CSE240 C/C++ Assignment 3

Inheritance & Polymorphism

Topics:

- Object orientation fundamentals
- Application of the 4 pillars of OOP
- Use of Inheritance
- Use of Polymorphism
- Arrays

Description

The goal of this assignment is to create a simple Race with different terrains and different racers. You will make use of a similar technique to the terrain island assignment to generate a racetrack of different road pieces and have polymorphic cars race to the end!

Use the following Guidelines:

- Give identifiers semantic meaning and make them easy to read (examples numStudents, grossPay, etc).
- Keep identifiers to a reasonably short length.
- User upper case for constants. Use title case (first letter is upper case) for classes. Use lower case with uppercase word separators for all other identifiers (variables, methods, objects).
- Use tabs or spaces to indent code within blocks (code surrounded by braces). This includes classes, methods, and code associated with ifs, switches and loops. Be consistent with the number of spaces or tabs that you use to indent.
- Use white space to make your program more readable.

Important Note:

All submitted assignments must begin with the descriptive comment block. To avoid losing trivial points, make sure this comment header is included in every assignment you submit, and that it is updated accordingly from assignment to assignment.

Libraries:

You are allowed to use #include<string> if needed and allowed to use std::queue, std::stack and std::vector as needed.

UML Reminders:

- # means protected
- + means public
- - means private
- <method name>(<parameter>:<type>, …):<return type>

Programming Assignment:

Instructions:

You are creating software to simulate a race. This race will be a straight lane race from start to finish with the obstacles being different segments of road along the way.

You will be generating the race course and having different racers race on the road. Each racer will have different strengths and weaknesses for making progress in the race.

This program will be structured around a "game loop" to drive the software. In the game loop you will:

- Loop through each racer and have them make progress in the current segment
- Check for a winner

Classes:

- RoadSegment - Abstract base class
  - Asphalt
  - Crumbled
  - Gravel
  - Dirt
- Racer
  - Street
  - Steady
  - Rough
- Race
  - Runs the race logic
- Main
  - Builds a Race object and tells it to run the race
  - Don't want any real logic in the main method

You will create a Makefile for this assignment.

Racers should be in their own library (.h/.cpp combo)

Road Segments should be in their own library (.h/.cpp combo)

Race should be its own .h file (.h/.cpp combo)

main.cpp should be very simplistic, only instantiating a Race and calling a method to run the race.

Specifications:

Part 1 – Polymorphism Specifications – Build these classes

Road Segments

The Road Segments are going to help procedurally generate the race. Each Road Segment has a probability distribution of what kinds of other segments should be immediately after.

This is accomplished by calling the generateNeighbor() method of the object directly before the index you're currently trying to fill. The generateNeighbor() method should "roll a die" and return a new Road Segment object based on the probability distribution it has. This is a very simple method that is overridden in each segment, don't over think it.

Let's talk about the race road classes first:

Classes:

- RoadSegment
  - Abstract Base Class
  - Properties:
    - #length:int
      - # means protected
    - #modifier:double
  - Methods:
    - +generateNeighbor():RoadSegment
    - +getLength():int
    - +getModifier():int
- AsphaltSegment
  - Properties
    - modifier set to 1.0
  - Methods
    - generateNeighbor
      - 60% - AslphaltSegment
      - 25% - CrumbledSegment
      - 10% - GravelSegment
      - 5% - DirtSegment
- CrumbledSegment
  - Properties
    - Modifier set to between 0.6 and 0.8
  - Methods
    - generateNeighbor
      - 40% - CrumbledSegment
      - 25% - AslphaltSegment
      - 25% - GravelSegment
      - 10% - DirtSegment

RoadSemgents Continued:

- GravelSegment
  - Properties
    - Modifier set to between 0.3 and 0.6
  - Methods
    - generateNeighbor
      - 50% - GravelSegment
      - 35% - DirtSegment
      - 10% - CrumbledSegment
      - 5% - AsphaltSegment
- DirtSegment
  - Properties
    - Modifier set to between 0.0 and 0.3
  - Methods
    - generateNeighbor
      - 60% - DirtSegment
      - 30% - GravelSegment
      - 5% - CrumbledSegment
      - 5% - AsphaltSegment

The length of each RoadSegment should be decided by the default constructor of the parent class (RoadSegment). RoadSegments should be between 10 and 50 in length.

The modifiers will be handed to the cars and change how much progress per run the car can do.

Each class should also have a toString():string and/or cout<< overload to output the type and length of the segment.

Example:

```
cout << raceTrack[4].toString(); //might be racetrack[4]->toString()
- or -
cout << racetrack[4]; //with overloaded cout<<
//assuming a DirtSegment is stored in index 4…
```

Would output:

```
Dirt – 27 units
```

Racer Specifications

The racers are also going to be polymorphic. We're trying to avoid too much coupling so the racers are going to make progress in their road segment by taking in a progress modifier. The modifier will (of course) be provided by the segment they are currently racing on.

