Section 4 Lesson 5

Section 4 lesson 5 – Ecology

I Discuss new data structures, algorithms, or language features

Ecology covers a lot of ground with links, object orientation, inheritance, polymorphism, a custom memory management system, and a great demonstration of using the cross over and point mutation genetic algorithms. It is truly an object oriented program, requiring most of the features of C++.

II Describe functions and modules

Ecology is built from sixteen separate files. When menus are added, in the near future, there will be four more.

Gene.cpp & Gene.h create a Gene class with the trivial constructor, destructor, and another constructor which makes a new gene object from a gene sequence of m_length. There are methods for getting and setting the gene sequences, as well as three methods to display them. The mutation() method, being unary, is contained within the class structure while the two dyadic operators, crossover() and inversion(), are external to it. The Gene class has only two private variables: m_sequence and m_length. The unsigned integer used for the gene sequence (in binary) is 32 bits long. Here I have a dilemma in terminology. What I started off calling genes are better described as chromosomes, since I use parts of them to express genetic characteristics. Thus, each 32 bit chromosome would have two, three, or even four genes described by the string. I have just made this into an assignment for the eager, and most studious of you.

Gene.cpp defines three display functions, the mutation function, and adds the two helper functions crossover() and inversion(). displayGene(void) displays the string sequence of a gene. displayGene(Gene g1) displays any gene it is passed. The third display is used for showing any raw sequence it is passed. It was used for debugging the following genetic operators. mutation() is a genetic operator which induces a point mutation in the 32 bit gene. crossover() takes the first part of the first gene and the second part from the second gene. This genetic operator chooses a point in the first chromosome, and copies everything after that from the matching positions in the second chromosome. inversion() is the third genetic operator, which I use very rarely. It too chooses a random location from the 32 bits of the first chromosome. It shifts the sequence first to the right by that many positions and then back again the same number of times. What this does is zero out everything to the right of the randomly chosen position. The opposite is done for the second chromosome using the very same index. So you have the first part of the first chromosome, followed by all zeros and the second chromosome, with all zeros until you reach the chosen point. You now flip the second chromosome end for end (at least the part with information in it) and put that string into the first chromosome.

Organism.cpp & Organism.h hold the definitions and variables for the next level of the inheritance tree. Organisms have chromosomes and express the genes on them during their lives. Organism.h creates an Organism class with the normal constructors, a destructor, along with the gets, and sets. Organisms also have a few attributes – an x, y position, an energy level, an age, plus a metabolism chromosome. Since Organism is a base class, you need virtual functions to prepare for the future: move() and display(). Since they are virtual functions, they need to be fully described later in the inheritance tree. There are three types of organisms – plants, herbivores, and carnivores. Plants don't move so you don't need to define the move() virtual function for the class. But, herbivores and carnivores do move, so they need this method to be defined. This technique allows you to give one command <move> to either a herbivore or a carnivore and each expresses the function differently. We also have a method to allow an organism to die. If the energy level drops to zero an organism is dead, the object is deleted from the living lists, added to the pool lists, and its nutrients move to the nutrient layer. Organism.cpp defines a constructor and a method. The constructor creates a 32 bit random number which represents the metabolic chromosome. This chromosome has four genes on it – energy used, energy derived from food, energy value to next trophic level, and the age of sexual maturity for a total of 12 bits. This could be expanded in the future. You could add more chromosomes at the organism level. The method defined is used by the object, to extract the gene of choice from the metabolic chromosome.

Plant.cpp & Plant.h declare and define the Plant class, which inherits from the Organism class, while adding the Plant chromosome. The constructor creates a new Plant object at the input x,y location, sets the energy level to 10 and the age to 1. It also fills 9 bits of the plant chromosome with randomly generated bits representing pollination, poison probability, and spines. These chromosome have yet to be expressed in the simulation. They are all in my notes but where is the time?

Remember that virtual function we inherited from Organism? Well it gets defined here. display() prints both the metabolic and the plant gene sequences. We also have getGene() and setGene() to use, or modify, m_plant. Expressing these genes is an assignment for you eager beavers. Notice lines 34 and 35. The Plant class has the pointers to become a node of a linked list. Plant.cpp defines only one more method, getGene(int) which extracts the gene of choice from our plant chromosome.

Animal.cpp & Animal.h declare and define the Animal class which also inherits from Organism. Two constructors but a virtual destructor. This is because Animal is a base class for both the Herbivore and Carnivore classes. By making the destructor virtual here, the responsibility to destruct the object gets passed to the inheriting class. That's because ~Carnivore() and ~Herbivore() are different, but both call the destructor ~Animal(). The ultimate destructor code is written in the inheriting class.

Animal adds another chromosome for movement and a direction probability array. The chromosome and the array are used together, to provide the animal with its movement behavior. Animal.cpp defines two methods and the constructor. Animal::move() generates a random number on line 8, then uses it to pick a probability weighted direction. It gets its location, gets the speed gene, then calculates the new location.

The constructor takes a location then creates an Animal there, with a set energy level and an age of one. The movement chromosome is built of 9 genes. Eight direction probabilities of 3 bits each, and another 3 bits for the speed. Animal::getGene(int) extracts the chosen gene from the movement chromosome.

