather the same object that was passed-by-reference as the first parameter.

Design Of Abstract Base Class -- Non-Static Member Functions

- *BookingClass* has numerous *public non-static member functions*, one meant for each of the static data attributes of the instances of the derived template class. Each of these member functions is *virtual/polymorphic* to enable *dynamic dispatch*. In fact, these are *pure virtual* functions in the base class.

```
double BookingClass::GetLoadFactor() const
```

```
std::__cxx11::string BookingClass::GetName() const
```

```
bool BookingClass::IsSitting() const
```

```
bool BookingClass::IsAC() const
```

```
unsigned BookingClass::GetNumberOfTiers() const
bool BookingClass::IsLuxury() const
double BookingClass::GetReservationCharge() const
double BookingClass::GetTatkalCharge() const
double BookingClass::GetMaxTatkalCharge() const
double BookingClass::GetMinTatkalCharge() const
unsigned BookingClass::GetMinDistanceForTatkalCharge() const
```

The return types of the functions are compatible with the data type of the respective *static data member* of the derived class, the value of which they are returning.
Each one of them is a *constant method*. This is ensured by the *const* keyword on the extreme right of the signature. It is important because of the *constness* of the *singleton* object of any instance of the derived template class. Only a *const* method can be called on a *const* object.

Design Of Derived Template Class -- Singleton Classes

- *BookingClassTypes* is implemented as a template of *singleton* classes. That is, at most one object of any class instance of *BookingClassTypes* can be constructed. This is realised as a *Meyer's implementation of a singleton*.
- Therefore, the constructor is kept *private* and a public member function *BookingClassTypes<T>::Type* is used in the interface to *get* the *singleton instance of any class of BookingClassTypes template*.

Design Of Derived Template Class -- Static Data Members

- All the attributes of a booking class are stored in *static data members*. The template class *BookingClassTypes* has *11 static data members*, all of which are *const.*

BookingClassTypes<T>::sName	*string*
BookingClassTypes<T>::sLoadFactor	*double*
BookingClassTypes<T>::sIsSitting	*bool*
BookingClassTypes<T>::sIsAC	*bool*
BookingClassTypes<T>::sIsLuxury	*bool*

BookingClassTypes<T>::sNumberOfTiers	*unsigned int*
BookingClassTypes<T>::sReservationCharge	*double*
BookingClassTypes<T>::sTatkalCharge	*double*
BookingClassTypes<T>::sMaxTatkalCharge	*double*
BookingClassTypes<T>::sMinTatkalCharge	*double*
BookingClassTypes<T>::sMinDistanceForTatkalCharge	*unsigned int*

- For each of the *8 booking classes* (all of which are instances of the template class *BookingClassTypes* and can be accessed from the namespace of *BookingClass* class like "*BookingClass::ACFirstClass*"), all the *11 static data members* are initialized using the *master data* given in the specifications.

Design Of Derived Template Class -- Constructor

- The template class has a *private constructor*.
- *Signature:* `template<class T> BookingClassTypes<T>::BookingClassTypes()`
- A constructor does not have a return type and neither can it be a *const* method.
- The constructor has no arguments because there are no *non-static data members* associated with any instance of the template class to intialize. All the properties of a booking class are stored in *static data members*.
- It is important to keep the constructor *private* otherwise an instance of any *template<class T> BookingClassTypes<T>* class can be constructed anywhere, any number of times thus violating its *singleton* property.

Design Of Derived Template Class -- Destructor

- *BookingClassTypes* has a *private* destructor
- *Signature:* `template<class T> BookingClassTypes<T>::~BookingClassTypes()`
- Destructors do not have any arguments or return type and neither can they be *const* methods.
- Any instance of *BookingClassTypes* template is a *singleton* class and the only instance of this class is of *static* type. When a variable is declared as static, space for it gets allocated for the lifetime of the program. Therefore, the singleton instance is not *destructed* until the program gets terminated.
- It is a good idea to make a destructor of a *singleton* class private because then the client/application code won't call the destructor by accident. Calling the destructor would cause the singleton to fail for all applications in the project as the instance would become invalid.

Design Of Derived Template Class -- Static Member Function

- The template has a *public static member function* that acts as an interface between the private constructor and the outside/global scope.
- *Signature:*
```
template<class T> static const BookingClassTypes<T>
&BookingClassTypes<T>::Type()
```
- A *static object* of a *class-instance* of *BookingClassTypes* is constructed in this method. This is a singleton because static storage duration for a function local means that only one instance of that local exists in the program. This singleton instance is returned everytime a “*BookingClassTypes<T>::Type()*” call is made (where *T* is some template arguement). The *return is by reference* because the template does not have a *copy constructor*; moreover it is a template of singleton classes. It is returned as a *const* reference because the singleton object that is returned to the caller function must be treated as a constant object.
  Note that return by reference is possible here because the returned object is not a *local non-static object* but rather a *local static object* which is not allocated on the stack frame.

Design Of Derived Template Class -- Non-Static Member Function

- All the *11 pure virtual* member functions in the base class *BookingClass* are *overridden* in all the *static sub-types* of *BookingClass*.
- Each one of them is implemented as an *inline* function.
- Values of appropriate *static data members* is returned by each of these member functions.

Divyaang

Inclusion-Parametric Polymorphic Design

- *Divyaang* hierarchy is rooted at an *abstract base class* -- *Divyaang*
- A *template class DivyaangTypes* is derived from the abstract base class
- The template of the derived classes is designed in such a way that every instance of the template is a *singleton class*
- As is clear from the *abstractness* of the base class, this hierarchy is a polymorphic hierarchy that allows *dynamic dispatch* of calls to *polymorphic methods*.

Design Of Abstract Base Class -- Tag Types

- The booking application allows *4 divyaang sub-categories* (*Blind, OrthopaedicallyHandicapped, CancerPatients, TBPatients*) and corresponding to each of the divyaang categories a *user-defined data type* is defined in the *private* access-specifier as a *placeholder* to tag each of the specialized divyaange/disability type.

  struct Divyaang::BlindType
  struct Divyaang::OrthoHandicapType
  struct Divyaang::CancerPatientsType
  struct Divyaang::TBPatientsType
- In the *public* access-specifier, the *target sub-types* of the *Divyaang* (each one of which is instantiated from the derived template class using the appropriate *tag type* from above as the template argument) are *typedef*-ed so that they can be accessed by using *Divyaang* as the qualifier. For example --

  typedef DivyaangTypes<BlindType> Blind

Design Of Abstract Base Class -- Constructor

- *Divyaang* has a *protected* constructor.
- *Signature:* `Divyaang::Divyaang()`
- *Divyaang* is an abstract class and hence cannot be instantiated to construct a *stand-alone* object (though it will be instantiated when a derived class is instantiated). Therefore there is no need of the constructor outside of the hierarchy and hence it can be avoided to be kept as *public*.

  It cannot be kept *private* because it needs to be accessible to the derived class(es) when they are instantiated. Therefore *protected* is chosen as the apt access-specifier.
- Constructors do not have a return type and neither can they be *const* methods.

Design Of Abstract Base Class -- Destructor

- *Divyaang* has a *protected* destructor.

