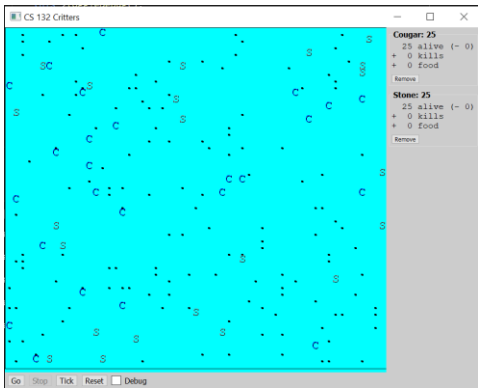
CS 132, Spring 2021
Programming Project #5: Critters (20 points)
Due: Thursday, February 24, 2022, 11:59 PM

Thanks to Marty Stepp and Stuart Reges for parts of this project

This assignment focuses on classes and objects. Turn in `Ant.h`, `Ant.cpp`, `Bird.h`, `Bird.cpp`, `Hippo.h`, `Hippo.cpp`, `Vulture.h`, `Vulture.cpp`, `Cat.h`, and `Cat.cpp`. Download the zip of the supporting files from the course web site. Run `main.cpp` to start the simulation.



Program Behavior:

You are provided with several client program classes that implement a graphical simulation of a 2D world of animals. You will write classes that define the behavior of those animals. Animals move and behave in different ways. Your classes will define the unique behaviors for each animal.

The **critter world** is divided into cells with integer coordinates. The world is 60 cells wide and 50 cells tall. The upper-left cell has coordinates (0, 0); x increases to the right and y increases downward.

Movement

On each round of the simulation, the simulator asks each critter object which direction it wants to move. Each round a critter can move one square **north**, **south**, **east**, **west**, or stay at its current location ("**center**"). The world has a finite size, but it wraps around in all four directions (for example, moving east from the right edge brings you back to the left edge).

This program will be confusing at first, because you do not write its `main` function; your code is not in control of the overall execution. Instead, your objects are **part of a larger system**. You might want your critters make several moves at once using a loop, but you can't. The only way a critter moves is to wait for the simulator to ask it for a single move and return that move. This experience can be frustrating, but it is a good way to practice object-oriented programming.

Fighting/Mating

As the simulation runs, animals may collide by moving onto the same location. When two animals collide, if they are from different species, they fight. The winning animal survives and the losing animal is removed from the game. Each animal chooses one of `ROAR`, `POUNCE`, or `SCRATCH`. Each attack is strong against one other (e.g. roar beats scratch) and weak against another (roar loses to pounce). The following table summarizes the choices and which animal will win in each case. To remember which beats which, notice that the starting letters of "Roar, Pounce, Scratch" match those of "**Rock, Paper, Scissors**." If the animals make the same choice, the winner is chosen at random.

		Critter #2		
		`ROAR`	`POUNCE`	`SCRATCH`
Critter #1	`ROAR`	random winner	#2 wins	#1 wins
	`POUNCE`	#1 wins	random winner	#2 wins
	`SCRATCH`	#2 wins	#1 wins	random winner

If two animals of the same species collide, they "**mate**" to produce a baby. Animals are vulnerable to attack while mating: any other animal that collides with them will defeat them. An animal can mate only once during its lifetime.

Eating

The simulation world also contains **food** (represented by the period character, `"."`) for the animals to eat. There are pieces of food on the world initially, and new food slowly grows into the world over time. As an animal moves, it may encounter food, in which case the simulator will ask your animal whether it wants to eat it. Different kinds of animals have different eating behavior; some always eat, and others only eat under certain conditions.

Every time one class of animals eats a few pieces of food, that animal will be put to "**sleep**" by the simulator for a small amount of time. While asleep, animals cannot move, and if they enter a fight with another animal, they will always lose.

Scoring

The simulator keeps a score for each class of animal, shown on the right side of the screen. A class's score is based on how many animals of that class are **alive**, how much food they have **eaten**, and how many other animals they have **killed**.

Provided Files:

Each class you write will **extend a superclass** named `Critter`. This is an example of inheritance, as discussed in class. Inheritance makes it easier for our code to talk to your critter classes, and it helps us be sure that all your animal classes will implement all the member functions we need. Your class headers should indicate the inheritance by writing `: public Critter`, like the following:

```
class Ant : public Critter { ...
```

The `Critter` class contains the following member functions, some / all of which you must write in each of your classes:

- `bool eat()`
  When your animal encounters food, our code calls this on it to ask whether it wants to eat (`true`) or not (`false`).
- `Attack fight(string opponent)`
  When two animals move onto the same square of the grid, they fight. When they collide, our code calls this on each animal to ask it what kind of attack it wants to use in a fight with the given opponent.
- `string getColor()`
  Every time the board updates, our code calls this on your animal to ask it what color it wants to be drawn with.
- `Direction getMove()`
  Every time the board updates, our code calls this on your animal to ask it which way it wants to move.
- `string getType() const`
  Every time the board updates, our code calls this on your animal to ask what its type name is.
- `string toString()`
  Every time the board updates, our code calls this on your animal to ask what letter it should be drawn as.

