Project 5: SunPass Tracker

Adding the Tracker application to 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 static (compile time) polymorphism
- Define dynamic (run time) polymorphism
- Use a class hierarchy to implement dynamic polymorphism

Operational Objectives: Create an object-oriented vehicle counter for use by the Department of Transportation (DOT).

Deliverables: `tracker.cpp`, `makefile`, `log.txt`

Ancillary Deliverables: This project depends on copies of these files delivered with the previous project: `vehicles.h`, `vehicles.cpp`, `shapes.h`, and `shapes.cpp`. These will also be collected by the submission process.

Assessment Rubric

```
-----------------------------------------------------
student build: (3 pts)
 stester.x                            [0..1]:    x
 vtester.x                            [0..1]:    x
 tracker.x                            [0..1]:    x
project build: (3 pts)
 build stester.x                      [0..1]:    x
 build vtester.x                      [0..1]:    x
 build tracker.x                      [0..1]:    x
compatibility of framework: (4 pts)
 diff shapes.h                        [0..1]:    x
 diff shapes.cpp                      [0..1]:    x
 diff vehicles.h                      [0..1]:    x
 diff vehicles.cpp                    [0..1]:    x
project tests: (40 pts)
 stester.x stester.com                [0..5]:    x
 vtester.x vtester.com                [0..5]:    x
 tracker.x < group1.in                [0..5]:    x
 tracker.x < group2.in                [0..5]:    x
 tracker.x < group3.in                [0..5]:    x
 tracker.x < group4.in                [0..5]:    x
 tracker.x < group_wide.in            [0..5]:    x
 tracker.x < group.paragraph          [0..5]:    x
software engineering:
 code quality, requirements        [-30..0]:   ( x)
 code standards                    [-10..0]:   ( x)
dated submission deduction       [2 pts per]:   ( x)
                                                 --
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 "client side" of the SunPass project. Using the various DOT Vehicle classes built in the preceding assignment, we build the tracker client program.

Procedural Requirements

1. Create and work within a separate subdirectory `cop3330/proj5`. Review the COP 3330 rules found in Introduction/Work Rules.
2. Begin by copying the following all files from the course home: into your `proj5` directory. You should see at least the following:

   ```
   proj5/deliverables.sh
   proj5/group0.data
   proj5/group1.data
   proj5/group2.data
   proj5/group3.data
   proj5/group_wide.data
   proj5/group3.paragraph
   proj5/makefile.partial
   ```

   Also copy `area51/tracker_i.x` and change its permissions to executable. `tracker_i.x` is the benchmark executable against which yours will be tested. Note that you can add these lines to your makefile to ensure that you have a current executable benchmark:

   ```
   tracker_i.x: /home/courses/cop3330p/LIB/area51/tracker_i.x
   <TAB>   cp  /home/courses/cop3330p/LIB/area51/tracker_i.x .
   <TAB>   chmod 700 tracker_i.x .
   ```

   (Also add "tracker_i.x" to the dependencies for the first target.)
3. Begin your log file named `log.txt`. (See Assignments for details.)
4. Begin by copying the various shapes and vehicles files from the preceding project: `shapes.h`, `shapes.cpp`, `vehicles.h`, and `vehicles.cpp`. Be sure that these files remain identical to those from the preceding project.
5. If in the current project you find that the shape and vehicle classes require modification, be sure that you make the modifications in both projects and resubmit the preceding one, so that when we check the requirement above your submissions will pass that check.
6. Create a client program for all of these classes in the file `tracker.cpp`.
7. Turn in the files `tracker.cpp`, `makefile`, and `log.txt` using the 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 - 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 data read in the previous steps
        4. Update various summary information for this segment
    4. After all the vehicles in the segment have been read and their corresponding objects created, report a summary of the various vehicles by type, along with the totals of tonnage and tolls of the segment.
    5. After the summary, report the details: for each vehicle in the segment:
        1. Report the vehicle data to screen
        2. Release the memory used to store the vehicle

    When in doubt, use the distributed area51 executables as a guide to output data and formatting.

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 `group0.data`. *Note also that we will assume that each vehicle and its data are in a separate line of the input file.*

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. NOTE: instantiate the objects with the "verbose" variable set to 0 = false (the default).

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 (50)
    iii. The maximum number of characters in a truck DOT license (50)

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. Your tracker project will be tested using the classes that you have previously submitted for the previous project. If you need to revise those, you will need to submit the revirions for that project prior to submitting this project.

10. Your tracker project will be tested using the classes that you have previously submitted for the previous project. If you need to revise those, you will need to submit the revirions for that project prior to submitting this project.

11. Your executables should produce output that is identical to that produced by the corresponding benchmark programs. In particular, these two command:

```
tracker.x < segment3.data
tracker.x < segment3.paragraph
```

should produce identical output.

Hints

- Model executable `tracker_i.x` is for your information only - it is not needed for your project.
- To execute tracker on a data file use redirection. For example, enter
  `prompt> tracker_i.x < group2.data`
  to run `tracker_i.x` with the data file `group2.data` as input.
- Run the distributed executables for tester and tracker in `LIB/area51/` to see how your programs are expected to behave.