Project 1: Tracker

Educational Objectives: After completing this assignment the student should have the following knowledge, ability, and skills:

- **Define a class hierarchy using inheritance**
- **Define a class hierarchy using multiple inheritance**
- **Define virtual member functions in a class hierarchy**
- **Implement a class hierarchy using inheritance**
- **Implement a class hierarchy using multiple inheritance**
- **Implement virtual member functions in a class hierarchy**
- **Use initialization lists to call parent class constructors for derived class constructors**
- **State the call sequence for constructors in an inheritance chain**
- **State the call sequence for destructors in an inheritance chain**
- **Define static (compile time) polymorphism**
- **Define dynamic (run time) polymorphism**
- **Use a class hierarchy to implement dynamic polymorphism**

Operational Objectives: Create (define and implement) classes `Box`**,** `Cylinder`**,** `Plane`**,** `Vehicle`**,** `Car`**,** `Truck`**,** `Van`**,** `Tanker`**, and** `Flatbed` **and an object-oriented vehicle counter for use by the Department of Transportation (DOT).**

Deliverables: Seven (7) files: `vehicles.h`**,** `vehicles.cpp`**,** `shapes.h`**,** `shapes.cpp`**,** `verbose.cpp`**,** `tracker.cpp`**, and** `makefile`**.**

The DOT Tracker Project

This project simulates an application called *tracker* for the Department of Transportation (DOT) in which highway traffic data is accumulated in real time using various sensing equipment. The sensors can passively detect a vehicle and actively inquire further data when that vehicle is a truck. For all vehicles a serial number is collected. The serial number can be decoded to determine the vehicle type (car, truck/van, truck/tanker, truck/flatbed), passenger capacity, and, for trucks, the dimensions of its carrier. Trucks actively respond with their DOT license number as well.

Tracker is set up at a specific point on a roadway, for example at a toll booth or a specific segment of interstate highway. Once activated, it keeps a running account of the passing vehicles. It can report summary data and also can keep full reports of all vehicles passing the checkpoint within a certain time block.

Procedural Requirements

1. **Create and work within a separate subdirectory** `cop3330/proj1`**. Review the COP 3330 rules found in Introduction/Work Rules.**

2. **Begin by copying the following files from the course directory** `/home/courses/cop3330p/fall07/` **into your** `proj1` **directory:**

```
proj1/tester.cpp
```

```
proj1/segment0.data
proj1/segment1.data
proj1/segment2.data
submitscripts/proj1submit.sh
area51/tester_s.x
area51/tester_i.x
area51/tracker_s.x
area51/tracker_i.x
```

The naming of these files uses the convention that `_s` **and** `_i` **are compiled from the same cource code on** `program` **(Sun/Solaris) and** `linprog` **(Intel/Linux), respectively. The executables are distributed only for your information and experimentation. You will not use these files in your own project.**

3. **You are to define and implement the following classes:** `Box`**,** `Cylinder`**,** `Plane`**,** `Vehicle`**,** `Car`**,** `Truck`**,** `Van`**,** `Tanker`**, and** `Flatbed`**.**

4. **File** `shapes.h` **should contain the definitions of the classes** `Box`**,** `Cylinder`**, and** `Plane`**. File** `shapes.cpp` **should contain the member function implementations for these classes.**

5. **File** `vehicles.h` **should contain the definitions of the classes** `Vehicle`**,** `Car`**,** `Truck`**,** `Van`**,** `Tanker`**, and** `Flatbed`**. File** `vehicles.cpp` **should contain the implementations for these classes.**

6. **File** `verbose.cpp` **should contain the verbose versions of the various class implementations (both shapes and vehicles).**

7. **Create a client program for all of these classes in the file** `tracker.cpp`**.**

8. **Create a makefile for all of the project in the file** `makefile`**.**

9. **Turn in all seven (7) files** `vehicles.h`**,** `vehicles.cpp`**,** `shapes.h`**,** `shapes.cpp`**,** `verbose.cpp`**,** `tracker.cpp`**, and** `makefile` **using the** `proj1submit.sh` **submit script.**

   Warning: Submit scripts do not work on the `program` ***and*** `linprog` ***servers. Use*** `shell.cs.fsu.edu` ***to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.***

Code Requirements and Specifications - Server Side

1. **You are to define and implement the following classes:**

Class Name:	`Box`
Services (added or changed):	`float Volume() const // returns volume of box object`
Private variables:	`float length_, width_, height_`