Herbivore.cpp & Herbivore.h declare and define the Herbivore class which inherits from Animal. The trivial constructor and destructor are included, as well as a constructor which creates new Herbivore objects, then sets 20 bits of the m_tactics chromosome, with five genes of four bits each. The display() method overwrites the underlying, virtual method by displaying all three of its chromosomes: metabolic, movement, and tactics. A few get and set routines for the tactics chromosome, as well as two pointers for the Herbivore linked list conclude Herbivore.h.

In Herbivore.cpp the Herbivore and Organism list heads are declared external, to keep the lists in order. Check this against Carnivore.cpp, and test if it is absolutely necessary. Herbivore::getGene(int) extracts the chosen gene from the tactics chromosome.

Carnivore.cpp & Carnivore.h declare and define the Carnivore class which inherits from Animal. One carnivore constructor creates the new Carnivore object at x, y, sets a few other values, and creates the 24 bit tactics chromosome from six 4 bit genes. Once again, the virtual display method is overwritten to show the metabolic, movement, and tactics chromosomes. hunt() is currently not defined. It would be used to express the tactics chromosome's various genes as hunting behaviors. I have notes for implementing all of them. Eager students can ask for a copy. Then the required get and set methods, to manipulate the tactics chromosome and pointers so any Carnivore object can be on its own linked list. In Carnivore.cpp there are no external declarations. The getGene(int) method extracts the chosen gene from the tactics chromosome.

World.cpp & World.h includes each of the Organism, Plant, Herbivore, and Carnivore header files. The World class is where the four linked lists fit together. Methods to add and delete Plants, Herbivores, and Carnivores are included as well as some data collection variables.

In World.cpp the add and delete methods are each defined. These are modified from older linked list libraries I've created. It is easiest to keep each species in its own list, since the ecosystem needs to keep track of all of the populations of plants, herbivores, and carnivores. There is an organism list which does keep track of all of them too, so there are really two lists threaded through each species' population. Keeping track of all of the pointers is a major test of programming (and debugging) skills. Once one function was done it could be used as a template to create the next and the next function to minimize anything other than cut and paste errors.

Ecosystem.cpp & Ecosystem.h are where the main functions are put together into an application. It is where WinMain() lives too. First we have our #includes, #defines, and function declarations in Ecosystem.h, then get to work creating the application in Ecosystem.cpp, where we begin with a few more #defines and function declarations. They should get moved to Ecosystem.h But these are also hooks where a menu/dialog box system could manage parameter settings. Planning ahead for menus is a good thing. I have the menu system in my files somewhere, now to implement it in this revision of Pond (the original name of the application in 1998).

WinMain() creates the display window with menu capability. None are attached yet. Randomize the random number generator by using the system clock, then call your initialization routine. Display and refresh the screen, attach the simulation thread, start it and the message loop. At present, the user interface thread (WndProc) is limited to initialization (WM_CREATE), refreshing the display (WM_PAINT), and getting rid of the evidence when the app closes (WM_DESTROY). Here is where the menu structure will fit, for parameter control.

The function scanOrgs() is the wrapper around the simulation. It calls the eating, moving, death, and reproduction routines. It does so on an Organism by organism basis; each object knows whether it is a Plant, a Herbivore, or a Carnivore and does the proper thing, each in its own way. It starts by adding more nutrients every 100 generations. Then each organism eats what it can, according to its genes. Each organism metabolizes (ages and uses energy) then moves if it can. If the organism uses all of its energy it dies. Of those remaining, a reproduction cycle occurs if they are old enough, and have enough energy. Each plant, herbivore, or carnivore which meets those criteria, finds a mate and exchanges genetic material. The simulation ends when any of the three species population levels drops to zero.

replenish() is used to add more nutrients to the world, in various patterns; rich areas and resource poor areas are created. More work needs to be done monitoring nutrient levels of the system. I think it is resource rich at the moment.

eat() scans the Plant list telling each of them to eat, either a set amount or whatever is remaining. The Herbivore list is scanned, and each of them eats as many plants as it can reach. Here is where a one of the Herbivore’s genes are expressed, as well as two of the Plant’s genes. Some of the plant energy is absorbed by the herbivore while the rest falls to the nutrient layer. As each plant is eaten it is erased from the screen, and deleted from the lists (Plant and Organism). Remember to erase from the background bitmap as well. Thirdly, the Carnivore list is scanned for hungry critters. Here is where those hunting genes could be expressed. Currently they search for prey much like the Herbivores eat Plants. They look for whatever is within their pouncing radius, and devour it. Again, the energy is exchanged through the system, the Herbivore is deleted from its lists, and erased from the display.

move() blanks the current location of the organism, in the foreground and on the background bitmap, calculates the new location with orgPtr→move(), and displays it in its new location.

die() has two routes through its routine, conditional upon the definition of SENESCENCE. There is something up with the Carnivore population. There is at least one of them in every run who proves to be immortal. This routine should limit that. A good assignment would be for the student to find out why there are immortal Carnivores. The routine checks if the Carnivore is over a certain age, then with a certain probability eliminates him from the book of life. If the condition is not defined, then we check orgPtr its energy level, then for its type. Is it a Plant, a Herbivore, or is it a Carnivore? They are all Organisms linked into the Organism list. If the energy level has dropped to zero (death by starvation) then the object is deleted from both lists.

