Polymorphic Ecosystem Simulation Testbed

For this project you will implement a simulation of a very simple ecosystem. Creatures of several types, specified below, will inhabit the system. The number of creatures in the system will vary. Most changes within the simulation are driven by a simulated clock; this simulation clock has no connection to the system clock. When the simulation clock "ticks" each creature currently in the ecosystem is notified and given the chance to update its state, according to the behavioral rules given below. The order in which the creatures update their state will determine how some subtle interactions play out. For consistency, we adopt the following rules. All plants are notified that a tick has occurred before any insects are notified. Plants are notified in the order they were created, and so are insects.

The ecosystem is envisioned as an infinite grid of spaces, rather like a chessboard with an x-y coordinate system imposed on the squares. The program will read a script file and carry out the commands it contains, writing all logged output to standard out (cout). The name of the script file will be specified as a command-line argument to the program.

Program Invocation:

Your program must take the name of the script file from the command line — failure to do this will irritate the person for whom you will demo your project. Typically, the program will be invoked, using output redirection, as:

```
EcoSim [-v] <script> > <log>
```

If the specified script file does not exist, the program should print an appropriate error message and either exit or prompt the user for a correction. Optionally, the first parameter may be a flag "-v"; if the flag is used then the program will write additional diagnostic output to aid in debugging errors. It is up to you to decide what information to display, but you should take this seriously as an opportunity to provide structured, verbose feedback intended to aid anyone analyzing the behavior of your program.

Creatures:

The ecosystem may contain creatures of several types, distinguished by behavior. For convenience, each creature will have a unique identifier, a name if you will. Each creature will have an energy level, represented by a nonnegative integer, and a location, represented by a pair of coordinates. Some types of creatures can move, and moving costs energy. Simply standing still also costs energy for some creatures, whereas others actually gain energy simply by existing. Some types of creatures eat other creatures under certain circumstances, acquiring some or all of the energy of the eaten creature. An eaten creature loses energy, sometimes some, sometimes all of its energy. If any creature's energy level drops to zero, it dies.

There are four kinds of creatures in the ecosystem. The first two are vaguely insect-like:

**sipper**
A mildly hungry insect. If it is not feeding, a sipper moves one space on each tick of the simulation clock; a sipper follows the movement pattern NE-SE-S-W-W, repeating the pattern indefinitely. Moving from one square to an adjacent square costs a sipper 2 units of energy.

If a sipper enters a square containing a plant, it will interact with the plant. If the plant is a fruit, the sipper will (on the next tick) begin consuming energy from the fruit. A feeding sipper will consume 5 units of energy per tick, or all the fruit's energy if it has less than 5 units, until the sipper has raised its energy level to 25 or the fruit's energy level reaches zero. At that point, the sipper stops feeding and will resume movement on the next tick, continuing its pattern from the point it stopped.

**muncher**
A marauder. If it is not feeding, on each tick a muncher is two spaces forward, then one space to its left, and then turns 90° counter-clockwise; munchers are always created facing east. Moving from one square to an adjacent square costs a muncher 2 units of energy; turning costs no energy.
If a muncher enters a square containing a plant, it will interact with that plant, interrupting its current move sequence if necessary. If the plant is a fruit, the muncher will consume 12 units of energy, or the fruit's total energy, whichever is less. The muncher will then cease feeding, and resume moving on the next tick. The muncher continues its movement pattern from the point it was interrupted.

A muncher does not limit the total amount of energy it may consume, but if its energy total reaches 50 it will divide into two new munchers, one with 30 units of energy and one with the remainder, which will be at least 20. By convention these will be given IDs obtained by concatenating B and S, respectively, to the ID of their "parent". If the fruit still has energy, the offspring will attack it on the next tick.

For both sippers and munchers, simply existing for one tick costs one unit of energy. The other two types of creatures are essentially plantlike:

**fruit**

An edible plant. Fruits eat any other type of creature. However, unlike all other types of creature, these gain one unit of energy per tick simply by existing. There is no upper limit on the amount of energy a fruit may store.