Class Name:	`Cylinder`
Services (added or changed):	`float Volume() const // returns volume of cylinder object`
Private variables:	`float length_, radius_`

Class Name:	`Plane`

Services (added or changed):	`float Area() const // returns area of plane object`
Private variables:	`float length_, width_`

Class Name:	`Vehicle`

Services (added or changed):	`const char*  SerialNumber  () const // returns serial number`
	`unsigned int  PassengerCapacity  () const // returns passenger capacity`
	`float  LoadCapacity  () const // returns 0`
	`const char*  ShortName  () const // returns "UNK"`
	`static VehicleType  SnDecode  (const char* sn)`

Private variables:	`char*  serialNumber_;` `unsigned int passengerCapacity_;`

Class name:	`Car`
Inherits from:	`Vehicle`
Services (added or changed):	`const char* ShortName() const // returns "CAR"`

Class name:	`Truck`
Inherits from:	`Vehicle`
Services (added or changed):	`const char*  ShortName  () const // returns "TRK"` `const char*  DOTLicense  () const // returns the license no`
Private variables:	`char* DOTLicense_;`

Class name:	`Van`
Inherits from:	`Truck , Box`
Services (added or changed):	`float  LoadCapacity  () const // returns volume of box` `const char*  ShortName  () const // returns "VAN"`

Class name:	`Tanker`
Inherits from:	`Truck , Cylinder`
Services (added or changed):	`float  LoadCapacity  () const // returns volume of cylinder` `const char*  ShortName  () const // returns "TNK"`

Class name:	`Flatbed`
Inherits from:	`Truck , Plane`
Services (added or changed):	`float  LoadCapacity  () const // returns area of plane` `const char*  ShortName  () const // returns "FLT"`

2. **Each class should have the following:**

   i. **Default constructor**
   ii. **Parametrized constructor that initializes the class variables**
   iii. **Destructor**

iv. **Private copy constructor prototype**
v. **Private assignment operator prototype**
vi. **Follow the notation conventions:**
    a. **Compound names use uppercase letters to separate words** `likeThis` **or** `LikeThis`
    b. **Class, method, and function names begin with upper case letters** `LikeThis`
    c. **Object and variable names names begin with lower case letters** `likeThis`
    d. **Class member variables end with underscore** `likeThis_`

3. **Be sure to make exactly the methods virtual that are needed - that is, those that are overridden in derived classes. Do not make a method virtual unless it is needed virtual.**

4. **During development and testing of the classes, each constructor and destructor should include a line of code that sends an identifying message to standard output. (This requirement serves as a code testing device. These identifying output statements will be removed after development. But leave them in** `verbose.cpp` **when you submit!) For example, the** `Van` **destructor should output the message "**`~Van()`**".**

5. **The user-defined type** `VehicleType` **is an enumerated type:**

Type name:	`VehicleType`
Enumerated values:	`badSn, vehicle, car, truck, van, tanker, flatbed`

6. **The static method** `VehicleType Vehicle::SnDecode(const char* sn)` **returns the vehicle type based on the first (index 0) character of the serial number** `sn` **according to this table:**

```
      sn[0]: 0        1        2        3        4        5        6
VehicleType: badSn    vehicle  car      truck    van      tanker   flatbed
```

7. **After your classes have been fully developed and tested with** `tester.cpp`**:**

    i. **Concatenate the files** `shapes.cpp` **and** `vehicles.cpp` **into the file** `verbose.cpp`
    ii. **Add file documentation to** `verbose.cpp` **as usual**
    iii. `#include shapes.h` **and** `vehicles.h` **into** `verbose.cpp`**, between your file documentation and the beginning of the code**
    iv. **Comment out the verbose output statements in the constructors and destructors in** `shapes.cpp` **and** `vehicles.cpp`**, but leave them in in** `verbose.cpp`
    v. **Make sure that** `verbose.cpp` **compiles to object code with our usual compile command**
    vi. **We are now ready to proceed to client side development.**

Code Requirements and Specifications - Client Side

1. **You are to implement a client program** `tracker` **of the vehicle system described above.**

2. **Tracker processes data from a file that is input through redirection and sends results to standard output. (Thus tracker does not deal directly with files but reads from and writes to standard I/O.)**

3. **Tracker goes through the following processing loop:**

    1. **Read the number of vehicles in the next segment**
    2. **If the number is zero, exit**
    3. **For each vehicle in the segment,**