reproduce() is fairly complex. First we scan the list for Carnivores of the proper age, who have enough energy. Once we find a suitable candidate a mate needs to be found. In this search there is no check for age nor energy (good idea for a student assignment), we just select the first candidate and exchange genetic material. {*Nota bene:* is this where the carnivore becomes immortal?} The object's location is used, and a nearby location is calculated from it. Here is a good place to use a gene for offspring distance from parent. Now the Carnivore free pool is contacted, and a new Carnivore object is requested. If there is one in the pool then all is well with the world. If there are none left then the simulation should die. This is a matter of calculating how many of each type of organism would possibly be necessary, at any given time over the course of a simulation. I took numbers from thin air and used them. I have exceeded them a few times - especially overpopulation of plants. Tweaking is necessary here. If there is a Carnivore in the free pool then put it into the Carnivore list, and delete it from the free pool Carnivore list. Each variable is set, all the chromosomes are created, and any mutations are applied. Energy levels are adjusted for the first parent, the offspring, and to the nutrient layer. Herbivores and Plants reproduce in almost exactly the same manner, differing only in which chromosomes involved. Once each species type has had a chance to reproduce, the offspring lists are appended to the main lists.

displayOrg() scans the Organism list, determines the type, then displays a pixel at the proper location with the proper color.

init() is the function which initializes the ecosystem. Blank the display and begin. Add nutrients to the nutrient layer array then initialize the free pool to create an inventory of Plants, Herbivores, and Carnivores ready for use in the simulation. Once the free pool is filled, a set number of Plants, Herbivores, and Carnivores is pulled from it. Once they have all been created, the Organism list is scanned, and each object is displayed according to its type.

Data gathering tools

gleanData() collects data during the simulation run. The number of organisms, plants, herbivores, and carnivores are collected as well as ages, both oldest and average.

storeData() saves data to a disk file. In order: generation, carnCount, carnOld, carnAvg, herbCount, herbOld, herbAvg, plantCount, plantOld, plantAvg, and the current time.

displayData() displays the generation and various species' populations in the lower right corner of the simulation.

State saving tools

EcoFileWrite()

EcoFileRead()

These are for saving and loading the simulation state at the time it was saved. Everything is saved for a snapshot of the events. I have not yet written the code to restore the saved state to a running system. Good idea for an assignment for that eager student :)

III Compile and run the code

- Open a terminal window
- Navigate to … S4L5_Ecology using cd (change directory in Linux or Windows using MinGW)
- Type **make** which will link and compile the application called Ecology.exe {Remember Win7/MinGW environment needed here}
- Type **wine Ecology** (in Linux) or **Ecology** (in Windows) to run the application
- Read gene.cpp, gene.h, organism.cpp, organism.h, plant.cpp, plant.h, animal.cpp, animal.h, herbivore.cpp, herbivore.h, carnivore.cpp, carnivore.h, world.cpp, world.h, ecosystem.cpp, and ecosystem.h in your editor to find how the code creates the application's output

IV Describe code line by line

Gene.h creates the class Gene with variables: m_sequence and m_length. The predicate m_ is used to designate class member variables in C++. The genetic operators are a little different than your normal method definitions. mutation() is normal by being included as a class member. However, crossover() and inversion() are, helper methods, external to the class. This is explained by how they are used. mutation() is a unary operator, while crossover() & inversion() are dyadic operators. The latter require an external gene to perform their work. getSequence() & setSequence() are used to examine or manipulate a gene. displaySequence() and displayGene() are used to check for errors. I will modify them to store, and restore, genes which survived for long periods, from previous runs. Notice the polymorphism of displayGene(), more on this later.

Gene.cpp defines the display and genetic operators declared in Gene.h. The genetic operators mutation(), crossover(), and inversion(), are defined using bit-wise operators. Each chromosome begins as an unsigned int, of 32 bits in length. The getLength() operator finds the length of the given chromosome. In this simulation there are 9, 12, 20, and 27 bit long chromosomes. This may change. crossover() has a mother chromosome, and a father chromosome. The crossover point is chosen at random, and used to clear out the unused portion of either chromosome, preparing them to be ORed together in line 81, to output the new gene. inversion() goes through the same steps as crossover(), but adds a reversal step. mutation() simply flips one bit in the sequence at the rate of 0.1%.

Organism.h has a chromosome which regulates the metabolism of each organism. The position, energy level, and age variables are added at this level. The m_metabolic chromosome has four genes: 3 bits for energy used per step, 3 bits for digestion efficiency, 3 bits for food value to the next trophic level, and 3 bits for the age of sexual maturity. These genes are expressed during eating, reproduction, and death. A few virtual functions are created as placeholders for future classes' modifications. move() will be defined at the Animal level. Notice, there is NO setAge() method. m_age is initialized to 1 at birth, and only modified in metabolize(). This is the advantage of having it as a private variable of the class. The other member variables each have their own get() and set() functions, usable by the outside world. Notice also die(). It, too, is a virtual function, but is partially defined here. It will be further defined later in the inheritance tree. die() examines the third Organism gene (food value to next trophic level), and adds that much energy to the nutrient layer at the dead organism's X,Y location. Birth, eating, and death each give some, or all, of an organism's energy back to the environment.