**flytrap**

A carnivorous plant. Existing costs a flytrap one unit of energy per tick. If a sipper or muncher is unlucky enough to interact with a flytrap, the flytrap will eat that creature, consuming all its energy. This happens immediately, as soon as the creature enters the flytrap's square. There is no upper limit on the amount of energy a flytrap may store.

Plants never move from the square in which they were created.

A square will never contain more than a single plant, mainly to simplify the simulation rules. However, a square may contain an arbitrary number of insectoids.

As a general rule, an insectoid will determine if there is an available plant when the insectoid first enters the square. So, if an insectoid wanders into a square holding a flytrap it will be eaten immediately. We will avoid the scenario in which a new creature is created in a square already containing another creature.

**Death:** creatures of all types can die. For some types, there are multiple ways to achieve this. For others, there is only one. Some creatures may run out of energy (or suffer other fates), and die, during their own update operation. Some creatures may be killed by another creature during that creature's update operation. In all cases, the simulation should result in the dead creature being removed as soon after its death as possible. Slight penalties may be assessed for delayed removal; larger ones will be assessed if a dead creature is not removed at all.

The creatures in the ecosystem must be organized using some sort of dynamic list structure. You are required to have two lists: one for all the plants and one for all the insectoids. Since the creatures will be created dynamically, and the use of polymorphism is an essential goal of this assignment, each of these lists will store base-class pointers. It is NOT acceptable to have separate lists for each type of creature. Since doing so would eliminate a fundamental aspect of this project, students who take that approach will be assigned a final project score no higher than 30 points.

**Script File Description:**

For a change, lines beginning with the character (‘#’) are comments; your program will ignore comments. An arbitrary number of comment lines may occur in the script file.

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. **Command words will always be followed by a single tab character.** A newline character will immediately follow the final “token” on each line.
create [sipper | muncher | fruit | flytrap] <name> <energy> <x> <y>
Create a creature of the specified type, with the given name and initial energy, at the given coordinates, and add it to the (appropriate) list.

tick <nticks>
This causes the simulation clock to tick the specified number of times. On each tick of the simulation clock, each creature in the ecosystem is notified of a tick. Plants should be updated first, then other creatures. Within each of those categories (plant and other), the notifications should be done in the order the creatures were created.

status <name>
Logs the current status of the named creature (if it exists). The status report should include the creature's type, its name, its energy level, and its current location.

exit
This causes your program to log a status display for every existing creature, and then deallocate all dynamic memory and terminate. The script file is guaranteed to contain an exit command.

Legend: in the commands above:

- <name> a string which is a unique identifier for a creature
- <energy> a positive integer representing the initial energy of a creature
- <x> an integer representing the initial x-coordinate of a creature
- <y> an integer representing the initial y-coordinate of a creature
- <nticks> a positive integer representing the number of ticks to be simulated

Each command must be logged to cout, along with an informative message indicating the results when the command was processed. Note that every command should produce some informative output. The output from each command should be delimited in some manner, similar to the parsing homework assignment.

You may assume that the script file will conform to the given syntax, so syntactic error checking is not required. However, it is certainly likely that search commands may specify non-existent creatures and your program must deal with that gracefully. If an error occurs during the parsing of the script file, there's an error in your code. However, your program should still attempt to recover, by “flushing” the current command and proceeding to the next input line. Here is a very simple script file:

```
# Eco System Simulation Script
#
# Populate the grid:
create fruit Juicy 30 4 7
create fruit Sour 15 3 2
create fruit Tart 40 5 8
create flytrap Charles 18 3 3
create sipper Sue 20 2 2
create sipper Jeff 25 2 7
create muncher Dave 25 2 5
#
# tick 5
#
status Sue
status Jeff
status Dave
status Juicy
status Charles
#
tick 5
#
exit
```
Design:

The script file should be handled as in the previous projects, via some sort of file manager class. The interpretation of commands should be handled by a controller class. There should be a factory class that dynamically instantiates creatures of various types, when requested by the controller. (It is acceptable for a dividing muncher to handle creating its offspring directly, rather than asking the factory to handle that.) The factory class should return a base class pointer of the appropriate type to the controller. (That way the controller does not retain any specific type information for the creatures it receives, so it's easier to guarantee true polymorphism.)