- *Signature:* `Divyaang::~Divyaang()`
- Before construction of any instance of a derived class, there must be an instantiation of the base class. Every derived class has a base class part in its *object layout* and therefore call to the destructor of a derived class is always followed by the call to the destructor of the base class. For this to happen, the destructor of the base class must actually be accessible from the derived class. Therefore, destructor in a base class should not be *private*.
  If the destructor of any base class, which has singleton derived classes (or as in this case a template of singleton classes) is made *public*, it will become transparent to the client-side. If in the application, the base class destructor is called on the singleton instance of the derived class, the base class part of the object will get destroyed hence failing the singleton for all applications in the project.
  Therefore *protected* is chosen as the apt access-specifier.
- The destructor is *virtual (polymorphic).* It is important that in the base class of a *polymorphic hierarchy*, the destructors are also declared as polymorphic. Polymorphic destructors in the base classes enable *dynamic dispatch* of destructors and prevent *object slicing.*
- Destructors do not have a return type and neither can they be *const* methods. They also do not have any arguments.

Design Of Abstract Base Class -- Operator Overloading

- The *copy-assignment operator '='* in the *derived template class DivyaangTypes* is blocked (declared in *private* access-specifier).
- This is done because of the *const*-ness of the *singleton instance* of any *sub-type* of *Divyaang*.

- *Divyaang* has an overloaded *output streaming operator* that prints all the details of the *singleton* object of any instance of the derived template class on the console (particularly it prints the name of the divyaang category and its respective column in the *disability concession factor matrix*).
- *Signature:*
  `std::ostream &operator<<(std::ostream &, const Divyaang &)`
- *Friendship:* This method is not a member function but a *friend* function that is declared in the global scope. *Friend functions* can access private (and protected) members of the class in which they are declared as a *friend.*
  Friendship is realized by the keyword *friend* which is precedes the signature of the function when it is declared inside a class.
- *Arguments:* This operator function has two parameters. The first one is of output stream object *std::ostream* type. The standard objects like *std::cout* or *std::cerr*

are of this type. The LHS entity in the expression "*cout << x*" (that is, *cout*) is passed as the first parameter. This parameter is passed as a *non-const reference*; *non-const* because the state of the output-stream should be changed by the function when it inserts *formatted output* to that stream. The parameter is *passed by reference* because it is imperitive to output the content/message on the same output-stream on which it is intended to by the expression "*cout << x*" in the caller function.

The second arguement is of user-defined type *Divyaang* which is actually the RHS entity in the same expression. It is *passed by reference* because it is an *abstract class* and pass-by-value will mean constructing a local instance of *Divyaang* which is not possible. The parameter is passed as a *const* reference because the only way the singleton object of an instance of *DivyaangTypes* can be obtained is by the call to the template method *DivyaangTypes<T>::Type*, that returns a *const reference* to the singleton object. *const* reference cannot be upcasted to a *non-const* reference of its base class.

(*NOTE:* Design of *DivyaangTypes* is discussed later)

- *Return Type:* The return type is *std::ostream*. The same stream that is passed as an arguement is returned to enable *chained output streaming* (like "*cout << x << y << z*"). To ensure that the same stream is returned, the *return is by reference*. It is a re*turn by non-const-reference* because the state of the returned output stream object might have to change in the caller function in case there is another instruction for formatted output chained with the former one. Return-by-reference is possible here because the returned output stream object is not a local object but rather the same object that was passed-by-reference as the first parameter.

Design Of Abstract Base Class -- Non-Static Member Functions

- *Divyaang* has a *public non-static member function* that returns the name of the divyaang sub-category.
- *Signature:* `std::__cxx11::string Divyaang::GetName() const`
- *Return type:* This method returns the value of the *static data member DivyaangTypes<T>::sName* of any class of the derived template *DivyaangTypes* (discussed later) on the instance of which the method is called. This *static data member* is implemented as a *string* and hence the return type.

- *Divyaang* has a *public non-static member function* that returns the concession factor for a *static Divyaang sub-type* given the booking class,
- *Signature:*

```
double Divyaang::GetConcessionFactor(const BookingClass &)
const
```

- *Return type:* The value of a concession factor lies in the interval *[0,1]* and hence its value must be a *double*.
- *Arguments:* It has only one argument, *BookingClass,* and that is passed as a *const* reference. It is so because any instance of a *static sub-type* of *BookingClass* is a singleton instance that is available through a *public static member function* that returns the singleton instance as a *const reference*. Here actually an *implicit upcast* is happening; a *const reference* of a derived class cannot be upcasted to a non-*const* reference of its base class.

- The common features of the two *non-static member functions* include *constness* and *polymorphism*.
- *Constant method:* The *const*-ness of the method is ensured by *const* keyword on the extreme right of the signatures. It is important because of the *constness* of the *singleton* object of any instance of the derived template class. Only a *const* method can be called on a *const* object.
- Both the methods are *virtual/polymorphic* to enable *dynamic dispatch* to appropriate *static sub-types*. In fact these are *pure virtual* functions in the abstract base class.

Design Of Derived Template Class -- Singleton Classes

- *DivyaangTypes* is implemented as a template of *singleton* classes. That is, at most one object of any class instance of *DivyaangTypes* can be constructed. This is realised as a *Meyer's implementation of a singleton*.
- Therefore, the constructor is kept *private* and a public member function *DivyaangTypes<T>::Type* is used in the interface to *get* the *singleton instance of any class of DivyaangTypes template*.

Design Of Derived Template Class -- Static Data Members

- In derived template *DivyaangTypes*, a *private static const data member* is defined that stores the name of the disability/divyaang category. It is of *string* data type.

```
static template<class T> const std::__cxx11::string
DivyaangTypes<T>::sName
```

- The template has another *private static const data member* that is of utmost importance in computing fare for any instance of the *Booking sub-type* associated with the *Divyaang* sub-type of *BookingCategory* (these classes are discussed later in great detail). It stores the *concession factors* for a *Divyaang static sub-type* against the names of the booking classes.

```
static template<class T> const std::map<std::__cxx11::string,
double> DivyaangTypes<T>::sConcessionFactors
```

- It is implemented as a *map* (previously discussed in *Railways* design) in which the first template arguement is chosen as *string* for the type of *key* and the second template argument is chosen as *double* as the type of *value*.
- The *key* is the name of the *static sub-type* of *BookingClass* and the *value* is the corresponding concession factor as given in the *master data* in the *specifications*.
- The map consists of *8 key-value pairs*.

Design Of Derived Template Class -- Constructor

- The template class has a *private constructor*.
- *Signature:* `template<class T> DivyaangTypes<T>::DivyaangTypes()`
- A constructor does not have a return type and neither can it be a *const* method.
- The constructor has no arguments because there are no *non-static data members* associated with any instance of the template class to intialize. All the properties of a divyaang sub-category are stored in *static data members*.
- It is important to keep the constructor *private* otherwise an instance of any *template<class T> DivyaangTypes<T>* class can be constructed anywhere, any number of times thus violating its *singleton* property.

Design Of Derived Template Class -- Destructor

- *DivyaangTypes* has a *private* destructor
- *Signature:*
  `template<class T> DivyaangTypes<T>::~DivyaangTypes()`
- Destructors do not have any arguments or return type and neither can they be *const* methods.
- Any instance of *DivyaangTypes* template is a *singleton* class and the only instance of this class is of *static* type. When a variable is declared as static, space for it gets allocated for the lifetime of the program. Therefore, the singleton instance is not *destructed* until the program gets terminated.
- It is a good idea to make a destructor of a *singleton* class private because then the client/application code won't call the destructor by accident. Calling the destructor would cause the singleton to fail for all applications in the project as the instance would become invalid.