Organism.cpp defines one constructor, Organism(x, y, energy, age), and one class method, getGene(). getGene() returns the chosen gene to use as needed. Once again, the shift operator is used to retrieve the correct gene of the chromosome. Organism.h is also important, because it is where you change the size of the ecosystem's world: WORLDX, WORLDY. I have been using a 1.5 aspect ratio lately; 900 x 600, or 1050 x 700. 1600 x 900 worked, but slowly.

Plant.h inherits from Organism, and adds a plant chromosome with pollination, poison, and spine genes. We also see two plant list pointers. display() remains virtual, but is further defined to display each of its chromosomes, metabolic & plant. getGene() and setGene() are included for later. Plant.cpp defines getGene(num) to select the correct gene of the plant chromosome. The nine bit chromosome has three genes, each of three bits in length: pollination probability, poison probability, and spine probability. The latter could also be used as a spine length gene, to keep the herbivores at bay.

Animal.h inherits from Organism adding a movement chromosome. Two virtual methods define movement and display the animal's genes. getGene() and setGene() do their normal functions for the animal gene. Animal.cpp brings direction and direction probability arrays, and a move() routine so animals can search for food. There is no weighting to the direction probabilities, that is determined by genetics. The m_movement chromosome has 9 genes, 8 of them for direction probability and 1 for speed. Each gene has three bits so speed can range from 0 to 7 units per cycle. Animal::getGene() is used during reproduction, and during movement steps.

Herbivore.h inherits from Animal, and from Organism, adding a tactic chromosome. This chromosome has five genes, of four bits each, for: run, hide, and fight probabilities, camouflage effectiveness, and ability to eat marginal (thorny or poisonous) food. {*Nota bene:* for every measure there should be a countermeasure. This allows balance.} Herbivore.cpp adds the Herbivore::getGene() routine during the eating and movement routines.

Carnivore.h inherits from Organism and Animal, adding a tactics chromosome. This chromosome has six genes, of four bits each: probabilities for attack, rest, and hiding as well as ability to hide, attack range (0-15 units), and how long it will pursue its prey before resting. Carnivore.cpp defines the

Carnivore::getGene() routine so Carnivore::hunt() can use the tactics genes. The hunt routine needs to be filled in using the tactics chromosome genes.

World.h uses routines declared in Organism.h, Plant.h, Herbivore.h, and Carnivore.h, to create the four lists used by ecology.exe. This is where the inheritance tree forms a world of interacting lists. World.cpp is where the doubly-linked list add and delete routines are defined. This is where the most errors can be introduced. It gets complex since you are using four lists. Because plants, herbivores, and carnivores are all organisms, they show up on their own list, as well as on the organism list. This is also where memory leaks occur. Work needs to be done on the destructors for all the classes of the inheritance tree, so they work effectively, and don't cause the application to crash. Work has been done, bugs cleaned out, but still the destructors won't free memory. A new tack has been taken: a freePool of memory has been allocated, set aside for dynamic use during the simulation. It's still not working, so this app has a massive memory leak. It will only run until it uses all of the memory allocated by the OS for any given process (approximately 2 GB). However, this allows the simulation to last for about 60,000 generations; enough to get a good feel for the methods, and to allow for good genetic structure to be bred into each species. {This bug has been squashed. kjr}

Ecosystem.h ties the world to the routines it needs to become an ecosystem. Here is where eating, moving, dying, and reproducing are defined.

Ecosystem.cpp begins with a series of #define statements. Herbivore and carnivore thresholds are a little different. Herbivores have a browsing radius while carnivores have a pouncing or capture radius. Next are the energy levels necessary for each species to reproduce. The final #defines determine how much the nutrient layer is replenished.

WinMain() creates the world window, seeds the random number generator from the current time, and sets up the working threads. WndProc() handles the user interface thread, while scanOrgs() handles the work thread.

Ecosystem.exe uses heap memory, extensively. Each time an organism is born it allocates memory from the main pool. However, when each organism dies, the memory is not currently recovered. This memory leak is the main problem left to fix. {fixed} The difficulty is caused by the delete() functions. The complexity of C++ and the number of inheritance steps does not allow me to call delete() to free the memory without causing an error. I am working out a method to return this memory to the free pool. At the moment that is not working. I have added conditional compilation flags: lines 17 & 18. If either or both of these are uncommented you are running freePool() code. Leave them as they are until I fix this mess. By leaving them commented you can make Ecosystem.exe, run it, and watch the memory leak progress. I've been able to get 56,000 generations before ~2 GB of memory is filled, causing the system to crash. This is a valuable lesson, so I am leaving this bug intact until you learn it. I will fix this :) {fixed}

scanOrgs() is where the lion's share of the work takes place. It uses a variety of styles to call the various eco-functions: replenish the nutrient layer, eating, metabolizing, moving (for carnivores and herbivores), dying, and reproduction. I will go through all of these routines and give them the same working style. However, the system grew over a series of years while I was going to class, and working full time. There are over 40 versions of it in my files. I chose one which was suitably complex for this lesson. but does not have all of the functionality of the final version. This one does not have the extensive menu system of that one. I never did add the function I really wanted to implement: a way of saving the system at any moment to reload at a later time. You must remember the system 'breeds' better plants, herbivores, and carnivores as the generations pass. The genes are randomly created when the simulation begins. Often the ecosystem does not survive, because one of the species dies out or a population inversion occurs. If any of the plant, herbivore, or carnivore population drops too low the application halts. Being able to begin with the survivors from a long running ecosystem would be interesting. Even more fun would be the ability to examine the genes of the best individuals, to see why they are so well fit to the environment.

