

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 ecosystem is envisioned as an infinite grid of spaces, rather like a chessboard with an x-y coordinate system imposed on the squares. Each space can hold an arbitrary number of creatures, subject to the behavioral rules.

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 <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.

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. Most types of creatures can move, and moving costs energy. Simply standing still also costs energy for most creatures. Most 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, usually all of its energy.** If a creature's energy level drops to zero, it also dies. The following types of creatures must be supported:

bush A plant. These don't move, nor do they eat any other type of creature. However, unlike all other types of creature, these gain energy simply by existing. A bush gains one unit of energy for every tick of the simulation clock.

grazer A seriously hungry herbivore. A grazer moves one space on each tick of the simulation clock, following the pattern:

east, south, ... (repeated indefinitely)

Simply existing for one tick of the clock costs a grazer one unit of energy. Moving from one space to an adjacent space costs a grazer one unit of energy. (This is assessed before the grazer has a chance to nibble anything in the square.) If a grazer enters a space containing a bush, the grazer will eat the bush, consuming all of the bush's energy.

browser A mildly hungry herbivore. A browser moves one space on each tick of the simulation clock, following the same pattern as a grazer, with the same energy costs. If a browser enters a space containing a bush, and the bush has at least **20** units of energy, the browser will eat **2** units of the bush's energy, which may or may not kill the bush. If the bush has less than **20** units of energy, the browser will ignore the bush.

grendel A pure carnivore. A grendel moves according to the following pattern:

east, east, no move, **north**, no move, west, ... (repeated indefinitely)

For a grendel, existing for one tick of the simulation clock costs one unit of energy and moving from one square to an adjacent square costs two units of energy. (These costs are assessed before the grendel has a chance to nibble anything in the square it just entered.) If a grendel enters a square containing a grazer or a browser, the grendel eats the herbivore, consuming up to 30 units of its energy. If a grendel enters a square containing another grendel, the new arrival eats the other one (advantage of surprise), consuming up to 30 units of its energy. **In each case, the grendel's victim will die, even if the grendel did not consume all of its energy.**

As a general rule, a predatory creature will only decide to consume another creature when the predatory creature first enters the square. So, if an herbivore wanders into a square already holding a grendel, no harm is done (too dumb to be good food, if you like). Similarly, if a creature is created in a square containing another creature, neither will attack the other unless one (or both) leave that square and they subsequently converge.

For simplicity, we will guarantee that no square will ever hold more than one plant at the same time, nor will a square ever hold more than two animals at the same time. Therefore, you don't have to worry about issues like which available herbivore would a grendel attack if it entered a square containing five herbivores. Of course, you're free to design protocols to handle more complicated scenarios, but our test data will avoid them.

The creatures in the ecosystem must be organized using some sort of dynamic list structure. It is acceptable to have two lists: one for plants and one for all other creatures. 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. A newline character will immediately follow the final "token" on each line.

create [**bush** | **browser** | **grazer** | **grendel**] <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 deallocate all dynamic memory and then terminate immediately. The current in-memory databases are not automatically saved to disk. 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 non-negative 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
#
create browser Thumper 30 4 7
create grazer Bessie 15 3 2
create bush Vannevar 3 5 8
create bush George 1 3 3
create grendel Charles 20 2 2
create grendel Vlad 25 2 7
#
tick 5
#
status Charles
status Thumper
status Vannevar
#
tick 5
#
exit
```

Check the website for additional samples.

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 manager class. 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. 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 manager, 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 non-plants, it is acceptable to use two creature lists to separate plants from non-plants. It is not, however, even remotely acceptable to have separate lists for herbivores and grendels, much less a separate list for each creature type. The manager class, or whoever manages the creature lists, must be unaware of the precise type of each creature object that's not a plant. To guarantee that, we have some restrictions:

- The creature inheritance hierarchy must have an abstract class at its ultimate base.
- The creature list must store a pointer of the (abstract) base type from the creature 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 manager (or any other class) and parsed in order to cheat and determine the actual type of a creature.

Also, the manager should not decide when a creature dies; rather, the manager should be informed when a creature dies. Who informs the manager 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 should be removed from the creature list and deallocated. (If you missed it, the comments above on the creature lists imply that the creatures must be allocated dynamically.)

Data structures are an issue in two places. For the creature list(s) you may use an STL container template. (That's what I did.) Most creatures must also store a list of movements and keep track of which movement to perform next. These should not be separate, unorganized data members. I also used 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 manager 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 manager 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. We will discuss an example of this in class.

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 second project, 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 a plant since it'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 animal class; which one you do first may depend upon your inheritance design, but I'll assume it will be an herbivore. 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 herbivore interacts correctly with plants.

Next, implement the predator class (grendel), and test whether it interacts correctly with plants and herbivores. Then you're ready to see how grendels interact with each other.

From this point it should be a relatively simple matter to add new creature types to the existing implementation. The specification only calls for a few, so you're almost done adding code. 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. You are required to log each command that you process, **along with a command counter**, so that it's easy to determine which command each section of your output corresponds to. You are also required to delimit the output from each command. Sample log output that corresponds to the script file given earlier is available on the course website.

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.

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://spasm.cs.vt.edu:8080/curator/>

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 GTA. The procedure for scheduling your demo will be the same as for the earlier projects, although the partitioning of students among GTAs will be changed. At the demo, you will download your submitted project, perform a build, and run your program on the supplied test data. The GTA will evaluate the correctness of your results. In addition, the GTA will evaluate your project for good internal documentation and software engineering practice.

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 main source code file.

```
// 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.
//
// <Student Name>
```

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