Design Of Derived Template Class -- Static Member Function

- The template has a *public static member function* that acts as an interface between the private constructor and the outside/global scope.
- *Signature:*
  ```
  template<class T> static const DivyaangTypes<T>
  &DivyaangTypes<T>::Type()
  ```

- A *static object* of a *class-instance* of *DivyaangTypes* is constructed in this method. This is a singleton because static storage duration for a function local means that only one instance of that local exists in the program. This singleton instance is returned everytime a "*DivyaangTypes<T>::Type()*" call is made (where *T* is some template arguement). The *return is by reference* because the template does not have a *copy constructor*; moreover it is a template of singleton classes. It is returned as a *const* reference because the singleton object that is returned to the caller function must be treated as a constant object.
  Note that return by reference is possible here because the returned object is not a *local non-static object* but rather a *local static object* which is not allocated on the stack frame.

Design Of Derived Template Class -- Non-Static Member Function

- The *pure virtual* method *Divyaang::GetName* in *Divyaang* class is overridden in the derived template class.
- This method returns the value of the *static data member DivyaangTypes<T>::sName* of any class of the derived template *DivyaangTypes* on the instance of which the method is called.
- This method is implemented as an *inline* function.

- The *pure virtual* method *Divyaang::GetConcessionFactor* in *Divyaang* class is overridden in the derived template class.
- *Exceptions:* The address of *BookingClass* passed as reference is first matched with that of all the *8 booking classes* to ensure that it is a valid booking class. If it matches none of them, a *Bad_Access exception* is thrown. Otherwise the *const static data member* in the *concrete derived class of Divyaang* that stores the concession factors in a *map* (*DivyaangTypes<T>::sConcessionFactors*) is searched with the key as the name of the *BookingClass* and the corresponding value is returned.

BookingCategory

Inclusion-Parametric Polymorphic Design

- *BookingCategory* hierarchy is rooted at *abstract base class* -- *BookingCategory*
- A *template class* *BookingCategoryTypes* is derived from the abstract base class
- The template of the derived classes is designed in such a way that every instance of the template is a *singleton class*
- As is clear from the *abstractness* of the base class, this hierarchy is a polymorphic hierarchy that allows *dynamic dispatch* of calls to *polymorphic methods*.

Design Of Abstract Base Class -- Tag Types

- The booking application allows *6 kinds of booking categories* (*General*, *Ladies*, *SeniorCitizen*, *Divyaang*, *Tatkal*, *PremiumTatkal*) and corresponding to each of the *booking categories* a *user-defined data type* is defined in the *private* access-specifier as a *placeholder* to tag each of the *booking category*.

  struct BookingCategory::GeneralType
  struct BookingCategory::LadiesType
  struct BookingCategory::SeniorCitizenType
  struct BookingCategory::DivyaangType
  struct BookingCategory::TatkalType
  struct BookingCategory::PremiumTatkalType
- In the *public* access-specifier, the *target sub-types* of the *BookingCategory* (each one of which is instantiated from the derived template class using the appropriate *tag type* from above as the template argument) are *typedef*-ed so that they can be accessed by using *BookingCategory* as the qualifier. For example --

  typedef BookingCategoryTypes<GeneralType> General

Design Of Abstract Base Class -- Constructor

- *BookingCategory* has a *protected* constructor.
- *Signature:* `BookingCategory::BookingCategory()`
- *BookingCategory* is an abstract class and hence cannot be instantiated to construct a *stand-alone* object (though it will be instantiated when a derived class is instantiated). Therefore there is no need of the constructor outside of the hierarchy and hence it can be avoided to be kept as *public*.

  It cannot be kept *private* because it needs to be accessible to the derived class(es) when they are instantiated. Therefore *protected* is chosen as the apt access-specifier.
- Constructors do not have a return type and neither can they be *const* methods.

Design Of Abstract Base Class -- Destructor

- *BookingCategory* has a *protected* destructor.
- *Signature:* `BookingCategory::~BookingCategory()`
- Before construction of any instance of a derived class, there must be an instantiation of the base class. Every derived class has a base class part in its *object layout* and therefore call to the destructor of a derived class is always followed by the call to the destructor of the base class. For this to happen, the destructor of the base class must actually be accessible from the derived class. Therefore, destructor in a base class should not be *private*.
  If the destructor of any base class, which has singleton derived classes (or as in this case a template of singleton classes) is made *public*, it will become transparent to the client-side. If in the application, the base class destructor is called on the singleton instance of the derived class, the base class part of the object will get destroyed hence failing the singleton for all applications in the project.
  Therefore *protected* is chosen as the apt access-specifier.
- The destructor is *virtual (polymorphic).* It is important that in the base class of a *polymorphic hierarchy*, the destructors are also declared as polymorphic. Polymorphic destructors in the base classes enable *dynamic dispatch* of destructors and prevent *object slicing.*
- Destructors do not have a return type and neither can they be *const* methods. They also do not have any arguments.

Design Of Abstract Base Class -- Operator Overloading

- The *copy-assignment operator* '=' in the *derived template class BookingCategoryTypes* is blocked (declared in *private* access-specifier).
- This is done because of the *const*-ness of the *singleton instance* of any *sub-type* of *BookingCategory*.

- *BookingCategory* has an overloaded *output streaming operator* that prints the name of the *singleton* object of any instance of the derived template class on the console (which is actually stored in a *static data member*).
- *Signature:*
  `std::ostream &operator<<(std::ostream &, const BookingCategory &)`
- *Friendship:* This method is not a member function but a *friend* function that is declared in the global scope. *Friend functions* can access private (and protected) members of the class in which they are declared as a *friend.*
  Friendship is realized by the keyword *friend* which is precedes the signature of the function when it is declared inside a class.
- *Arguments:* This operator function has two parameters. The first one is of output stream object *std::ostream* type. The standard objects like *std::cout* or *std::cerr*

are of this type. The LHS entity in the expression “*cout << x”* (that is, *cout*) is passed as the first parameter. This parameter is passed as a *non-const reference*; *non-const* because the state of the output-stream should be changed by the function when it inserts *formatted output* to that stream. The parameter is *passed by reference* because it is imperitive to output the content/message on the same output-stream on which it is intended to by the expression "*cout << x*" in the caller function.

BookingClass parameter is passed as a *const reference* because the singleton instance of any of its sub-types is available by a *static member function* that returns the singleton as a *const reference*. *const* instance of a derived class cannot be upcasted to a non *const* reference of its base class. Besides, *BookingClass* is an abstract class and passing the parameter by value will be equivalent to *instantiating* a local instance of *BookingClass* which is not possible. (*NOTE:* Design of *BookingCategoryTypes* is discussed later)

- *Return Type:* The return type is *std::ostream*. The same stream that is passed as an arguement is returned to enable *chained output streaming* (like "*cout << x << y << z*"). To ensure that the same stream is returned, the *return is by reference*. It is a ret*urn by non-const-reference* because the state of the returned output stream object might have to change in the caller function in case there is another instruction for formatted output chained with the former one. Return-by-reference is possible here because the returned output stream object is not a local object but rather the same object that was passed-by-reference as the first parameter.

Design Of Abstract Base Class -- Static Data Members

- *BookingCategory* has *4 protected static const data members* that store various values concerning the eligibility of a person/*Passenger* for a *booking category*.
- *BookingCategory::sMaxAgeMalesForLadies* is of *unsigned int built-in type* that stores the maximum age a *Male* can have to be eligible to book under *Ladies* booking category.
- *BookingCategory::sMinAgeMalesForSenCit* is of *unsigned int built-in type* that stores the minimum age a *Male* can have to be eligible to book under *SeniorCitizen* booking category.
- *BookingCategory::sMinAgeFemalesForSenCit* is of *unsigned int built-in type* that stores the minimum age a *Female* can have to be eligible to book under *SeniorCitizen* booking category.
- *BookingCategory::sHoursBeforeTravelForPriority* is of *unsigned int built-in type* that stores the hours before the travel a person/*Passenger* should book to be eligible for a *priority (Tatkal* or *PremimumTatkal)* booking category.
- There values are taken from the *master data* in the *specifications* and are accordingly initialized to *12*, *60*, *58* and *24* respectively.