This means the primary point of polymorphism for the racers will be their individual progress equation.

Classes:

- Racer – abstract base class
  - Properties
    - #carNumber:int
    - #speed:double
    - #currentProgress:double
  - Methods:
    - +getCarNumber:int
    - +getSpeed:double
    - +getCurrentProgress:double
    - +resetProgress:void
      - Sets progress back to 0.0
    - +makeProgress(double modifier):void
- StreetRacer
  - Properties
    - speed should be set to 5.5~7.0
  - Methods
    - makeProgress(double modifier)
      - The modifier passed in should directly modify the speed
      - Progress += (speed*modifier)+0.5
        - The 0.5 is for a minimum amount of progress on bad roads and a slight bonus on good roads
- SteadyRacer
  - Properties
    - Speed should be set to 3.0~4.0
  - Methods
    - makeProgress(double modifier)
      - The steady racer ignores all modifiers
      - Progress += speed
- ToughRacer
  - Properties
    - speed should be set to 2.0~3.0
  - Methods
    - makeProgress(double modifier)
      - local variable bonusSpeed:double
      - bonusSpeed set to 5 * (1.0 – modifier)
      - progress += speed + bonusSpeed

Racer Continued

Each racer should have a toString():string and/or a cout<< overload to output their carNumber, type and current progress.

Example:

```
cout << racers[2].toString();  //might be racers[2]->toString()
-  or -
cout << racers[2];
//assuming a SteadyRacer is index 2
```

Would output:

```
Racer #3 Steady Car - 17 units
```

Part 2 – Set up the Race Class

Your Race Class is where the action happens. You should notice that the racers know nothing about the race, they just exist to make progress. It's up to the Race Class to keep track of what's going on.

Generate the Race Track:

First you'll want to generate your road as a single dimension array of RoadSegments. Essentially the race is linear. We've done this before – you are just generating a single dimension strip of segments, so randomly decide first item and then "ask left" for each subsequent segment.

You should give the user the option to directly input how many segments are in the race, or randomly decide.

```
Welcome to the Racer Derby!
Would you like to:
1 - Determine the length of the race
2 - Run a random race
-1 - Exit
```

If the user chooses the length, size the array to that many segments and generate the segments array.

If the user chooses random, then choose a number of segments between 10 and 50 and generate the array.

Generate the Racers:

Create an array of 8 random racers. Also create a parallel array of integers that keeps track of which segment the racer is racing on.

Part 3 - Run the Race:

Write a method to run the race.

In this method you should run a loop that is nearly infinite. Use something like: while(noWinner) as a condition.

Each time through the loop:

- Update a loop counter
- Loop through each racer and see if they should move up to the next segment.
    - If the racer's current progress >= the segment's length
        - Reset the racer's progress
        - Increase their segment tracker (parallel array) by one
    - If a racer's segment tracker value >= number segments in the race, they are considered to be the winner.
        - Save the racer's index value as the winner
        - Set the conditions to break your race loop
            - Yes, this gives the advantage to the earlier index racers.
- If there was no winner:
- Loop through all of the racers and call their makeProgress() method
    - Pass in the modifier from the current segment they are on
    - Once again, use that parallel array that is keep track of what segment they are on to get that value
- End of the loop:
    - Output some sort of divider that outputs the loop counter
    - Output the track
    - Output all the racers and their progress

Don't forget to announce the winner at the end!

Recommended Helper Methods for Race

OutputTrack()

Write a method to output your track:

Sample output:

```
Track:
A – A – C – G – D – G – G – D – G – C – C – A – A – A – C - A
```

If you want, also include the length of each segment

```
A:15 – A:12 – C:20 …
```

OutputRacers()

Write a method in your Race Class that will pleasantly output the racers, their current segment and their current progress on that segment.

Make use of your toString() and/or cout<< overload!!

Example:

Racers:

```
Racer #1 Street Car – 5 units into Segment #4 Dirt – 27 units long
Racer #2 Street Car – 5 units into Segment #4 Dirt – 27 units long
Racer #3 Steady Car – 6 units into Segment #3 Gravel – 18 units long
Racer #4 Tough Car – 3 units into Segment #3 Crumbled – 20 units long
…

Etc…
```

OutputDivider()

Write a method to output a divider between updates… something like:

```
########################################
Update 35!
----------
```

Notes:

The goal of this is to practice good "separation of responsibility/knowledge" which is why the racers don't keep track of what segment they are on or what kind of segment they are on. Theoretically the racers can be used elsewhere. This is why you need a parallel to keep track of the segment.

If you want to be clever about tracking that (with a struct or something) feel free, but don't give the Racers any more knowledge than needed.

All of the values in here are arbitrarily made up from the top of my head. I have not done any fine tuning.

If you want to tweak the numbers and equations, be my guest! However, you should document the changes as part of your Header Comment on the Race Class

The race is somewhat deterministic with duplicates of the 3 cars acting nearly identically due to very little randomizers. Feel free to add some small randomizers to change things up. One place you might do this is by adding a changeSpeed method that would re-randomize the speed of the car.