There should be a class for each type of creature. The list data structure(s) that organize the creatures should be instances of some class. If you do not make that a template, you will have to implement two different containers. (So, you should make it a template.) Since all output is to cout, the management of output can be local. Each class is allowed to perform output; you are encouraged to organize this elegantly and with an eye toward later modification.

For this project, we will discuss some required and some recommended classes in the project specification. You are expected to conform to these guidelines. We will not, however specify all details of the relationships among these objects.

The determination of inheritance relationships among your classes, and of reasonable attributes and responsibilities for each class is largely left to you. That does not mean that all decisions are equally good, nor that all decisions will receive full credit. Your goals should be to place responsibilities where they most logically belong, and to provide for maximum cohesion and reusability. Inheritance should be used where it makes sense to do so. It is certainly permissible to have additional classes not mentioned in this specification, including classes that help to organize the inheritance hierarchy.

In particular, it is useful to have a class to represent the ecosystem itself, serving as an intermediary between the controller, which interprets commands, and the creatures that inhabit the ecosystem. This is NOT a required element of your design, but you may find it actually simplifies some of the issues. We may choose to address the specific implications of these goals for this project in class.

For the creature list(s) you may implement your own container, as a template, or you may use one of the container templates from the Standard Template Library. In particular, you may wish to consider using the queue template declared in <queue> and/or the list template declared in <list>.

One of the main points of this project is to achieve true polymorphism in the use of the various creature objects. Because it is sometimes necessary to distinguish plants from insectoids, it is required to use two creature lists to separate plants from insectoids. It is not, however, even remotely acceptable to have separate lists for sippers and munchers, or for fruits and flytraps. The controller class, or whoever manages the creature lists, must be unaware of the precise type of each creature object. There are many type-specific behaviors; in all cases, the controller should not be telling any creature object precisely what to do (e.g., "move north" or "attack sipper"). The controller must only give completely generic instructions, such as "clock has ticked". To guarantee that, we have some restrictions:

- There must be two disjoint inheritance hierarchies, one for insectoids and one for plants.
- Each inheritance hierarchy must have an abstract class at its ultimate base.
- Each creature lists must store a pointer of the (abstract) base type from the appropriate inheritance hierarchy. This must not be of any actual creature type.
- Creature objects may not store their type as an explicit data member. They are allowed to display their type to an output stream, but that must not be intercepted by the controller (or any other class) and parsed in order to cheat and determine the actual type of a creature.
- The use of RTTI (runtime type information), or typeid is explicitly forbidden.

Also, the controller should not decide when a creature dies; rather, the controller (or ecosystem) should be informed when a creature dies. Who informs the controller a creature has died? Perhaps the creature itself, in some cases; perhaps its killer in other cases. That's not really too unnatural. That communication issue is a bit delicate since a dead creature must be removed from the creature list and deallocated.
Data structures are an issue in two places. For the creature list(s) you may use an appropriate STL container template. Insectoids must also store a list of movements and keep track of which movement to perform next. These should not be separate, unorganized data members. You may also use an STL container for this purpose.

The project does not involve only inheritance. There will be aggregation relationships, and there will be association relationships. One bears a comment. There must be some association between the controller or ecosystem object and the various creatures, perhaps via container objects that organize the pointers to the creatures. The creature objects may also need to ask questions of the controller or ecosystem object. In that case there will be a bi-directional association; i.e., each class must contain a reference to the other class. In order to accomplish that in C++ you will almost certainly have to make use of one or more forward declarations, as in the first project.