Design Of Abstract Base Class -- Non-Static Member Functions

- *BookingCategory* has a *public non-static member function* that returns the name of the *BookingCategory* sub-type.
- *Signature:* `std::__cxx11::string BookingCategory::GetName() const`
- *Return type:* This method returns the value of the *static data member* *BookingCategoryTypes<T>::sName* of any class of the derived template *BookingCategoryTypes* (discussed later) on the instance of which the method is called. This *static data member* is implemented as a *string* and hence the return type.

- *BookingCategory* has a *public non-static member function* that returns *true* if the *Passenger* passed as argument satisfies the *elligibility criteria* for the particular *BookingCategory sub-type* and *false* otherwise.
- *Signature:* `bool BookingCategory::IsElligible(const Passenger &, const Date &) const`
- *Return type: bool* is the apt return type because it can only return from two possible values.
- *Arguments:* From the *master data* in the *specifications*, the following observations were made.
    - elligibility for *General* is always *true* (so the arguments do not matter)
    - elligibility for *Ladies* depends only on the *Passenger* (so the first argument matters)
    - elligibility for *SeniorCitizen* depends only on the *Passenger* (so the first argument matters)
    - elligibility for *Divyaang* depends only on the *Passenger* (so the first argument matters)
    - elligibility for *Tatkal/PremiumTatkal* depends only on *date of reservation* relative to the *date of booking* (so the first argument does not matter)

  Consequently, to design a *general member function* that can judge the eligibility for any sub-type of *BookingCategory*, it should have *two arguments* -- first one is the *Passenger* whose eligibility has to be checked and the second one is *Date* of booking on which the *Passenger* wants to travel. The second argument is imperative only while checking the eligibility for *priority booking categories*.

  Note that there is no need to pass another *Date* as argument for the *date of reservation* because it is always *set* by the application at the time of booking as the *current date* on the machine/system. So the method *rightly* treats the *current date* as the *date of reservation* and then tests the eligibility for a *priority booking category*.

- Both the arguments are of *user-defined type* and hence are *passed by reference* to avoid copying overheads. Besides they are passed as *const* reference so that the state of the *actual arguments* stays intact.

- *BookingCategory* has a *public non-static member function* that returns the address of an instance of the *corresponding Booking sub-type* (*dynamically allocated memory*) with the attributes same as the ones passed as arguements.
- *Signature:* `const Booking *BookingCategory::SelectBooking(const Station &, const Station &, const Date &, const BookingClass &, const Passenger &, const Date &) const`
- *Purpose:* This member is of utmost importance in implementing the *virtual construction idiom*. Based on the *static sub-type* of *BookingCategory*, it *selects* the suitable specialization of *Booking* class the instance of which needs to be constructed. *More details on Virtual Construction Idiom are discussed in the design of Booking class and hierarchy.*
- *Arguments:* The role/purpose of each of the arguments will be clear once the design of *Booking* hierarchy is introduced. For now, it has *6 arguements*. The first two arguments of *Station* type are the *departure* and *destination* stations respectively. The third argument of *Date* type is the *date of booking/travel*. The fourth argument is of *BookingClass* type that can be any one of the *8 concrete static sub-types* of *BookingClass*. The fifth argument is of *Passenger* type and the sixth argument is of *Date* type that stands for the *date of reservation*.
  BookingClass parameter is passed as a *const reference* because the singleton instance of any of its sub-types is available by a *static member function* that returns the singleton as a *const reference*. *const* instance of a derived class cannot be upcasted to a non *const* reference of its base class. Besides, *BookingClass* is an abstract class and passing the parameter by value will be equivalent to *instantiating* a local instance of *BookingClass* which is not possible.
  The other parameters are *passed by reference* to avoid copying overheads. They are passed as *const* references to ensure that the state of the *actual arguments* stays intact.
- *Return type:* This method returns the address of an instance of the suitable *sub-type* of *Booking* class (with *data members* initialized with these arguements) and hence the return type is *const Booking**. Returning as a pointer to *const Booking* ensures that it is not *tampered* with in the caller function. Here an *implicit dynamic upcasting* is happening that is not *lethal* because *Booking* has a *polymorphic hierarchy*.

- The common features of all the *non-static member functions* include *constness* and *polymorphism*.

- *Constant method:* The *const*-ness of the method is ensured by *const* keyword on the extreme right of the signatures. It is important because of the *constness* of the *singleton* object of any instance of the derived template class. Only a *const* method can be called on a *const* object.
- All the methods are *virtual/polymorphic* to enable *dynamic dispatch* to appropriate *static sub-types*. In fact these are *pure virtual* functions in the abstract base class.

Design Of Derived Template Class -- Singleton Classes

- *BookingCategoryTypes* is implemented as a template of *singleton* classes. That is, at most one object of any class instance of *BookingCategoryTypes* can be constructed.
  This is realised as a *Meyer's implementation of a singleton*.
- Therefore, the constructor is kept *private* and a public member function *BookingCategoryTypes<T>::Type* is used in the interface to *get* the *singleton instance of any class of BookingCategoryTypes template*.

Design Of Derived Template Class -- Static Data Members

- In derived template *BookingCategoryTypes*, a *private static const data member* is defined that stores the name of the booking category in *string* data type.

```
static template<class T> const std::__cxx11::string
BookingCategoryTypes<T>::sName
```

Design Of Derived Template Class -- Constructor

- The template class has a *private constructor*.
- *Signature:*

```
template<class T> BookingCategoryTypes<T>::BookingCategoryTypes()
```

- A constructor does not have a return type and neither can it be a *const* method.
- The constructor has no arguments because there are no *non-static data members* associated with any instance of the template class to intialize.
- It is important to keep the constructor *private* otherwise an instance of any *template<class T> BookingCategoryTypes<T>* class can be constructed anywhere, any number of times thus violating its *singleton* property.

Design Of Derived Template Class -- Destructor

- *BookingCategoryTypes* has a *private* destructor
- *Signature:*

```
template<class T> BookingCategoryTypes<T>::~BookingCategoryTypes()
```

- Destructors do not have any arguments or return type and neither can they be *const* methods.

- Any instance of *BookingCategoryTypes* template is a *singleton* class and the only instance of this class is of *static* type. When a variable is declared as static, space for it gets allocated for the lifetime of the program. Therefore, the singleton instance is not *destructed* until the program gets terminated.
- It is a good idea to make a destructor of a *singleton* class private because then the client/application code won't call the destructor by accident. Calling the destructor would cause the singleton to fail for all applications in the project as the instance would become invalid.

Design Of Derived Template Class -- Static Member Function

- The template has a *public static member function* that acts as an interface between the private constructor and the outside/global scope.
- *Signature:*
```
template<class T> static const BookingCategoryTypes<T>
&BookingCategoryTypes<T>::Type()
```
- A *static object* of a *class-instance* of *BookingCategoryTypes* is constructed in this method. This is a singleton because static storage duration for a function local means that only one instance of that local exists in the program. This singleton instance is returned everytime a "*BookingCategoryTypes<T>::Type()*" call is made (where *T* is some template arguement). The *return is by reference* because the template does not have a *copy constructor*; moreover it is a template of singleton classes. It is returned as a *const* reference because the singleton object that is returned to the caller function must be treated as a constant object.
  Note that return by reference is possible here because the returned object is not a *local non-static object* but rather a *local static object* which is not allocated on the stack frame.