We'll dive in to the ecosystem routines by looking at the eat() function. Plants extract nutrients from the nutrient layer, at each plant's x, y location. Each plant requires a certain amount of energy, equal to its 'energy absorbed' gene (gene #2). If the nutrient layer has enough energy stored in it the plant extracts all it needs to live. Otherwise, the plant gets whatever it can from that location. Herbivores have a browsing range, so they scan that circular area around them for plants. Each herbivore has an efficiency gene (gene #1) which determines how much of the plant's energy they can consume. The excess energy should revert to the nutrient level but at the present does not do so {FIXED}. Each plant eaten is removed from the plant list. This is one point where the memory leak occurs. The deletePlant() function should link the old memory into the free memory pool for reuse but doesn't do so at the moment. Carnivores have a similar routine to eat their prey. They scan the herbivore list for any prey within their capture radius. They absorb energy from the herbivores at their genetically allotted rate (gene #3). The excess energy reverts to the Nutrient layer.

The move() routine is a wrapper function, which calls the move function of the animal level. There is no difference at this level. The present location is blacked out on both the foreground and background bitmaps. Finally, the routine calls the displayOrg() function, to redraw all the organisms with their own colors. Plants are redrawn too, for clarity.

Next we examine the die() function. In this case death is from natural causes; plants run out of nutrients, the animals starve. Scan the Organism list with the organism pointer. Each organism type is determined using the typeid() function. For instance, if the organism is a carnivore, the orgPtr is cast to a Carnivore pointer. Next we come to those dreaded delete() functions. More memory leakage until I can fix this. There is a pointer problem in my present return to freePool routines. I will fix this! {fixed kjr}

An aside: my Senior advisor was trying to make this application more and more complex by adding design patterns. In my opinion, added complexity for its own sake, without concern for solving the problem at hand, creates more locations for errors (and arguments with your advisor). That is the major reason I chose to start my rework from this version of the code. Any later versions have too much added complexity. I was able to compile this version under both Windows Visual Studio and with g++. Doing so was quite helpful for finding the errors which were causing crashes. By using two compilers

I got different warning and error messages. These messages lead me to different bugs. Mostly, they were uninitialized variables. I was ignoring those warnings but once I examined them I rid myself of a number of errors. By initializing all variables the error checking traps were more effective.

The reproduction routines come next. Scan the list of carnivores, determine whether or not there is enough energy to reproduce, if there is then scan the list again for a mate. (One error I trapped was in this routine. I fixed it by making sure there were at least two organisms which could exchange genetic material.) Select the three chromosomes of both parents: metabolism, movement, and tactics. Place the offspring at a random location within 5 units of the mother. Create a new carnivore from new memory. The energy is allocated thusly: 1\2 remains with the mother, 1/3 goes to the child, and 1/6 drops into the nutrient layer (afterbirth). The new carnivore is added into the linked list structure, and its chromosomes are created through crossover operations between the parent's sets of chromosomes. These same steps are repeated for herbivores. For plants it is slightly simpler because a plant only has two chromosomes to deal with. Once all the organisms have had a chance to reproduce, the new organism's list is appended to the original list of organisms.

gleanData() counts each type of organism, finds the oldest, and determines the average age of each cohort. storeData() sends the generation number, carnivore count, oldest carnivore age, and average age to a stored file. Similar herbivore and plant information is stored as well. Each line of data is time stamped then stored to a file called "data.txt". These files are good for setting initial conditions. The initial population of organisms is not a stable ecosystem, but after a few hundred generations the system stabilizes.

displayData() sends the generation number, carnivore population, herbivore population, and the plant population to the screen. These are printed with each species' color. This acts as a visual reference, to key the values to the screen.

displayOrg() is used by the move() routine to color each species' pixel, after it has moved to its new location. This really doesn't need to happen for plants, but it is nice for them to get redrawn after they have been stomped on by carnivores. init() blackens the entire window so there is no lower boundary between the simulation and the data display. Makes for a more elegant looking screen. The nutrient layer gets filled with its initial amount of energy, and the freePool of memory is created. The initial populations of plants, herbivores, and carnivores are created from the freePool.

The remaining functions are in a state of flux. They are for freePool memory adds and deletes. Once they are working the memory leak will be gone. The initial pool of memory will be allocated, then the initial populations will be created from the pool as it is now. But when any organism dies, or is eaten, the memory it had will be sent back to the freePool. The memory needed is allocated from the freePool when a new organism is created through reproduction. {fixed kjr}

V Describe makefile line by line

Line 1 defines the name of the app as eecology. The macro is used on line 26 and again on line 59. Line 4 sets warning levels to high. Line 5 shows the compiler where to find include files. Line 6 assures us we are using the correct compiler for C++. Line 7 is for when I get the menu and dialog boxes working. I will uncomment lines 54 and 55 then and compile the resources as well as the code.

Look at the makefile line 11. The two options: -static-libgcc & -static-libstdc++ let you create an executable which includes the .dll files. This allows you to send the .exe files to people who don't have MinGW set up on their computer. The app gets somewhat larger, but you don't need to include two extra .dll files. Just change line 27 to use LSFLAGS instead of LDFLAGS and you can create static code.

