COMP 2404 – Assignment #3

Due: Thursday, November 19 at 11:59 pm

1. Goal

For this assignment, you will write a C++ program to implement a simplified version of the Strategy design pattern. Your program will simulate a race between a tortoise and hare, using behaviour classes to implement different types of runner movements.

2. Learning Outcomes

With this assignment, you will:

- apply the OO concepts of inheritance and polymorphism
- practice drawing a UML class diagram
- implement a simplified version of the Strategy design pattern
- work with virtual and pure virtual functions in C++

3. Instructions

3.1. Draw a UML class diagram

We are going to implement the Strategy design pattern in a hierarchy of move behaviours, similar to the hierarchy of dance behaviours from the in-class coding example of section 3.4, program #8. The program will represent a collection of runners making their way from the left-hand side of the screen to the right-hand side, and each runner's move behaviour will change randomly at every step.

To do this, you will implement an abstract `MoveBehaviour` class, along with five concrete sub-classes: the `WalkBehaviour` class will cause a runner to move forward by one position, the `SleepBehaviour` class will cause a runner to stay in their current position, the `JumpBehaviour` class will cause a runner to move forward by a random number of positions, the `SlideBehaviour` class will cause a runner to slip backwards by a random number of positions, and the `TeleportBehaviour` class will cause a runner to move to a random location on the screen.

The abstract `MoveBehaviour` class will require a pure virtual `move()` function, and the appropriate behaviour will be implemented in the concrete sub-classes.

Your program will implement a `Race` class that serves as the control object, and a `View` class that handles all output to the screen. The `Race` class will store a collection of `Runner` objects corresponding to the runners of the race, and each `Runner` object will store a `MoveBehaviour` object that computes the runner's next move.

You will begin by drawing a UML class diagram, drawn by you using a drawing package of your choice, and your diagram will follow the conventions established in the course material covered in section 2.3. Your diagram will represent all the classes in your program, including the control, view, and entity classes, as well as the hierarchy of move behaviour classes. Your diagram will show all attributes, all operations (including the parameters and their roles such as in, out, in-out), where applicable. As always, do not show collection classes, as they are *implied* by multiplicity. Do not show other objects as attributes, as these must be shown as associations. Do not show simple getters, setters, constructors, or destructors.

While you must begin drawing your UML diagram in this step, you will need to update it throughout the instructions below. The diagram that you submit with your assignment must match your completed program.

3.2. Implement the `Position` class

You will begin by implementing a new `Position` class that contains the following data members:

3.2.1. a row, as an integer

3.2.2. a column, as an integer

The `Position` class will contain the following member functions:

3.2.3. a constructor that initializes the data members to default values

3.2.4. getter member functions for both data members

3.2.5. setter member functions for both data members

3.3. Implement the `MoveBehaviour` classes

You will implement the hierarchy of `MoveBehaviour` classes, which you will model on the `DanceBehaviour` classes that we worked on during the lectures. You can find these classes in the coding example posted in *cuLearn*, in section 3.4, program #8.

You will implement the `MoveBehaviour` classes as follows:

3.3.1. You will create a new `MoveBehaviour` abstract class that serves as the base class for the other behaviour classes. The `MoveBehaviour` class will contain a pure virtual member function, as follows:

(a) `void move(Position& oldPos, Position& newPos, string& log)` computes a new position `newPos` for the runner, given its current position `oldPos`; the `move()` function will return a short, text description of the move in the `log` parameter, which will be printed to the screen in a later step

3.3.2. You will create five concrete move behaviour classes, which derive from the `MoveBehaviour` class, as follows:

(a) the `WalkBehaviour` class will have a `move()` function that keeps the runner in the same row and moves it one column to the right

(b) the `SleepBehaviour` class will have a `move()` function that keeps the runner in the same row and column

(c) the `JumpBehaviour` class will have a `move()` function that keeps the runner in the same row and moves it forward by a random number between 1 and 8 of columns to the right

(d) the `SlideBehaviour` class will have a `move()` function that keeps the runner in the same row and moves it backward by a random number between 1 and 5 of columns to the left

(e) the `TeleportBehaviour` class will have a `move()` function that moves the runner to a valid position at a random row and a random column

NOTE: You may keep the move behaviour class definitions in a single header file, and the corresponding member function implementations in a single source file.