Design Of Derived Template Class -- Non-Static Member Functions

- The *pure virtual* method *BookingCategory::GetName* in *BookingCategory* class is overridden in the derived template class.
- This method returns the value of the *static data member* *BookingCategoryTypes<T>::sName* of any class of the derived template *BookingCategoryTypes* on the instance of which the method is called.
- This method is implemented as an *inline* function.

- The *pure virtual* method *BookingCategory::IsElligible* in *BookingCategory* class is overridden in the derived template class.
- The method implements for each of the *6* static sub-types of *BookingCategory* the *eligibility criteria* as given in the *master data* of the *specifications.*

- This method may use the *member functions Passenger::GetGender*, *Passenger::GetDisabilityType*, *Passenger::GetAge* from the interface of *Passenger* class in implementation.

- The *pure virtual* method *BookingCategory::SelectBooking* in *BookingCategory* class is overridden in the derived template class.
- For each of the *6 booking categories*, the overridden method is implemented differently.
  BookingCategory::General::SelectBooking calls the *static member function Booking::GeneralBooking::CreateSpecialBooking* to construct an instance of *Booking::GeneralBooking sub-type* and return it.
  BookingCategory::Ladies::SelectBooking calls the *static member function Booking::LadiesBooking::CreateSpecialBooking* to construct an instance of *Booking::LadiesBooking sub-type* and return it.
  BookingCategory::SeniorCitizen::SelectBooking calls the *static member function Booking::SeniorCitizenBooking::CreateSpecialBooking* to construct an instance of *Booking::SeniorCitizenBooking sub-type* and return it.
  BookingCategory::Divyaang::SelectBooking calls the *static member function Booking::DivyaangBooking::CreateSpecialBooking* to construct an instance of *Booking::DivyaangBooking sub-type* and return it.
  BookingCategory::Tatkal::SelectBooking calls the *static member function Booking::TatkalBooking::CreateSpecialBooking* to construct an instance of *Booking::TatkalBooking sub-type* and return it.
  BookingCategory::PremiumTatkal::SelectBooking calls the *static member function Booking::PremiumTatkalBooking::CreateSpecialBooking* to construct an instance of *Booking::PremiumTatkalBooking sub-type* and return it.

- This is the core principle of *virtual construction idiom* which will be further elaborated in the design of *Booking* class and hierarchy.

Concessions

Ad-Hoc Polymorphic Design

- *Concessions* hierarchy is rooted at *base class* -- *Concessions*
- It is a *flat single-level static-polymorphic hierarchy*. It has *4 specialized classes* -- *GeneralConcession*, *LadiesConcession*, *SeniorCitizenConcession* and *DivyaangConcession*.

Overview

- The hierarchy is meant to *store the information related to concessions* in various booking categories in a structured design.
- The classes in this hierarchy need not be instantiated at all because their objects do not serve any useful purpose in the rest of the design. Hence the *constructors* and *destructors* of the classes can be encapsulated.
- All the information concerning the concessions is kept in *static data members* and *static member functions* are used in the interface to retrieve this information.

Design of Base Class

- Base class *Concessions* has a very trivial design, without any *static/non-static data members* and *static/non-static member functions*.
- It has a *protected constructor* -- `Concessions::Concessions()`
  As discussed in *Overview* neither the base class nor any of the derived classes in the hierarchy are needed to be instantiated anywhere in the rest of the design because all the information is stored in *static data members* and is made available through *static member functions*. In this scenario, the constructors (and hence the *destructors*) of all the classes in the heirarchy can be encapsulated in the *private access specifier* to minimize the functionality of instantiating these classes. But instead of making *private*, the constructor of the base class is advised to be kept *protected* because a call to the constructor of any *derived class* is followed by the call to the base class's constructor.
  Here it might not matter if *private* or *protected* access specifier is used for the base class constructor because the constructors of the derived classes are all *private* and hence will never be called from outside the scope of the class; otherwise, it could have mattered.
- It has a *protected destructor*-- `Concessions::~Concessions()` as discussed.

Design of Derived Classes

GeneralConcession

- It has a *private static const data member* of *double data type* that stores the concession factor for *General booking category* -- *sConcessionFactor*.

- It has a *public static member function* that returns the value of this *static data member*. `static double GeneralConcession::GetConcessionFactor()`
  The member function does not need any arguments because as given in the *master data* of the *specifications*, the concession factor for *General booking category* is invariantly *0.00*.
- Besides as already discussed, it has *private constructor* and a *private destructor*.

LadiesConcession

- It has a *2 private static const data members* -- *sConcessionFactorMales* and *sConcessionFactorFemales*. Both of them are of *double data type*.
- It has a *public static member function* that returns the value of the concession that is applicable to the *Passenger* passed as the argument.
  `static double LadiesConcession::GetConcessionFactor(const Passenger &)`
  It has one argument of *user-defined Passenger* type. It is *passed-by-reference* to avoid copy overheads and is passed as a *const* reference to ensure that the state of the *actual argument* stays intact.
  The logic of this function *might* first check the *eligibility* of the *Passenger* passed as argument for the *Ladies booking category*, using the method *BookingCategory::Ladies::IsElligible*. If the *Passenger* is not elligible then a *Bad_Elligibility exception* may be thrown.
- Besides as already discussed, it has *private constructor* and a *private destructor*.

SeniorCitizenConcession

- It has a *2 private static const data members* -- *sConcessionFactorMales* and *sConcessionFactorFemales*. Both of them are of *double data type*.
- It has a *public static member function* that returns the value of the concession that is applicable to the *Passenger* passed as the argument.
  `static double SeniorCitizenConcession::GetConcessionFactor(const Passenger &)`
  It has one argument of *user-defined Passenger* type. It is *passed-by-reference* to avoid copy overheads and is passed as a *const* reference to ensure that the state of the *actual argument* stays intact.
  The logic of this function *might* first check the *eligibility* of the *Passenger* passed as argument for the *SeniorCitizen booking category*, using the method *BookingCategory::SeniorCitizen::IsElligible*. If the *Passenger* is not elligible then a *Bad_Elligibility exception* may be thrown.
- Besides as already discussed, it has *private constructor* and a *private destructor*.

DivyaangConcession

- Though *Concessions* hierarchy is meant for storing the information concerning the *concessional booking categories*, *Divyaang booking category* further has *4 sub-categories* that are defined in the *inclusion parametric-polymorphic hierarchy* rooted at *Divyaang* (already discussed in detail).
  As already discussed in the design of *Divyaang* hierarchy, *concessions* associated with different *booking classes* for each *static sub-type* of *Divyaang* is stored in the *static data members* of the instances of the derived template class.
  Hence there is no need to declare another set of same static information. Therefore, *DivyaangConcession* lacks any need for *static data members*.
- It has a *public static member function* that returns the value of concession applicable to a *Passenger* travelling in a particular *BookingClass*.
  ```
  static double DivyaangConcession::GetConcessionFactor(const
  Passenger &p, const BookingClass &b)
  ```
  The first *Passenger* argument is *passed-by-reference* to avoid copy overheads and is passed as a *const* reference to ensure that the state of the *actual argument* stays intact.
  The second *BookingClass* parameter is passed as a *const reference* because the singleton instance of any of its sub-types is available by a *static member function* that returns the singleton as a *const reference*. *const* instance of a derived class cannot be upcasted to a non *const* reference of its base class. Besides, *BookingClass* is an abstract class and passing the parameter by value will be equivalent to *instantiating* a local instance of *BookingClass* which is not possible.
  The logic of this function *might* first check the *eligibility* of the *Passenger* passed as argument for the *Divyaang booking category*, using the method *BookingCategory::Divyaang::IsElligible*. If the *Passenger* is not elligible then a *Bad_Elligibility exception* may be thrown.
  Otherwise, the method may use the *polymorphic member function Divyaang::GetConcessionFactor* from the *Divyaang hierarchy* to get the concession factor applicable to the *Passenger*'s disability type and the *BookingClass* she is travelling in.
- Besides as already discussed, it has *private constructor* and a *private destructor*.