Line 20 sets the compiler flags to maximum optimization and the targeted version of Windows.

Lines 25 through 27 link the executable file Ecology (.exe) from the eight object files.

Lines 30 and 31 compile Gene.cpp and Gene.h into the object file Gene.o

Lines 33 and 34 compile Organism.cpp and Organism.h into the object file Organism.o

Lines 36 and 37 compile Plant.cpp and Plant.h into the object file Plant.o

Lines 39 and 40 compile Animal.cpp and Animal.h into the object file Animal.o

Lines 42 and 43 compile Herbivore.cpp and Herbivore.h into the object file Herbivore.o

Lines 45 and 46 compile Carnivore.cpp and Carnivore.h into the object file Carnivore.o

Lines 48 and 49 compile World.cpp and World.h into the object file World.o

Lines 51 and 52 compile Ecosystem.cpp and Ecosystem.h into the object file Ecosystem.o

To clean up while in the edit compile test mode use make clean which removes the object files (forcing every one to be compiled) and the executable (which may only force a relinking). Use make clean instead of simply deleting the executable. It forces a compilation of the entire application. This assures you any changes to an .rc file, or to a header file, will be compiled into the new executable.

VI Deeper

This program began when I needed a senior project. I was curious about genetic algorithms, and had a desire to learn C++ much better. I am fond of the wilderness and its life cycles. So I decided to put these all together, and build an ecosystem. It is based on an underlying nutrient layer, which is replenished by waste products from the organisms depending on it. Plants grow from these nutrients, which are fed upon by herbivores. These herbivores in turn are fed upon by carnivores. While I could have added another carnivore trophic level, I thought this was complex enough to make a good study.

Normally a genetic algorithm needs a fitness function, to determine which genes to use for the next generation. In this case, fitness is determined by survival. If you don't survive until the next generation your genes cannot be used for reproduction.

Creating a carnivore takes a number of steps. Everything is based on genes. Organisms inherit genes and add position, energy levels, and age. Organisms can be somewhere, have genes, age, and metabolize. Plants, herbivores, and carnivores each inherit these traits. Plants add their own genes: pollination, poison, and spines, as well as the ability modify their genes. Animals inherit from organisms, plus they add the ability to move, and the requisite movement genes. Herbivores inherit their traits from animals, adding genes for running, hiding, fighting, and camouflage. Carnivores also inherit from animal adding genes for attacking, resting, hiding, camouflage, attack range, and pursuit time. They have an another method which uses those genes during hunting.

This brings us to the 'is-a' 'has-a' set of relationships. A plant ‘is’ an organism, it 'has' genes. A herbivore 'is an' animal, it 'is' also an organism, it 'has' genes. A carnivore 'is an' animal, it too 'is an' organism, it 'has genes'. A world 'has' nutrients, plants, herbivores, and carnivores. An ecology 'has' the world and its own behaviors. C++ can trace inheritance with these ideas. Plants ‘inherit’ from organisms but they ‘have’ genes, which they express, and exchange with others of their own kind. So plants inherit the chromosomes of organism, carnivores, while herbivores and carnivores inherit the chromosomes of organism, and animal. They each add their own chromosomes. The last is what I’m trying to differentiate. Reading the code for reproduce() will show you better than I can describe it. The is-a relationship describes inheritance. The has-a relationship is the composition of different parts. A plant inherits all of its ‘behaviors’ except what its own chromosome expresses. A plant composites the chromosome containing the poison, spines, and pollination genes on to what it inherits from organism.

World is where the plant, herbivore, and carnivore lists are grouped into one assembly. The adding & deleting plants, herbivores, and carnivores methods are in world. Lastly we have ecosystem where all the parts get put together into an ecosystem simulation.

This is by far the most involved code you have seen from me. It does just what I want it to do. Getting the object recycling working took about six different tries. But once I really understood what the debugger was telling me (days of hard won knowledge), I was able to get rid of a lot of code, a couple superfluous loops, and a number of subtle glitches. After each rebuild it was running faster. When the app ran overnight I was overjoyed. That same app is still running. I have a number of generations of changes in the code I have sent you from the one that is running but I don't have the heart to shut it down. It is at 678,770 generations right now. {It got to 16 million before I shut it down}

Have fun running this app and reading the code. Ask questions I know this one is complex.

Current chromosomes

```
Metabolism        000      111      111      111           =  511
                  \ /      \ /      \ /      \ /
                 energy   percent  food     age of
                 used /    food    value    sexual
                 cycle   absorbed          maturity

Plant             111      111      111
                  \ /      \ /      \ /
                 polli    poison   spines
                 nation

Movement          111   111   111   111   111   111   111   111   111
                  \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /
                   N     NE    E    SE     S    SW     W    NW   speed

Herbivore         1111     1111     1111     1111     1111
tactics           \  /     \  /     \  /     \  /     \  /
                  run      fight    hide     camou     eat
                                             flage    marginal
                                                      food ability

Carnivore         1111     1111     1111     1111     1111     1111
tactics           \  /     \  /     \  /     \  /     \  /     \  /
                 attack    rest     hide     camou    attack   pursuit
                                             flage    range    time
```

I have been chasing a bug but haven't quite got it licked yet. I have been chasing this bug since I sent you the last lesson. But I have gotten the app to a point where it will run correctly, for approximately 6000 generations. Before it uses up all of the memory the OS allows it. There is a memory leak which is massive. Every deleted object is unhooked from a list and cannot be reused. So the app just allocates more memory from the OS until there is no more left. {fixed}

I have tried fixing this the way it should be done but am not having any success. When I get done using any chunk of memory I should be able to call delete() on that file handle and be safe. But not this time. When I try that I only get errors. The inheritance tree for this app is extensive and some of it is virtual. That is the main reason I cannot use delete() to free the memory. I am have a virtualization problem, and can't quite lick it.