3.4. Implement the `Runner` class

You will implement a new `Runner` class that contains the following data members:

3.4.1. the runner's name, as a `string` object

3.4.2. the runner's avatar, as a `char`

3.4.3. the runner's bib number, as an integer; this is the number pinned to the runner's shirt that uniquely identifies them

3.4.4. the runner's lane number, as an integer; this is the row on the screen where the runner will be at the beginning of the race

3.4.5. the runner's current position, as a `Position` object

3.4.6. a pointer to the runner's current `MoveBehaviour` object

3.4.7. the current log, as a `string` object, describing the runner's last move

The `Runner` class will contain the following member functions:

3.4.8. a constructor that takes a name, avatar, bib number, and lane number as parameters, and initializes all the data members; this includes:
   (a) initializing the runner's current position to the row indicated by the lane number, and a column set to zero to represent the left-hand side of the screen
   (b) initializing the move behaviour to walking behaviour

3.4.9. a destructor that deallocates the required dynamically allocated memory

3.4.10. getter member functions for the name, avatar, current position, and current log

3.4.11. a setter member function for the current position
   NOTE: Do not provide any additional getter or setter member functions; they are not necessary.

3.4.12. a `bool lessThan(Runner* r)` member function that compares two runners; a runner is considered "less than" another if its bib number is lesser

3.4.13. a `void update(Position& newPos)` function that does the following:
   (a) randomly select the runner's next move behaviour, as follows:
      (i) a new walking behaviour 40% of the time
      (ii) a new sleeping behaviour 10% of the time
      (iii) a new jumping behaviour 20% of the time
      (iv) a new sliding behaviour 20% of the time
      (v) a new teleporting behaviour 10% of the time
   (b) use the new behaviour object to compute a new position that will be stored in the `newPos` parameter
   (c) document the move in the runner's current log data member
      (i) for example, if the runner is named Timmy, and the new move behaviour is walking, the current log data member will store the string "Timmy walked one step"
      (ii) you can see examples of the contents of this string in the assignment video's sample execution, which is posted in *cuLearn*

      NOTE #1: In accordance with good design practice, this string is constructed by multiple objects! The move behaviour object constructs the portion of the string that describes the action, using a parameter to the `move()` function, and the runner's `update()` function adds the runner's name to the string. Every portion of this string is created by the object that owns the information.

      NOTE #2: We do not print the current log from this function. That will be done in a later step by the control object, using the view object, again following correct design principles.

3.5. Modify the `Array` class

You will modify the `Array` class that we implemented in the coding example of section 2.2, program #1, as follows:

3.5.1. modify the array to be a *dynamically allocated* array of `Runner` pointers
   (a) you can refer to the coding example in section 1.6, program #5, for examples of the four different kinds of arrays

3.5.2. modify the constructor and destructor accordingly; you can assume a fixed size for the capacity

3.5.3. modify the `add()` function as follows:
   (a) take a `Runner` pointer as parameter
   (b) add the given runner to the array in its correct place, in *ascending* (increasing) order
      (i) you must **shift** the elements in the array towards the back of the array to make room for the new element in its correct place; **do not** add to the end of the array and sort; **do not** use any sorting function or sorting algorithm
      (ii) you must use the `Runner` class `lessThan()` function to perform the comparison

3.5.4. implement a `Runner* get(int index)` function that returns the runner found at the index indicated in the parameter; this function must return null if the specified index is invalid

3.5.5. implement a getter member function for the size

3.5.6. remove the print function

3.6. Implement the `View` class

You will implement a new `View` class that contains the following data members:

3.6.1. the display, as a `char` double pointer
   (a) this will be a dynamically allocated 2-dimensional array of characters
   (b) it will be used to store the rows and columns of the race shown on the screen

3.6.2. the maximum number of rows in the display, set to 5

3.6.3. the maximum number of columns in the display, set to 25

The `View` class will contain the following member functions:

3.6.4. a constructor that allocates the 2D array required for the display, and initializes it to all spaces

3.6.5. a destructor that deallocates the necessary dynamically allocated memory

3.6.6. a `void print()` member function that prints the display to the screen

3.6.7. a `void update(Position& oldPos, Position& newPos, char avatar)` function that sets the display to a blank at position `oldPos`, and sets the display to the character `avatar` at position `newPos`

3.6.8. a `bool validPos(Position& pos)` function that returns true if position `pos` is within the bounds of the display, and false otherwise

3.6.9. a `void printStr(string str)` member function that prints the string `str` to the screen

3.7. Implement the `Race` class

You will implement a new class called `Race`, and an instance of this class will serve as the control object in this program. It's important to note that being a control object is a job, it's not necessarily the name of a class. In some programs, a control class will be called `Control`, but most often, it will have a name that's more descriptive within the program.

