Inheritance and Polymorphism

This project will simulate the operation of Hookie Roads, a simple taxi service.

We will provide some design discussion, but the delegation of responsibilities is largely up to you. Note that does not mean that all designs are equally good, or that your designs decisions will not be evaluated.

A simulated clock drives most changes within the system; this simulation clock has no connection to the system clock. When the simulation clock "ticks" various objects currently in the system are notified and given the chance to update their state, according to the simulation rules given below. The order in which these notifications take place will determine the course of the simulation.

Hookie Roads employs a dispatcher who receives calls requesting that customers be picked up, and assigns those tasks to the various taxies that are available. Hookie Roads uses several types of taxies, described below. (That's where the inheritance/polymorphism comes into play.) The dispatcher may ask taxies if they are busy or available, and ask them if they can handle a specific task, but the dispatcher does not know which with type of taxi s/he is communicating.

The dispatcher has a list of available taxies, ordered according to when the vehicles were added to the Hookie Roads system. When the dispatcher receives a call, s/he stores that call in a FIFO queue. When the simulation clock "ticks", s/he assigns tasks to the available taxies in the order the calls were received. If an idle taxi cannot service a task, (due to party size) then the dispatcher must check to see if any other idle taxi can handle the task. For each task the Dispatcher must check all of the taxies (in the order they were added to the system) to see if they can handle a job. The dispatcher must then attempt to assign any remaining tasks, in a FIFO order, to still idle taxies. If there are not enough idle taxies to assign all of the tasks, the dispatcher saves the unassigned tasks for consideration on the next "tick". The dispatcher does not attempt to assign a task to the nearest available taxi. Tasks are assigned according to the taxi vehicle system ordering. There will never be more than 10 taxies.

The Hookie Roads Taxi Service (HRTS) operates in a very small community, consisting of a single street with addresses ranging from 1 to 99. The Hookie Roads home base is located at address 0 on the street.

For simplicity, we will not simulate any moving traffic other than the various taxies. The street will be initialized so that some addresses have a potential customer there and some addresses will have no customer. Some customers will need service and that will result in calls being made requesting rides. HRTS will only pick up customers for which a service request has been received. Those customers will be removed (in various ways) from the street; no other customers will change location. No additional customers will be added after system initialization.

Each customer will have a unique customer number, name, current address, destination address and party size, (i.e., the number of riders in their group). Each taxi will have a unique identifier, a location, and be pointing either up or down the street. (The “up” direction will be considered the direction of pointing/moving to a higher numbered address.) Specialized taxies may have additional members, as needed. When the simulation clock "ticks", each taxi will be notified (after the dispatcher considers assigning tasks). On a tick, the taxi must determine what to do:

If the taxi has no assigned task, it does nothing until the next tick.

When the dispatcher informs an idle taxi that it has a task for them, the taxi must determine if it can service the task, (i.e., handle the size of party).

If it has a task, and has not yet acquired the relevant customer, the taxi must determine whether it is at the right location to pick up the customer. If not, it must move toward the address at which customer was reported. All taxies move at a speed of one address per tick. Turning around is instantaneous. If the taxi is at the right location, it must verify that the right customer is there by comparing customer numbers. If so, it must pick up the customer. (Note: a customer’s location is the same as the taxi in which they are riding.) If not, an error message must be generated (somehow) and the taxi must abandon its assigned task, so it's available for another job. All of this is accomplished in a single tick. Since it takes a taxi a tick to stop and pick up a customer, it also takes a tick to stop and determine a customer is not at a pickup location.
If the taxi has a task, and has acquired the relevant customer, the taxi will deal with it in the manner determined by the type of the taxi. When a taxi has finished with a customer, it prepares for the next tick.

Note that there will never be a case where two or more taxies are competing to pick up the same customer; i.e., there are no "race conditions" in this simulation. The inclusion of such cases adds considerable complexity to a time-driven simulation.

Also note that it is entirely possible that the dispatcher will receive calls that specify non-existent customers. The dispatcher has no way to verify this, so such calls will eventually result in some taxi arriving at the specified address only to find that there is no customer there matching the given information. There may also be additional logical errors to consider. Any such errors must result in a logged error message detailing the problem.

The system will include a controller, a dispatcher who assigns tasks to taxies, a number of customers, and a number of taxies of various types. Your design of the system may very well include additional classes.

As usual, there will be a script file that provides commands to drive the simulation and you should implement a script manager/parser class that mediates between the controller and the script file. And, as usual, the script manager should not be involved in interpreting the commands from the script file. All file input should be handled by the script manager/parser on behalf of a controller.

**Controller:**

The controller will drive the simulation by interpreting the script commands and interacting as necessary with the objects that make up the simulation. The controller should only perform the minimal parsing necessary to determine the command and its parameters, and then invoke appropriate services of the other objects to actually carry out those commands.