**Implementation Plan:**

Since this project is the last, we will be covering some of the necessary concepts as you are working on your design and implementation. What follows is a description of one way to approach the implementation phase so that you have the best chance of producing a solution that is either complete, or at least that correctly handles what it handles and successfully avoids being broken by the things it does not handle.

Obviously the place to start is with the parsing of the input and the recognition of commands. Most of that should be recycled from the earlier projects, with relatively minor changes to reflect the specific set of commands for this project. It helps to design the interpretation code so that commands that are unrecognized have no effect on the simulation; in other words, for commands you haven't yet implemented handlers for, just have a selection clause that does nothing other than log the command and a message indicating it isn't handled yet.

The next step would be to implement any abstract base classes your design calls for, and to derive one of the creature classes from it. I suggest starting with the plant hierarchy and the fruit class since that's the simpler case. Implement the container logic and verify that you can create and store plants, and update their state correctly as the simulation clock ticks.

Next, implement an insectoid base class and one of the specific insectoids. Again, verify that you can create and store creatures of this type, and update their location correctly as the simulation clock ticks. Implement and test whether the insectoid interacts correctly with plants.

Next, implement the remaining insectoid class, and test whether it interacts correctly with the plant types you've already created. Then you're ready to implement the remaining insectoid type and test its interactions.

By adding creature types one by one you isolate testing and debugging. So, at each stage you should have an implementation that correctly handles the behavior and interactions of the creatures you've gotten to, and essentially ignores any commands relating to the creature types you haven't gotten to. Ideally you will get everything implemented. If not, this approach is at least likely to produce a solution you can demo to your best advantage. (In other words, it's usually better to have an implementation that only handles a subset of the requirements, but handles those perfectly, than to have an implementation that attempts all of the requirements but handles none of them perfectly and handles only a few of them well.)

As you add each creature type, test its movement, energy updating and interactions with the other creature types that were added before it. That will isolate the specific issues each type raises.

**Log Description:**

Since this assignment will be graded by TAs, rather than the Curator, the format of the output is left up to you. Of course, your output should be clear, concise, well labeled, and correct. The first few lines should contain your name, course (CS 2704), and project title.
The remainder of the logged output should come directly from your processing of the script file. As a general rule, each creature should report what it is doing during its update cycle. That should include some information regarding any movement, feeding actions, etc.

As usual, you are required to log each command that you process, along with a command counter, and to delimit the output from each command. When processing a tick command, you should also delimit any output on the individual ticks. Number ticks starting at 1. Sample log output that corresponds to the script file given earlier will soon be available on the course website.

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:

Shortly before the due date for the project, we will announce which TA will be grading your project. 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. In addition, the TA will evaluate your project for good 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 a zipped file containing your project to the Curator System (read the Student Guide), and it will be archived until you demo it for one of the GTAs. Instructions for submitting are contained in the Student Guide. You will find a generic list of the required contents for the zipped file on the course website. Follow the instructions there carefully; it is very common for students to suffer a loss of points (often major) because they failed to include the specified items. Do not submit unnecessary files. See the course website for detailed instructions.

Be very careful to include all the necessary source code files. It is amazingly common for students to omit required header or cpp files. In such a case, it is obviously impossible to perform a test of the submitted program unless the student is allowed to supply the missing files. When that happens, to be fair to other students, we must assess the late penalty that would apply at the time of the demo.

You must also include an MS Word doc containing a revised inheritance diagram, and a complete class diagram and class forms, reflecting your final design.

You will be allowed up to five submissions for this assignment, in case you need to correct mistakes. Test your program thoroughly before submitting it. If you discover an error you may fix it and make another submission. Your last submission will be graded, so fixing an error after the due date will result in a late penalty.

The submission client can be found at: http://eags.cs.vt.edu:8080/curator/

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you must include the pledge statement from the course website in the header comment for your main source code file.