The `Race` class will contain the following data members:

3.7.1. the `View` object that will be responsible for all user I/O

3.7.2. the collection of all runners in the race, stored as an `Array` object

The `Race` class will contain the following member functions:

3.7.3. a constructor that does the following:
   (a) seed the random number generator with the command: `srand((unsigned)time(NULL ));`
   (b) dynamically allocate two runner objects: one that represents a tortoise, and another a hare
   (c) place both avatars on the display
   (d) add both runner objects to the runners collection

3.7.4. a `bool isOver(string& outcome)` member function determines whether or not the race has been won; it must traverse the runners collection to see if any of them has reached the right-hand side of the display; if so, the `outcome` parameter is set to the winning runner's name, and the function returns true; if not, the function returns false

3.7.5. a `void run()` member function that loops until the race has ended. At every iteration of this loop:
   (a) print the display to the screen
   (b) loop over the runners collection; for every runner:
      (i) compute a new position for the runner
      (ii) check that the new position is valid; if not, do nothing
      (iii) if the new position is valid:
         - set the runner's current position to the new one
         - update the display with the runner's new position
         - use the `View` object to print the runner's last move, as stored in their current log
   (c) once the race is over, print the display one last time, and declare the winner

NOTE #1: all error checking must be performed

NOTE #2: existing functions must be reused everywhere possible

3.8. **Write the `main()` function**

Your `main()` function must declare a `Race` object and call its `run()` function. The entire program control flow must be implemented in the `Race` object as described in the previous instruction, and the `main()` function must do nothing else.

3.9. **Test the program**

3.9.1. Run your program several times, making sure that the runner behaviour is randomized. All types of move behaviours must be tested, and they must work correctly.

3.9.2. Check that the display and runner logs are correct when they are printed.

3.9.3. Make sure that all dynamically allocated memory is explicitly deallocated when it is no longer used. Use `valgrind` to check for memory leaks.

4. Constraints

Your program must comply with all the rules of correct software engineering that we have learned during the lectures, including but not restricted to:

4.1. The code must be written in C++98, and it must compile and execute in the default course VM. It must not require the installation of libraries or packages or any software not already provided in the VM.

4.2. Your program must follow correct encapsulation principles, including the separation of control, view, entity, and collection object functionality.

4.3. Your program must not use any classes, containers, or algorithms from the C++ standard template library (STL), unless explicitly permitted in the instructions.

4.4. Your program must follow basic OO programming conventions, including the following:

4.4.1. Do not use any global variables or any global functions other than `main()`.

4.4.2. Do not use `struct`s. You must use classes instead.

4.4.3. Objects must always be passed by reference, never by value.

4.4.4. Except for simple getter functions, data must be returned using output parameters, and not using the return value.

4.4.5. Existing functions must be reused everywhere possible.

4.4.6. All basic error checking must be performed.

4.4.7. All dynamically allocated memory must be explicitly deallocated.

4.5. All classes must be thoroughly documented in every class definition, as indicated in the course material, section 1.3.

5. Submission

5.1. You will submit in *cuLearn*, before the due date and time, the following:

5.1.1. A UML class diagram (as a PDF file), drawn by you using a drawing package of your choice, that corresponds to the entire program design.

5.1.2. One `tar` or `zip` file that includes:

(a) all source and header files

(b) a Makefile

(c) a README file that includes:

(i) a preamble (program author, purpose, list of source and header files)

(ii) compilation and launching instructions

NOTE: Do not include object files, executables, hidden files, or duplicate files or directories in your submission.

5.2. Late submissions will be subject to the late penalty described in the course outline. No exceptions will be made, including for technical and/or connectivity issues. Do not wait until the last day to submit.

5.3. Only files uploaded into *cuLearn* will be graded. Submissions that contain incorrect, corrupt, or missing files will receive a grade of zero. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading

6.1. **Marking components:**

- 30 marks: correct UML class diagram
- 2 marks: correct implementation of `Position` class
- 12 marks: correct implementation of `MoveBehaviour` classes
- 20 marks: correct implementation of `Runner` class
- 8 marks: correct modifications to `Array` class
- 12 marks: correct implementation of `View` class
- 16 marks: correct implementation of `Race` class

6.2. **Execution requirements:**

6.2.1. all marking components must be called and execute successfully in order to earn marks

6.2.2. all data handled must be printed to the screen for marking components to earn marks

6.3. **Deductions:**

6.3.1. Packaging errors:

(a) 10 marks for missing Makefile

(b) 5 marks for missing README

(c) up to 10 marks for failure to correctly separate code into header and source files

(d) up to 10 marks for bad style or missing documentation

6.3.2. Major design and programming errors:

(a) 50% of a marking component that uses global variables or `struct`s

(b) 50% of a marking component that consistently fails to use correct design principles

(c) 100% of a marking component where unauthorized changes are made to provided code

(d) 100% of a marking component that uses prohibited library classes or functions

(e) 100% of a marking component where Constraints listed are not followed

(f) up to 10 marks for bad style

(g) up to 10 marks for memory leaks

6.3.3. Execution errors: 100% of any marking component that cannot be tested because it doesn't compile or execute in the course VM, or the feature is not used in the code, or data cannot be printed to the screen, or insufficient datafill is provided.