**Taxi Hierarchy:**

HRTS uses three distinct types of taxies. They differ in how they deal with a customer. You should create a corresponding inheritance hierarchy, with an abstract base class at its root to capture the shared behavior of the subtypes.

**Caddy**

A caddy is basically just a standard taxi. It can carry one customer request at a time, servicing a party size up to five. It will always take its customer to their destination address and drop them off. Once a customer reaches their destination they leave the street and are no longer able to request other rides.

**Rambler**

A rambler picks up its unsuspecting customer and deposits them at a different location, but not necessarily at their destination. It can carry one customer request at a time, servicing a party size up to five. A rambler employs the following logic. If the customer is picked up at the address A, the customer will be carried to the address (A * A) % 100. If that address is vacant, the rambler will deposit the customer there. If that address is occupied, the rambler will then carry the customer to their desired destination and drop them off there. **A rambler must stop (for a tick) to allow a customer the chance to get off, even if they cannot do so.**

**Vanagon**

A vanagon is the big taxi for HRTS. It can handle up to two customer requests at a time, servicing a combined party size up to twelve. Once a vanagon has picked up a customer it will begin taking them to their destination. If the dispatcher gives it another customer service request it will stop and pick up that customer also, provided that their address is on the path to which it is taking the first customer. Once it has dropped off a customer, the vanagon is again available for another task. **The main behavior guideline for a Vanagon is that it will proceed to service the first customer assigned to it, before changing direction to service a second stored request. While servicing the first customer, if it has been assigned a second request the Vanagon should check to see if it can either pickup or drop off the second customer along the way. Two vanagon customers can be dropped off at the same destination during the same tick.**
Dispatcher:

The dispatcher will maintain a single list (or some other data structure) to organize the taxies. This list must store pointers of the base taxi type. Violation of either of these rules will result in massive deductions since that would avoid the fundamental polymorphism issues that are a major aspect of this assignment.

Other classes:

There are certainly other candidates for classes in this design. We do not guarantee that this specification has explicitly discussed all of the classes that a good design should contain. You will definitely not be penalized for having additional classes. (My planned solution will employ at least ten classes altogether.)

Initialization and Script Files:

On startup, the system will initialize the necessary objects and establish the static relationships that are employed. The first input file will contain customer data used to create the customers and their parties that will be on the street. This initialization file will consist of lines of the form:

```
<customer number><tab><name><tab><address><tab><destination><tab><party size>
```

Here's a sample:

```
1105 Wayne, J.  43 53 1
7459 Rogers, R.  37 25 2
2637 Eastwood, C.  42 50 7
9016 Fonda, H.  21 10 3
7432 Brennan, W.  46 55 5
```

The system must create a customer corresponding to each of the data lines, and "place" that customer at the specified address.

The script will contain comment lines and command lines. The comment lines will begin with a semicolon character. A command line will consist of a command string and one or more parameters, separated by tabs. Here's a sample:

```
; Test script for Hookie Roads Simulation
;
; Create a couple of taxies:
make caddy Caddy1 4 up
make rambler Rambler1 9 up
; Create some tasks to carry out:
call 2637  42  7
call 7432  46  4
; Tick, tick, tick...
tick  5
tick  5
tick  1
; A couple more tasks:
call 9017  21  6
call 9016  20  3
; More ticking...
tick  10
exit
```

This project is a good place to learn to include factory objects in a system design, so we will expect that whenever a taxi or other inheritance hierarchy object is created, it is obtained by calling an appropriate factory object, which will allocate the requested object dynamically and return a base type pointer to the new object.
Each non-comment line of the script file will specify one of the commands described below. Each line consists of a sequence of “tokens” which will be separated by single tab characters. **Bold** text indicates commands or keywords that will be used verbatim. Tokens will never contain a tab character. A newline character will follow the final “token” on each line. Any spurious data following the last parsed token for a command must be ignored.

**make**

```
caddy | rambler | vanagon]   <ID>   <address>   [up | down]
```

Create a taxi of the specified type, with the given ID and direction, at the given initial address, and add it to the dispatcher's list. QTP: in a good design, who should be responsible for the actual creation?

**call**

```
<number>   <address>   <size>
```

Notify the dispatcher that a call has been received requesting that a customer with the specified number and party size has requested service from the specified address.

**tick**

```
<nticks>
```

This causes the simulation clock to tick the specified number of times. On each tick of the simulation clock, various objects in the system are notified of a tick, as described above.

**loiter**

```
<address>
```

Report which customer, if any, is currently at the specified address. **The loiter command will only report on customers who are on the street, (i.e., it will not report on any taxi passengers).**

**locate**

```
<number>
```

Report the status of the customer, if any, which has the specified number. This will either be a street address or the identifier of the taxi which is carrying the customer.

**exit**

This causes your program to deallocate all dynamic memory and then terminate immediately. The script file is guaranteed to contain an **exit** command.