The other way to fix this problem is to allocate all of the memory necessary for this app to run during initialization. What I have done is allocated this memory into a free pool of objects ready to be used. Then I create all of the initial population of organisms from that pool. However, when I try to return the used memory back to the free pool I keep getting errors. Not the same errors as above. These I am sure I can lick. I only need to figure out where all of the pointers are pointing, and I'll get it fixed.

This app is a true C++ app, with most of the bells and whistles. I had had classes in C++ but I never used it in code as extensive as this. You will find a lot of examples you can use later. But this is also a tour de force in multiply linked lists, and genetic algorithms. These organisms breed. Only the survivors gain enough energy to reproduce. There is no fitness function other than being able to survive. The organisms get better at survival as the simulation runs. The ecosystem becomes stable, with periodic changes in population levels. {Current test is at generation 411120.}

I have been expressing more and more of the genes, contained in each organism. Today I worked on the eating routines, so the energy not absorbed by the herbivores, or carnivores reverts back to the nutrient layer. This perked up the carnivore population quite a bit. They had been starving to death quite frequently with the former code. Now, instead of their being a decided food chain pyramid, the sides have become noticeably straighter, with far more carnivores and fewer plants. It used to be for every carnivore there were 100 herbivores, and for every herbivore there were about 100 plants.

I also have been playing with the size and aspect ratio of the simulation. Currently it is a rectangle. If you change it to a square you'll notice the carrying capacity of the ecosystem changes. It supports fewer carnivores and more plants.

The ecosystem becomes fairly stable after the initial 100 to 200 generations. You will get population inversions, and very large numbers of plants or herbivores until then. It is easy for the simulation to end because one of the species populations drops below the minimum during this period. The minimum number of any species is two, since the genetic operators use sexual reproduction.

I now have the recycling of organisms debugged and running well. I have also defeated the display glitch. Now you can minimize the window, let the app run in the background, and have it display properly when you reopen it. If it doesn't work correctly the first time, just toggle it again; there is a race condition between the two threads

There were a number of places where moving the object between the simulation and the free pool had pointer errors. About four days with a debugger found those. Most of them were corrected by capturing a pointer to where the next object would be. If an object was deleted, or not, this would point to the correct next object. I had to use a series of conditional compilations to find all of the pointer problems. The last one was defeated by using the same technique, one call level sooner (around the inner scanOrg() loop 129-137).

The other major errors I overlooked for quite some time. For instance, look in Ecosystem.cpp in the die() function, starting with line 294. The if() statement is asking which object type this pointer is pointing to. Normally, you can have a list of if() statements followed by other if() statements, asking the same thing with different answers. If you do this with numbers instead of pointers you'll see no errors. But here, things are very different. You are examining pointers then deleting them. If the first object in the list was a plant, and it was followed by a herbivore, you would get two deletions in this one function: die(). So here, and in another place with similar structure, I changed the series of if() statements into one where there can only be one path through the function, at each pass. I used the if, else if, else if … form. If any of these is true, it does its job, then drops through to the end of the function. That logic error caused a few subtle errors, and may have caused a few crashes. The debugger did not find it. I was staring at the same code for weeks, when all of a sudden I looked at it and really saw what may occur. Shazam!

That left one error: the display glitch. That took a few days of digging, with a few sideways searches. Then I found the answer in one line of code. What was happening, is any time the window would get covered, or minimized, it would lose contact between threads. The scanOrg() thread was still running just fine, and drawing to the background bitmap. But the UI thread was not finding the foreground bitmap for a refresh. Lots of frustration, but after taking my pruning clippers around the house I came back in, and found the answer. Look at Ecosystem.cpp again (in fact I am fairly certain all the changes I have made between this version and the last are in Ecosystem.cpp and Ecosystem.h) on line 35. The window class style CS_OWNDC means this app will create its own custom device context, and never let it go. You can now minimize the window and let things run in the background. It also speeds up the code a bit.

Currently, I have one sim running from two days ago. It is at generation 671,999 now but at 3 to 6 generations per second it increases quickly. The organisms have all become more efficient, and the browsing and hunting patterns are easy to follow. The next steps for this program are to keep expressing more genes, and craft a way to store and restore system states, with a means to examine genetic material.

I have been working on two ways of expressing a few of the genes. Herbivores have genes for running, fighting, and hiding (with a side order of camouflage). Carnivores have attack, rest, hide, and pursuit time (with camouflage too). I need to express these genes as behaviors. My first thought was to look at how you use all those dice in Dungeons and Dragons. That lead me to a few ideas.

if attack > fight   then eat()

if run > pursuit time then don't eat()

if attack > rest then eat().