Booking

Inclusion-Parametric Polymorphic Design

- *Booking* hierarchy is rooted at *abstract base class* -- *Booking*
- A *template class* BookingTypes is derived from the abstract base class
- As is clear from the *abstractness* of the base class, this hierarchy is a polymorphic hierarchy that allows *dynamic dispatch* of calls to *polymorphic methods*.

Design Of Abstract Base Class -- Tag Types

- The booking application allows *6 kinds of booking categories* as already discussed in the design of *BookingCategory* polymorphic hierarchy. Corresponding to each *static sub-type* of *BookingCategory*, a *sub-type* of *Booking* is designed. So there are *6 static sub-types* of *Booking* class (*GeneralBooking*, *LadiesBooking*, *SeniorCitizenBooking*, *DivyaangBooking*, *TatkalBooking*, *PremiumTatkalBooking*) and corresponding to each of the *booking types* a *user-defined data type* is defined in the *private* access-specifier as a *placeholder* to tag each of the *booking type*.

  struct Booking::GeneralType
  struct Booking::LadiesType
  struct Booking::SeniorCitizenType
  struct Booking::DivyaangType
  struct Booking::TatkalType
  struct Booking::PremiumTatkalType
- In the *public* access-specifier, the *target sub-types* of the *Booking* (each one of which is instantiated from the derived template class using the appropriate *tag type* from above as the template argument) are *typedef*-ed so that they can be accessed by using *Booking* as the qualifier. For example --

  typedef BookingTypes<GeneralType> GeneralBooking

Design Of Abstract Base Class -- Constructor

- *Booking* has a *protected* constructor.
- *Signature:* `Booking::Booking(const Station &, const Station &, const Date &, const BookingClass &, const BookingCategory &, const Passenger &, const Date &)`
- *Booking* is an abstract class and hence cannot be instantiated to construct a *stand-alone* object (though it will be instantiated when a derived class is instantiated). Therefore there is no need of the constructor outside of the hierarchy and hence it can be avoided to be kept as *public*.

It cannot be kept *private* because it needs to be accessible to the derived class(es) when they are instantiated. Therefore *protected* is chosen as the apt access-specifier.

- Constructors do not have a return type and neither can they be *const* methods.
- *Arguments:* The constructor has *7 arguments* that are used to initialize the respective *non-static data members*.
  BookingCategory and *BookingClass* parameters are passed as a *const reference* because the singleton instance of any of its sub-types is available by a *static member function* that returns the singleton as a *const reference*. *const* instance of a derived class cannot be upcasted to a non *const* reference of its base class. Besides, *BookingClass* and *BookingCategory* are abstract classes and passing the parameters by value will be equivalent to *instantiating* local instances of them which is not possible.
  The other parameters are also *passed by reference* to avoid copy overheads. They are passed as *const* references so that the state of the *actual parameters* stays intact.

Design Of Abstract Base Class -- Destructor

- *Booking* has a *protected* destructor.
- *Signature:* `Booking::~Booking()`
- Before construction of any instance of a derived class, there must be an instantiation of the base class. Every derived class has a base class part in its *object layout* and therefore call to the destructor of a derived class is always followed by the call to the destructor of the base class. For this to happen, the destructor of the base class must actually be accessible from the derived class. Therefore, destructor in a base class should not be *private*.
  If the destructor of any base class, is made *public*, it will become transparent to the client-side. If in the application, the base class destructor is called on the instance of the derived class, the base class part of the object will get destroyed which, in this case, consists of all the *non static data members*.
  Therefore *protected* is chosen as the apt access-specifier.
- The destructor is *virtual (polymorphic).* It is important that in the base class of a *polymorphic hierarchy*, the destructors are also declared as polymorphic. Polymorphic destructors in the base classes enable *dynamic dispatch* of destructors and prevent *object slicing.*
- Destructors do not have a return type and neither can they be *const* methods. They also do not have any arguments.

Operator Overloading

- The *copy-assignment operator '='* in the *derived template class BookingTypes* is blocked (declared in *private* access-specifier).
- This is done because of the *const*-ness of almost all the *attributes* of any object of any class of template *BookingTypes*.

- *Booking* has an overloaded *output streaming operator* that prints all the details of instance of any *static sub-type* of *Booking* on the console.
- *Signature:*
```
std::ostream &operator<<(std::ostream &, const Booking &)
```
- *Friendship:* This method is not a member function but a *friend* function that is declared in the global scope. *Friend functions* can access private (and protected) members of the class in which they are declared as a *friend.*
  Friendship is realized by the keyword *friend* which is precedes the signature of the function when it is declared inside a class.
- *Arguments:* This operator function has two parameters. The first one is of output stream object *std::ostream* type. The standard objects like *std::cout* or *std::cerr* are of this type. The LHS entity in the expression "*cout* << *x*" (that is, *cout*) is passed as the first parameter. This parameter is passed as a *non-const reference*; *non-const* because the state of the output-stream should be changed by the function when it inserts *formatted output* to that stream. The parameter is *passed by reference* because it is imperitive to output the content/message on the same output-stream on which it is intended to by the expression "*cout* << *x*" in the caller function.
  The second arguement is of user-defined type *Booking* which is actually the RHS entity in the same expression. The parameter is *passed by reference* because it is an *abstract class* and pass by value would mean instantiating a local object of *Booking* which is not possible. It is passed as a *const* reference so that the state of the *actual argument*, whose dynamic type can be any of the *static sub types* of *Booking*, stays intact.
  (*NOTE:* Design of *BookingTypes* is discussed later)
- *Return Type:* The return type is *std::ostream*. The same stream that is passed as an arguement is returned to enable *chained output streaming* (like "*cout* << *x* << *y* << *z*"). To ensure that the same stream is returned, the *return is by reference*. It is a ret*urn by non-const-reference* because the state of the returned output stream object might have to change in the caller function in case there is another instruction for formatted output chained with the former one. Return-by-reference is possible here because the returned output stream object is not a local object but rather the same object that was passed-by-reference as the first parameter.

Design Of Abstract Base Class -- Non-Static Data Members

*Booking::fromStation_*	*const Station*
*Booking::toStation_*	*const Station*
*Booking::dateOfBooking_*	*const Date*
*Booking::bookingClass_*	*const BookingClass &*
*Booking::bookingCategory_*	*const BookingCategory &*
*Booking::passenger_*	*const Passenger*
*Booking::dateOfReservation_*	*const Date*
*Booking::pnr_*	*const unsigned int*
*Booking::fare_*	*unsigned int*

Booking class has these *non-static data members*. *Booking::bookingClass_* and *Booking::bookingCategory_* are declared as *const references* of the respective abstract classes *BookingClass* and *BookingCategory* because the *singleton* instance of any of their *static-sub types* is available through a *static member function* that returns that instance as a *const reference*, that cannot be upcasted to a *non-const base class reference*.

