Project 4: SunPass Vehicle Classes

A class framework for the SunPass project

Revision dated 02/06/18

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

Operational Objectives: Create (define and implement) classes `Box`, `Cylinder`, `Plane`, `Vehicle`, `Car`, `Truck`, `Van`, `Tanker`, and `Flatbed`.

Deliverables: Six (6) files: `vehicles.h`, `vehicles.cpp`, `shapes.h`, `shapes.cpp`, `makefile`, and `log.txt`

Assessment Rubric

```
student builds:
 stester.x                          [0..5]:   x
 vtester.x                          [0..5]:   x
assessment builds:
 stester.x                          [0..5]:   x
 vtester.x                          [0..5]:   x
testing:
 stester.x stester.com              [0..5]:   x
 vtester.x vtester.com             [0..25]:  xx
code quality and requirements     [-30..00]: (xx)
dated submission deduction      [2 pts per]: (xx)
                                             ---
total                              [0..50]:  xx
```

The SunPass Tracker Project

This project simulates an application called *tracker* for the Florida Turnpike Authority in which data from SunPass transponders is accumulated in real time using various sensing equipment. The sensors detect a SunPass-equiped vehicle and actively inquire further data when that vehicle is a truck. (The data is used, among other things, to charge a passage toll on the vehicle's SunPass account, thus eliminating the need to stop at toll booths. SunPass is valid on all toll roads and bridges in Florida.) 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, near a toll booth or a specific segment of limited access highway. Once activated, it keeps a running account of the SunPass equipped passing vehicles. It can report summary data and also can keep full reports of all vehicles passing the checkpoint within a certain time block. It also keeps track of individual toll charges and can produce a summary of the charges accumulated in a segment.

The current assignment focusses on the "server side" of the SunPass project: creating the various classes that are used by a client program to collect data and tolls.

Procedural Requirements

1. Create and work within a separate subdirectory `cop3330/proj4`. Review the COP 3330 rules found in Introduction/Work Rules.

2. Begin by copying all files from the course project directory into your `proj4` directory. These should include:

   ```
   proj4/deliverables.sh
   proj4/stester.cpp
   proj4/vtester.cpp
   ```

   Also copy the sample executables from area51:

   ```
   area51/stester_i.x
   area51/vtester_i.x
   ```

   Change the permissions on the area51 copies to executable:

   ```
   chmod 700 *.x
   ```

   It is a good idea to start by looking at the source code for the two test programs and running the area51 executables to get a feel for the testing environment. You can run these in batch mode with a com file:

   ```
   stester_i.x stester.com
   vtester_i.x vtester.com
   ```

3. Begin your log file named `log.txt`. (See Assignments for details.)

4. You are to define and implement the following classes: `Shape`, `Box`, `Cylinder`, `Rectangle`, `Vehicle`, `Car`, `Truck`, `Van`, `Tanker`, and `Flatbed`.
5. File `shapes.h` should contain the definitions of the classes `Shape`, `Box`, `Cylinder`, and `Rectangle`. File `shapes.cpp` should contain the member function implementations for these classes.
6. 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.
7. Turn in the files `vehicles.h`, `vehicles.cpp`, `shapes.h`, `shapes.cpp`, `makefile`, and `log.txt` using the submit script.

   It is assumed that you have an executable copy of the most current submit script `LIB/scripts/submit.sh` in your `~/.bin` available as a command.

   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:	`Shape`

Services (added or changed):	`const char* Name () const // returns "generic"`
	`float Volume () const // returns volume of object`
	`float Area () const // returns area of object`

Protected variables:	`float x_, y_, z_; // dimensions of shape`
	`bool verbose_; // default value 0 given in constructor prototype`

Class Name:	`Box`
Inherits from:	`Shape`

Services (added or changed):	`const char* Name () const // returns "box"`
	`float Volume () const // returns volume of box object`
	`float Area () const // returns surface area of box object`

Class Name:	`Cylinder`
Inherits from:	`Shape`

Services (added or changed):	`const char* Name () const // returns "cylinder"`
	`float Volume () const // returns volume of cylinder object`
	`float Area () const // returns surface area of cylinder object`

Class Name:	`Rectangle`
Inherits from:	`Shape`
Services (added or changed):	`const char* Name () const // returns "rectangle"` `float Area () const // returns area of rectangle object`

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"`
	`float Toll () const // returns toll using fee schedule`
	`static VehicleType SnDecode (const char* sn)`

Private variables:	`char* serialNumber_;` `unsigned int passengerCapacity_;`
Protected variable:	`bool verbose_; // default value 0 given in constructor prototype`

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"`
	`float Toll () const // returns toll using fee schedule`
	`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 , Rectangle`
Services (added or changed):	`float LoadCapacity () const // returns area of rectangle` `const char* ShortName () const // returns "FLT"`

2. Each class should have the following:

   i. Default constructor
   ii. Parametrized constructor that initializes the class variables; default value for `verbose_` is `0 = false`. Put the default value in the constructor prototype.
   iii. Destructor
   iv. Private copy constructor prototype *with no implementing code*
   v. Private assignment operator prototype *with no implementing code*
   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_`

   Note that (iv) and (v) together ensure that copies of objects cannot be made. Privatizing the copy constructor and assignment operator prototypes will cause a compiler error if client code attemps to make a copy, and not providing implementing code will cause a link error if an attempt is made to copy objects anywhere in the class implementations.

3. Note that `Vehicle::verbose_` is protected so that derived classes can access it directly.

4. 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.

5. The toll fee schedule is:
   minimum for all vehicles: $2.00
   all trucks: $10.00

6. For development and testing of the classes, each constructor and destructor should include a line of code that conditionally sends an identifying message to standard output, whenever `verbose_` is `1 = true`. For example, the `Van` destructor should output the message "`~Van()`" if `Vehicle::verbose_` is true.

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

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

8. 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
```

9. After your classes have been fully developed and debugged, so they compile without warnings, it is time to test with the tester programs.

   Thoroughly test your vehicle objects with this program. Note that this program prompts you for a serial number. The serial number is decoded to get a vehicle type, and an object of that type is created dynamically. You should see the constructor calls displayed, in correct order, because tester creates the objects with "verbose" set to 1 = true. Then the methods of this object are called. You should see correct serial number (and, for trucks, dot license) displayed. An "OOPS" message may be displayed if a problem is detected with your constructors. Finally the object is deleted, and you should see the destructors called in correct order. Read the source code in the tester programs both to understand how it works.

Hints

- You will need a value for π = "Pi" for calculating the volume of a cylinder. This is supported by the math library under the name `M_PI`:

  ```
  #include <cmath> // M_PI defined to 21 significant digits
  ```

- The example executables `stester.x` and `vtester.x` are for your information only - it is not needed for your project. However, the source files `stester.cpp` and `vtester.cpp` are indispensable: These are 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 in the next assignment.

- To execute tester, enter a serial number at the prompt. The first digit determines the vehicle type. Tester uses the verbose = 1, so you should see all of the constructor and destructor calls for the selected type.

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

- The provided testing source code is central to your effort in several ways:

  i. It allows you to debug your classes, prior to even thinking about the `tracker` client program (to be developed in the next assignment). 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 `vtester.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 `vtester` on an exam.*

- To compile any component of the project, use the makefile and mention a specific target. For example, to debug the syntax in `shapes` enter the command: `make shapes.o`.

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

- Run the distributed executables in `LIB/area51/` to see how your programs are expected to behave.