    1. **Decode the vehicle serial number**
    2. **If other data is needed, read that data**
    3. **Create a vehicle of the appropriate type using the read data**
4. **After all the vehicles in the segment have been read and their corresponding objects created, for each vehicle in the segment:**
    1. **Report the vehicle data to screen**
    2. **Release the memory used to store the vehicle**

4. **Note that the tracker processing loop continues until zero is read for a segment size. It may be assumed that the file of data is correctly structured so that whenever an appropriate item is expected, it is next in the file. For all vehicles, the data will begin with the serial number** `sn` **and then give the passenger capacity** `pc`**. For all specific truck types, the next entry will be the DOTlicense** `DOTL` **followed by the dimension data** `d1 d2 d3(optional)`**. For example, a car, truck, van, tanker, and flatbed would have these lines of data:**

```
sn pc
sn pc DOTL
sn pc DOTL d1 d2 d3
sn pc DOTL d1 d2
sn pc DOTL d1 d2
```

   The dimensional data should be interpreted in the order *d1 = length*, *d2 = width* or *radius*, *d3 = height*. Note that this is more or less self-documenting in the data file `segment0.data`

5. **Tracker should instantiate the objects of a segment using an array whose elements are of type** `Vehicle *`**, that is, pointer to type** `Vehicle`**. At the end of reading the segment data, this array should have pointers to vehicle objects representing the entire segment. These objects should exist until the line in the report representing the object is generated.**

6. **Use declared constants (not hardcoded literal values) for the following:**

    i. **The maximum number of vehicles in a traffic segment (100)**
    ii. **The maximum number of characters in a vehicle serial number (20)**
    iii. **The maximum number of characters in a truck DOT license (20)**

7. **Check for a segment size greater than tracker can handle, and exit if that happens. Thus tracker would exit if either size 0 is read or some size greater than the declared constant 6.i above.**

8. **Your** `tracker.cpp` **source file should** `#include <vehicles.h>`**, but *not* any of the other project files. Your** `makefile` **should create separate object files** `vehicles.o`**,** `shapes.o` **, and** `tracker.o` **and then create the executable** `tracker.x`**.**

9. **Do not submit "verbose" classes for tracker.**

Hints

- **Model executables** `tester.x` **and** `tracker.x` **are for your information only - they are not needed for your project. However,** `tester.cpp` **is indispensable: This is one of the test programs that will be used to assess your project. Moreover, it will help you debug your classes and gives some example code that can serve as a model for your client** `tracker` **program.**

- **To execute tester, enter a serial number at the prompt. The first digit determines the vehicle type. Tester uses the verbose implementations, so you should see all of the constructor and destructor calls for the**

selected type.

- **To execute tracker on a data file use redirection. For example, enter**
  ```
  prompt> tracker.x < segment2.data
  ```
  to run `tracker.x` **with the data file** `segment2.data` **as input.**

- **All destructors should be declared** `virtual`**.** ***Be sure you understand why. There will likely be an exam question related to this.***

- **It is wise to develop and test all of the classes prior to working on any code for** `tracker.cpp`**. This way you can concentrate on class development - some call this "wearing your server hat".** ***After*** **you are satisfied that your classes are correct and ready for submission, including all documentation etc, only then do you switch to your "client" hat and start working on** `tracker`**.**

  If you were starting from "scratch", you would first write a test harness like `tester.cpp` **that exercises your classes and allows you to debug them as a system.**

- **The program** `tester.cpp` **is central to your effort in several ways:**

  i. **It allows you to debug your classes, prior to even thinking about** `tracker`**. This way, you are wearing your "server" hat while classes are being created. (One exception: memory management issues may not be exposed by tester.)**
  ii. **After the classes are debugged and working correctly, you change to your "client" hat and start working on the** `tracker` **program. The file** `tester.cpp` **has model code that you should understand in every detail and use as a model for your** `tracker` **code.**
  iii. ***Note: you will likely see code similar to*** `tester` ***on an exam.***

- **To compile tester, a command line compile is all that is required, because the source code for the dependent files is included into the client program:**
  ```
  prompt> g++ -I. -Wall -Wextra -otester.x tester.cpp
  ```
  Note, this is different from the way you should compile tracker, which requires a makefile to manage separate compilation of all the code files.

- **Your classes will be indepently tested with client programs written to the interfaces defined above.**

- **Run the distributed executables for tester and tracker in** `LIB/area51/` **to see how your programs are expected to behave.**