All other *data members* (excluding *Booking::fare_*) are also *const* data members because once a *Booking* is done, its details (like destination, departure station, date of booking, passenger etc) cannot be changed. Besides, *Booking::fare_* is a *non-const data member* because a meaningful value can only be assigned to it when the *Booking sub-type* object has been constructed and the *BookingTypes<T>::ComputeFare* method is called on it. If this data member is *const*, once it is assigned a *garbage value* before the *ComputeFare* method is called, the value cannot be over-written.

Design Of Abstract Base Class -- Static Data Members

- *Booking::sBaseFareRate* is a *private static const data member* of *double built-in type*. Its value (*0.5*) is taken from the *master data* in the *specifications*.
- *Booking::sNextAvailablePNR* is a *private static data member* of *unsigned int built-in type*. It keeps track of the number of objects constructed for any *static sub-type* of *Booking*. The *PNR* number of booking must be allocated sequentially starting from one. Therefore it has to be incremented after every instantiation and hence is kept as *non-const*. It is initially initialized to 1.

Design Of Abstract Base Class -- Non-Static Member Functions

- *Booking* has a *public non-static member function* that returns the name/type of the *Booking sub-type*.

- *Signature:* `std::__cxx11::string Booking::GetType() const`
- *Return type:* This method returns the value of the *static data member BookingTypes<T>::sBookingType* of any class of the derived template *BookingTypes* (discussed later) on the instance of which the method is called. This *static data member* is implemented as a *string* and hence the return type.

- *Booking* has a *public non-static member function* that returns the *fare* for any instance of *Booking sub-type* by implementing the appropriate *business logic* on it, as given in the *master data* of *specifications*.
- *Signature:* `unsigned int Booking::ComputeFare() const`
- *Return type:* The fare computed might have some *fraction part* due to the various *factors* that are used in its computation. Before returning it is finally rounded off to the nearest integer. Since the fare will always be *non-negative*, *unsigned int* return type is apt.

- The common features of both the *non-static member functions* include *constness* and *polymorphism*.
- *Constant method:* The *const*-ness of the method is ensured by *const* keyword on the extreme right of the signatures. It ensures that the state of the object as the member of which these methods are called, stays intact.
- Both the methods are *virtual/polymorphic* to enable *dynamic dispatch* to appropriate *static sub-types*. In fact these are *pure virtual* functions in the abstract base class.

Design Of Abstract Base Class -- Static Member Functions

- *Booking* has a *public static member function* that is responsible for first checking the validity of all the arguments that are passed with respect to some obvious constraints and some other implementation constraints related to *railways booking*. Secondly, it acts as a *primary mediator* in the *so-called virtual construction* of a *static sub-type* of *Booking.*
- *Signature:* `static const Booking *Booking::CreateBooking(const Station &, const Station &, const Date &, const BookingClass &, const BookingCategory &, const Passenger &)`
- *Arguments:* *BookingCategory* and *BookingClass* parameters are passed as a *const reference* because the singleton instance of any of its sub-types is available by a *static member function* that returns the singleton as a *const reference*. *const* instance of a derived class cannot be upcasted to a non *const* reference of its base class. Besides, *BookingClass* and *BookingCategory* are abstract classes and passing the parameters by value will be equivalent to *instantiating* local instances of them which is not possible.

The other parameters are also *passed by reference* to avoid copy overheads. They are passed as *const* references so that the state of the *actual parameters* stays intact.

- *Return type:* This method returns (if it does not throw an *exception*) an instance of the *static sub-type* of *Booking* associated with the *dynamic type* of the *BookingCategory* passed as an argument. The return type is therefore *const Booking ** (*implicit upcating*). The return pointer is pointer to a *const* instance because the state of the *Booking* object should not be tampered with.
- The role of this method in the *Virtual Construction Idiom* will be discussed later.
- *Exceptions:* As will be later discussed in the *Virtual Construction Idiom*, this is the method that is called from the client side to construct an object of an *appropriate static sub-type* of *Booking* class associated with the *static sub-type* of *BookingCategory*. This function further calls another function then that one calls another and that one calls another function. So in this case, the erroneous values must be detected in the very first layer of calls in the *virtual construction process*.
    - *Bad_Booking_UndefinedTerminals:* If between the first and second parameters of *Station* type, there is no distance defined in the attributes of the *singleton Railways instance*, *Bad_Booking_UndefinedTerminals exception* is thrown.
    - *Bad_Booking_DateOfBooking:* If the *date of booking* (third arguement) is either on the same day as or before the *current date on the system / date of reservation*, *Bad_Booking_DateOfBooking exception* is thrown.
      If the *date of booking* (third arguement) is after *365 days* from the *current date on the system / date of reservation*, *Bad_Booking_DateOfBooking exception* is thrown.
    - *Bad_Booking_BookingCategory:* If the *booking category* (fifth arguement) does not match with any of the *6 valid BookingCategory sub-types*, *Bad_Booking_BookingCategory exception* is thrown.
    - *Bad_Booking_BookingClass:* If the *booking class* (fourth arguement) does not match with any of the *8 valid BookingClass sub-types*, *Bad_Booking_BookingClass exception* is thrown.
    - *Bad_Booking_Passenger:* If the attributes of the *passenger* (sixth arguement) are not consistent with the *booking category* that is chosen for the booking (in other words, passenger is not eligible for that booking category), *Bad_Booking_Passenger exception* is thrown.

Design Of Derived Template Class -- Static Data Members

- In derived template *BookingTypes*, a *private static const data member* is defined that stores the type of the booking in *string* data type.

```
static template<class T> const std::__cxx11::string
BookingTypes<T>::sBookingType
```

Design Of Derived Template Class -- Constructor

- The template class has a *private constructor*.
- *Signature:* `template<class T> BookingTypes<T>::BookingTypes(const Station &, const Station &, const Date &, const BookingClass &, const BookingCategory &, const Passenger &, const Date &)`
- A constructor does not have a return type and neither can it be a *const* method.
- The constructor has the same arguments as the *protected constructor* of *Booking* class (already discussed).
- The constructor is declared in *private* scope because *we need to control the values that are passed to the constructor as arguments*.
  Any groups of arguements will have to be first checked for a number of possible errors that can make the arguments invalid as far as the construction of a *Booking sub-type* object is concerned. *Constructors* however should never throw exceptions because they might lead to *inconsistent object states*. So the constructor must be declared in the *private* access-specifier and another *static member function* should be used in the public interface to filter the error-free values that are passed to the private constructor (*discussed later*).

Design Of Derived Template Class -- Destructor

- *BookingTypes* has a *public* destructor
- *Signature:* `template<class T> BookingTypes<T>::~BookingTypes()`
- Destructors do not have any arguments or return type and neither can they be *const* methods.

Design Of Derived Template Class -- Static Member Function

- The template has a *public static member function* that acts as an interface between the private constructor and the outside/global scope.
- *Signature:*
  ```
  template<class T> static const BookingTypes<T> *
  BookingTypes<T>::CreateSpecialBooking(const Station &, const Station
  &, const Date &, const BookingClass &, const BookingCategory &,
  const Passenger &, const Date &)
  ```
- The *arguments* are of the same type as the *protected constructor* of *Booking* class (already discussed).
- *Return type:* This method returns an instance of *Booking sub-type* with the same attributes as the passed parameters (if an *exception* is not thrown). Since the *template class* does not have a *copy constructor*, the object has to be *instantiated dynamically* so that it can be returned legally. Therefore the address

of the *dynamic object* is returned and hence the *return type* is *const BookingTypes<T>**.