**Legend:** in the commands above:

- **<ID>** a string which is a unique identifier for a taxi or customer
- **<number>**
- **<address>** a non-negative integer representing a street address
- **<size>**
- **<nticks>** a positive integer representing the number of ticks to be simulated

Each command must be logged, along with an informative message indicating the results when the command was processed. Note that every command should produce some informative output. Here are some specific data logging requirements:

- **On a make command,** log the creation of the object, specifying its type, identifier and location, and also a sequence number indicating its position in the dispatcher's list.
- **On a tick command** (each individual tick):
  - log the tick counter (start counting ticks at 1, not 0)
  - log every assignment of a task to a taxi
  - log the state of each taxi after it has performed its update in response to its tick notification; this should include the taxi identifier and location, a description of its assigned task, if any, and a description of its current customer, if any.

Feel free to log additional information if you like; just keep it well formatted and readable.

**Simulation Log:**
The simulation log will be written to the provided command line parameter naming an output text stream. The log should begin with the name of the programmer and the names of the input files involved in the current execution.

You should follow this with a display of the initial "street", indicating what customers are located at what addresses.

After that, an entry must be logged for each scripted command. Each command must be echoed, along with a sequence number as before. Following each echoed command, log relevant output.

The exact contents and formatting are up to you, but the logged messages must be precise and informative. The output relating to each command must be formatted so that it is easy to distinguish one command from another.

Additional sample initialization and script and corresponding simulation logs will be posted on the course website.

**System Design:**

Much of the evaluation of your implementation will depend on the quality of your design. All the relevant design guidelines given for the previous project still apply.

Given the stated pedagogical goals of this assignment, we have some specific restrictions:

- All taxi types must exist in a common inheritance hierarchy.
- For all inheritance hierarchies, the top-level base class must be abstract.
- Be careful when you factor the commonality of the taxi types into their base class(es), and likewise for the any other inheritance hierarchy.
- All accesses to derived type taxies and other inheritance hierarchy objects must be via base type pointers (for their respective hierarchies).
- Each taxi and other inheritance hierarchy types must have a destructor, which has been instrumented to log the destruction of the object. The destructor must log the identifier of the particular object being destroyed. This information is necessary in order to determine whether you are properly deallocating memory when objects are destroyed; failure to include this log information will be taken as *prima facie* evidence that your implementation does not deallocate objects correctly, and that it would crash if it attempted to do so. That will, of course, result in large deductions.
- The factory objects are allowed to use pointers of derived types internally, but derived type pointers must not occur elsewhere in your implementation.
- No object is allowed to have a data member that stores its type, or to provide a member function that could be used by client code to retrieve the object's type. The use of RTTI is forbidden. Violations will be penalized massively.
- Testing your implementation is much easier if you modify the processing of a tick command to automatically dump the street state, and possibly the taxies, after each tick. This is optional and will not be required, or penalized, at the project demos.

**Use of STL Components**

You are required to use the STL queue when implementing the task list management within the dispatcher. You may make use of any other STL components you wish, elsewhere in your implementation. You may also implement your own templates for this project if you like.

**Program execution:**

The name of your program executable is up to you... for this discussion we assume it is hrtssim.exe. Your program must accept three command-line parameters when it is executed:

```
hrtssim <init file name> <script file name> <log file name>
```
The file names will simply be strings. The first two files must already exist when the program is executed. You should check for the existence of the initialization and script files on program startup and issue appropriate error messages if either is missing.

**Programming Standards:**

The GTAs will be carefully evaluating your source code on this assignment for programming style, so you should observe good practice. See the Programming Standards page on the course website for specific requirements that should be observed in this course.

**Evaluation:**

You will schedule a demo with your assigned TA. The procedure for scheduling your demo will be announced later. At the demo, you will perform a build, and run your program on the test data, which we will provide to the TAs. The TA will evaluate the correctness of your results. The TA will evaluate your project for internal documentation and software engineering practice.

Note that the evaluation of your project will depend substantially on the quality of your code and documentation.

**Submitting your program:**

You will submit this assignment to the Curator System (read the Student Guide), but it will not be graded automatically. Instructions for submitting are contained in the Student Guide. Do not submit unnecessary files. See the course website for detailed instructions.

You will be allowed up to five submissions for this assignment. Test your program thoroughly before submitting it. Make sure that your program produces correct results for every sample data set posted on the course website. You should also construct your own test data.

**Pledge:**

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you must include the following pledge statement in the header comment for your program:

```c++
// On my honor:
// - I have not discussed the C++ language code in my program with anyone other than my instructor or the teaching assistants assigned to this course.
// - I have not used C++ language code obtained from another student, or any other unauthorized source, either modified or unmodified.
// - If any C++ language code or documentation used in my program was obtained from another source, such as a text book or course notes, that has been clearly noted with a proper citation in the comments of my program.
// - I have not designed this program in such a way as to defeat or interfere with the normal operation of the Curator System.
// - I have neither given nor received unauthorized aid in the completion of this assignment.
// <Student Name>  <Student PID>
```

Failure to include this pledge in a submission is a violation of the VT Honor Code.