The class specifications are by no means set in complete stone. If you want to add methods and such to help with calculations, algorithms, etc. feel free!

If you want to change things up a little to make it 'more interesting' … run those changes by the instructor first.

Be creative, but useful with your output. All my sample output is just that *sample.* Think of your user and how you want to make your software look (while at the same time meeting the specifications).

Extra Credit Opportunities:

- +1 – Don't do the racer's segment updates in order. Come up with a FAST (no slower than O(n) time) technique to randomize the order in which those checks happen so that way early index racers don't always have the advantage
- +2 – Add 3 more types of things to the race. This should be at least 1 racer. Use appropriate inheritance and polymorphism.
- +1 – Add color to your output and make it fun!
- +1 – Add a command line option to load a racetrack from a file:
    - <exe> <racetrack file>
    - The racetrack file should just consist of a number of characters to represent each segment
    - Include a test file with your assignment submission

Grading of Programming Assignment

The TA will grade your program following these steps:

(1) Compile the code. If it does not compile a U or F will be given in the Specifications section. This will probably also affect the Efficiency/Stability section.

(2) The TA will read your program and give points based on the points allocated to each component, the readability of your code (organization of the code and comments), logic, inclusion of the required functions, and correctness of the implementations of each function.

Rubric:

	Levels of Achievement						
Criteria	A	B	C	D	E	U	F
Specifications Weight 50.00%	100 % The program works and meets all of the specifications.	85 % The program works and produces the correct results and displays them correctly. It also meets most of the other specifications.	75 % The program produces mostly correct results but does not display them correctly and/or missing some specifications	65 % The program produces partially correct results, display problems and/or missing specifications	35 % Program compiles and runs and attempts specifications, but several problems exist	20 % Code does not compile and run. Produces excessive incorrect results	0 % Code does not compile. Barely an attempt was made at specifications.
Code Quality Weight 20.00%	100 % Code is written clearly	85 % Code readability is less	75 % The code is readable only by someone who knows what it is supposed to be doing.	65 % Code is using single letter variables, poorly organized	35 % The code is poorly organized and very difficult to read.	20 % Code uses excessive single letter identifiers. Excessively poorly organized.	0 % Code is incomprehensible
Documentation Weight 15.00%	100 % Code is very well commented	85 % Commenting is simple but solid	75 % Commenting is severely lacking	65 % Bare minimum commenting	35 % Comments are poor	20 % Only the header comment exists identifying the student.	0 % Non existent
Efficiency Weight 15.00%	100 % The code is extremely efficient without sacrificing readability and understanding.	85 % The code is fairly efficient without sacrificing readability and understanding.	75 % The code is brute force but concise.	65 % The code is brute force and unnecessarily long.	35 % The code is huge and appears to be patched together.	20 % The code has created very poor runtimes for much simpler faster algorithms.	0 % Code is incomprehensible

What to Submit?

You are required to submit your solutions in a compressed format (.zip). Zip all files into a single zip file. Make sure your compressed file is labeled correctly - <lastname>_<firstname>_assn4.zip

The compressed file MUST contain the following:

- <lastname>_race.h & <lastname>_race.cpp
- <lastname>_racers.h & <lastname>_racers.cpp
- <lastname>_segments.h & <lastname>_segments.cpp
- <lastname>_<firstname>_main.cpp
- Makefile

Any other necessary files such as color libraries or extra credit test files should be in the compressed folder.

If multiple submissions are made, the most recent submission will be graded, even if the assignment is submitted late.

Where to Submit?

All submissions must be electronically submitted to the respected homework link in the course web page where you downloaded the assignment.

Academic Integrity and Honor Code.

You are encouraged to cooperate in study group on learning the course materials. However, you may not cooperate on preparing the individual assignments. Anything that you turn in must be your own work: You must write up your own solution with your own understanding. If you use an idea that is found in a book or from other sources, or that was developed by someone else or jointly with some group, make sure you acknowledge the source and/or the names of the persons in the write-up for each problem. When you help your peers, you should never show your work to them. All assignment questions must be asked in the course discussion board. Asking assignment questions or making your assignment available in the public websites before the assignment due will be considered cheating.

*The instructor and the TA will **CAREFULLY** check any possible proliferation or plagiarism. We will use the document/program comparison tools like MOSS (Measure Of Software Similarity: http://moss.stanford.edu/) to check any assignment that you submitted for grading. The Ira A. Fulton Schools of Engineering expect all students to adhere to ASU's policy on Academic Dishonesty. These policies can be found in the Code of Student Conduct:*

http://www.asu.edu/studentaffairs/studentlife/judicial/academic_integrity.htm

ALL cases of cheating or plagiarism will be handed to the Dean's office. Penalties include a failing grade in the class, a note on your official transcript that shows you were punished for cheating, suspension, expulsion and revocation of already awarded degrees.