- *Exceptions:* Though in the *virtual construction process*, this method is not called directly from the application but is called from the *SelectBooking* method of corresponding *static sub-type* of *BookingCategory*, before which all the arguments are already validated in *Booking::CreateBooking* method and appropriate exceptions (if any) are thrown. But since this method is *public*, it might be called directly from the *application* and in this case it must be checked for *bad arguments*. So here too, we have to check for exception so that in no case the system goes to an inconsistent state.
    - *Bad_Booking_UndefinedTerminals:* If between the first and second parameters of *Station* type, there is no distance defined in the attributes of the *singleton Railways instance*, *Bad_Booking_UndefinedTerminals exception* is thrown.
    - *Bad_Booking_DateOfBooking:* If the *date of booking* (third arguement) is either on the same day as or before the *date of reservation* (seventh arguement), *Bad_Booking_DateOfBooking exception* is thrown.
      If the *date of booking* (third arguement) is after *365 days* from the *date of reservation*, *Bad_Booking_DateOfBooking exception* is thrown.
    - *Bad_Booking_BookingCategory:* If the *booking category* (fifth arguement) does not match with the particular *BookingCategory sub-type* associated with the *sub-type* of *Booking*, *Bad_Booking_BookingCategory exception* is thrown.
      (for example booking category passed to *Booking::GeneralBooking::CreateSpecialBooking* has to be *BookingCategory::General::Type()*)
    - *Bad_Booking_BookingClass:* If the *booking class* (fourth arguement) does not match with any of the *8 valid BookingClass sub-types*, *Bad_Booking_BookingClass exception* is thrown.
    - *Bad_Booking_Passenger:* If the attributes of the *passenger* (sixth arguement) are not consistent with the *booking category* (in other words, passenger is not eligible for that booking category), *Bad_Booking_Passenger exception* is thrown.

Design Of Derived Template Class -- Non-Static Member Functions

- The *pure virtual* method *Booking::GetType* in *Booking* class is overridden in the derived template class.
- This method returns the value of the *static data member BookingTypes<T>::sBookingType* of any class of the derived template *BookingTypes* on the instance of which the method is called.
- This method is implemented as an *inline* function.

- The *pure virtual* method *Booking::ComputeFare* in *Booking* class is overridden in the derived template class.
- The method implements for each of the *6* static sub-types of *Booking* the *business logic* as given in the *master data* of the *specifications.*

Virtual Construction Idiom

- In order to construct an object of a particular *static sub-type* of *Booking* class in the application, the *static member function Booking::CreateBooking* is called. This member function does the following two tasks.
    - Checks the validity and consistency of all the arguements. If any argument is found to be invalid/inconsistent, an *exception* is thrown.
    - Calls the *non-static member function BookingCategory::SelectBooking* on the *booking category* parameter, with the same arguments.
- *BookingCategory::SelectBooking* is a *polymorphic method* the call to which dispatches to the *overridden* method in the *BookingCategory sub-type* which was passed as the argument to *Booking::CreateBooking*.

  In this *overridden method* of any *sub-type* of *BookingCategory*, the *public static method CreateSpecialBooking* defined in the *derived template class BookingTypes* is called, for the corresponding *Booking sub-type*.

  In other words, in the implementation of *BookingCategory::General::SelectBooking*, the *static method Booking::GeneralBooking::CreateSpecialBooking* (will be called with the same parameters). Similarly *Booking::LadiesBooking::CreateSpecialBooking* will be called in the implementation of *BookingCategory::Ladies::SelectBooking*.
- Finally, *BookingTypes<T>::CreateSpecialBooking* further calls the *private constructor* of the respective *Booking sub-type* and hence the *Booking sub-type* is instantiated.
- *Refer to the Sequence Diagram attached in UML.pdf file for an even more visual understanding of how does this work*.

Exceptions

Ad-Hoc Polymorphic Design

- *Exceptions* hierarchy is rooted at *base class* -- *std::exceptions*; obviously along with all other exception types thrown by the *standard library*. The part that is designed particularly for this application is a *flat two-level (three* including *std::exceptions) static-polymorphic hierarchy*.
- *7 user-defined exception classes* are derived from the *std::exceptions base class* -- *Bad_Railways*, *Bad_Date*, *Bad_Passenger*, *Bad_Booking*, *Bad_Station*, *Bad_Access* and *Bad_Elligibility*.
- The former *4* classes are further extended into more *specialized* classes, each forming a *hierarchy* of its own. The last *3* are *"stand alone"* classes with no derived classes.
- The various *scenarios* in which the instances of these *exception classes* are thrown are already discussed in great detail in various sections of the document.

Derived Exception Classes

- *Bad_Railways* class has *6 derived class* --
  - *Bad_Railways_NotEnoughStations*,
  - *Bad_Railways_DuplicateStations*,
  - *Bad_Railways_DistBwSameStationsDefined*,
  - *Bad_Railways_NoDefinition*,
  - *Bad_Railways_RepeatedDefinition*,
  - *Bad_Railways_Distance*
- *Bad_Date* class has *4 derived class* --
  - *Bad_Date_Format*,
  - *Bad_Date_Year*,
  - *Bad_Date_Month*,
  - *Bad_Date_Day*
- *Bad_Passenger* class has *6 derived class* --
  - *Bad_Passenger_Name*,
  - *Bad_Passenger_AdhaarNumber*,
  - *Bad_Passenger_MobileNumber*,
  - *Bad_Passenger_DateOfBirth*,
  - *Bad_Passenger_Gender*,
  - *Bad_Passenger_DisabilityType*
- *Bad_Booking* class has *5 derived class* --
  - *Bad_Booking_UndefinedTerminals*,
  - *Bad_Booking_DateOfBooking*,
  - *Bad_Booking_BookingClass*,
  - *Bad_Booking_BookingCategory*,

Bad_Booking_Passenger

Non-Static Data Members

- The classes *Bad_Station, Bad_Access* and *Bad_Elligibility* (that do not have any derived classes) have a *private non-static data member -- description_* of *const char* * type.
- The other classes present on the same level of hierarchy as these (classes that are *specialized/extended* into *derived classes*) have a similar *non-static data member* but is *protected*.
- The derived classes do not have any *non-static data member*.

Non-Static Member Functions

- All the classes in the hierarchy have a *public non-static member function -- what*; that returns the *const char ** data member *description_* (and hence the *return type* of the method is *const char**). For *Bad_Station*, for instance, its *signature* would be
  ```
  const char *Bad_Station::what() const throw()
  ```
- *Constant method:* This is a *const* method that is ensured by the *const* keyword after the *closing parentheses*. This ensures that the state if the object as the member of which this method is called stays intact.

Constructors / Destructors

- All the classes in the hierarchy have a *public constructor* that takes one argument as a *const char ** that is assigned to *description_* data member.
  This argument is given a default value in all the classes. For *Bad_Station*, for instance, its *signature* would be
  ```
  Bad_Station::Bad_Station(const char * = "Bad_Station") throw()
  ```
- All the classes in the hierarchy have a *public destructor*. For *Bad_Station*, for instance, its *signature* would be
  ```
  Bad_Railways::~Bad_Railways() throw()
  ```
- *Constructors* and *destructors* cannot be *const* methods and neither do they have any return types.
- *Destructors* have no arguments.