Herbivores also have a gene for the ability to eat marginal food (spines or poison are the two choices). Plants have genes for poison and for spines. That lead me to

if ((poison probability + spines) < marginal food ability) then eat()

I also need to express the pollination gene. That lead me to the same method the herbivore or carnivore uses to choose where to eat. If the plant is within the pollination distance it can breed. This may take a fudge factor added on to make sure the radius covers enough ground; sometimes the plants can grow quite sparsely.

So that's the D & D way of solving the behavior modes of the animals. I thought things could be more interesting if I used neural nets with genetically determined weights, to solve the behavior mode choices. So more code to write, to do it both ways. It is my hypothesis that AI will arise through breeding neural nets, to make the best choices for their own survival. Woo hoo SkyNet!!!!

VII Assignments

1. Add menus and dialog boxes to control initial populations, etc.
2. Write code to load old states so you don't need to start from zero each time you run the simulation.
3. Implement the genes which have not been expressed.
4. Find out why there is an immortal carnivore.
5. Add more genes to the plants. Seed dispersal, pollen efficiency, etc.
6. Add another trophic layer. More than one carnivore layer would be nice.
7. Institute speciation. Limit ability to reproduce between certain populations of organisms.
8. Clean up gene chromosome terminology. Chromosomes are made up of genes. An organism has more than one chromosome.

VII Links

see Essay/Ecology.odt for an origin story.

CreateThread

https://msdn.microsoft.com/en-us/library/windows/desktop/ms682453(v=vs.85).aspx

Message loop

https://msdn.microsoft.com/en-us/library/windows/desktop/ms644928(v=vs.85).aspx#creating_loop

Constructors and Destructors in C++ or new & delete

http://www.cprogramming.com/tutorial/constructor_destructor_ordering.html

Destructors (C++)

https://msdn.microsoft.com/en-us/library/6t4fe76c.aspx

Constructors & Destructors

https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm

Using g++

http://www.cs.bu.edu/fac/gkollios/cs113/Usingg++.html

Debugging options

https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html

Virtuality

http://www.gotw.ca/publications/mill18.htm

When to use virtual destructors

https://stackoverflow.com/questions/461203/when-to-use-virtual-destructors#461224

Deleting a node in a doubly-linked list

http://www.geeksforgeeks.org/delete-a-node-in-a-doubly-linked-list/

Thread stack size

https://msdn.microsoft.com/en-us/library/windows/desktop/ms686774(v=vs.85).aspx

ofstream functions

http://www.cplusplus.com/reference/fstream/ofstream/

ShowWindow function

https://msdn.microsoft.com/en-us/library/windows/desktop/ms633548(v=vs.85).aspx

Running a process on specific CPU cores

http://xmodulo.com/run-program-process-specific-cpu-cores-linux.html

Topological sorting

https://en.wikipedia.org/wiki/Topological_sortingLtka

Lotka-Volterra equations

https://en.wikipedia.org/wiki/Lotka–Volterra_equations

Virtual destructors

http://www.studytonight.com/cpp/virtual-destructors.php

destructors

https://isocpp.org/wiki/faq/dtors

Destructors

https://msdn.microsoft.com/en-us/library/6t4fe76c.aspx

delete[] destructor problems

https://social.msdn.microsoft.com/Forums/en-US/b5a7d004-66db-45a8-8ec6-1fd544c28b16/problem-with-delete-and-destructors?forum=vcgeneral

Problems with delete in destructors

https://stackoverflow.com/questions/2569234/problems-with-delete-in-destructor

Destructors

https://ehsanakhgari.org/blog/2012-12-11/c-deleting-destructors

Virtual destructors

https://www.gamedev.net/forums/topic/452321-problem-with-virtual-destructors/

http://www.programmerinterview.com/index.php/c-cplusplus/virtual-destructors/

Class constructor and destructor

https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm

Virtuality

http://www.gotw.ca/publications/mill18.htm

https://www.w3resource.com/java-tutorial/inheritance-composition-relationship.php

Cost of virtuality

https://stackoverflow.com/questions/12756672/cost-of-virtuality-and-inheritance-for-non-virtual-members

IX Future work

With two editors and two compilers working on the same code, I can walk from one end of the house to the other while making changes in the code. Then run the code, and walk to the next computer. At one point I had five computers working on the one app; with the faster machines running two sims at once. Currently, ecosystem.exe is running on one of Pat's laptops at around 1,342,865 generations. I don't think it will crash again, its memory use is stable.

I have been working on ecosystem.exe since early June. I have gotten it running stably, from a free pool of objects. This is the most complex system of code I will use as an example. I wanted you to see what you could do with C++ by using most of its bells and whistles. I think I've accomplished that goal, and achieved a tool to test how ecosystems work. I added a function to write the complete state of the system at any time. This provides me with a means to look at the genetic structures, and how they change from snapshot to snapshot. I also have the data output file for generation by generation information.

It is the data file which is leading me to another program. Ecosystem generates the data: generation number, carnivore number, oldest, and average age, herbivore number, oldest, average, & plant number, oldest, average. I want to analyze that data for patterns. I have been reading about Fourier transforms and studied them in a few courses in grad school. What I have not done is actually use them. So I propose writing my version of the fast Fourier transform. It will take all of my population data and show me the patterns. At what frequencies do the population levels vary?

I also have the menu system I wrote for one of the versions of Ecosystem. I will add those menus to this code, to allow for a level of parameter control. That may take a while :)