Just by writing `: public Critter` as shown above, you receive a **default version** of all of these member functions *except* `getType`. The default behavior is to never eat, to always forfeit fights, to use the color black, to always stand still (a move of `CENTER`), and a `toString` of `"?"`. If you don't want this default, rewrite (override) the member functions in your class with your own behavior.

For example, below is a critter class `Stone`. A `Stone` is displayed with the letter `S`, is gray in color, never moves or eats, and always roars in a fight. Your classes will look like this class, except with member variables, a constructor, and more sophisticated code. The `Stone` does not need an `eat` or `getMove` member function; it uses the default behavior for those operations.

```
#ifndef _STONE_H
#define _STONE_H

#include "Critter.h"

class Stone : public Critter {
public:
    Stone();
    virtual Attack fight(std::string) override;
    virtual std::string getColor() override;
    virtual std::string getType() const override;
    virtual std::string toString() override;
};

#endif // _STONE_H
```

NOTE: You are not necessarily required to write `: public Critter` on every single animal class you write. If you find that two animal classes are very similar to each other, you should have one extend the other to reduce redundancy.

Running the Simulator:

When you press the Go button, it begins a series of turns. On each turn, the simulator does the following for each animal:

- move the animal once (calling its `getMove` member function), in random order

- if the animal has moved onto an occupied square, fight! (call both animals' `fight` member functions) or mate
- if the animal has moved onto food, ask it if it wants to eat (call the animal's `eat` member functions)

After moving all animals, the simulator redraws the screen, asking each animal for its `toString` and `getColor` values. It can be difficult to test and debug with many animals. We suggest adjusting the initial settings to use a smaller world and fewer animals. You will find these settings in ***`Nursery.cpp`***. There is also a **Debug checkbox** that, when checked, prints console output about the game behavior.

Critter Classes:

The following are the five animal classes to implement. Each has one constructor that accepts exactly the parameter(s) in the table. For random moves, each choice must be equally likely; use `rand()`.

1. Ant

constructor	`Ant(bool walkSouth)`
color	red
eating behavior	always returns `true`
fighting behavior	always scratch

movement	if the Ant was constructed with a `walkSouth` value of `true`, then
	alternates between south and east in a zigzag (S, E, S, E, ...); otherwise,
	if the Ant was constructed with a `walkSouth` value of `false`, then
	alternates between north and east in a zigzag (N, E, N, E, ...)

toString	`"%"` (percent)

2. Bird

constructor	`Bird()`
color	blue
eating behavior	never eats (always returns `false`)
fighting behavior	roars if the opponent looks like an Ant (`"%"`); otherwise pounces
movement	a clockwise square: first goes north 3 times, then east 3 times, then south 3 times, then west 3 times, then repeats
toString	`"^"` (caret) if the bird's last move was north or it has not moved; `">"` (greater-than) if the bird's last move was east; `"v"` (uppercase letter v) if the bird's last move was south; `"<"` (less-than) if the bird's last move was west

3. Hippo

constructor	`Hippo(int hunger)`
color	gray if the hippo is still hungry (if `eat` would return `true`); otherwise white
eating behavior	returns `true` the first `hunger` times it is called, and `false` after that
fighting behavior	if this `Hippo` is hungry (if `eat` would return `true`), then scratches; else pounces
movement	moves 5 steps in a random direction (north, south, east, or west), then chooses a new random direction and repeats
toString	the number of pieces of food this `Hippo` still wants to eat, as a `string`

The **`Hippo` constructor** accepts a parameter for the maximum number of food that `Hippo` will eat in its lifetime (the number of times it will return `true` from a call to `eat`). For example, a `Hippo` constructed with a parameter value of 8 will return `true` the first 8 times `eat` is called and `false` after that. Assume that the value passed is non-negative.

The **`toString` member function** for a `Hippo` returns the number of times that a call to `eat` would return `true` for that `Hippo`. For example, if a `new Hippo(4)` is constructed, initially its `toString` return `"4"`. After `eat` has been called on it once, calls to `toString` return `"3"`, and so on, until the `Hippo` is no longer hungry, after which all calls to `toString` return `"0"`. You can convert a number to a string by calling `to_string` and passing it in as a parameter. For example, `to_string(7)` makes `"7"`.

4. Vulture

constructor	`Vulture()`
color	black
eating behavior	Returns `true` if vulture is hungry. A vulture is initially hungry, and he remains hungry until he eats *once*. After eating once he will become non-hungry until he gets into a fight. After one or more fights, he will be hungry again. *(see below)*
fighting behavior	roars if the opponent looks like an Ant (`"%"`); otherwise pounces
movement	a clockwise square: first goes north 3 times, then east 3 times, then south 3 times, then west 3 times, then repeats
toString	`"^"` (caret) if the vulture's last move was north or has not moved; `">"` (greater-than) if the vulture's last move was east; `"V"` (uppercase letter v) if the vulture's last move was south; `"<"` (less-than) if the vulture's last move was west

A `Vulture` is a specific sub-category of bird with some changes. Think of the `Vulture` as having a "**hunger**" that is enabled when he is first born and also by fighting. Initially the `Vulture` is hungry (so `eat` would return `true` from a single call). Once the `Vulture` eats a single piece of food, he becomes non-hungry (so future calls to `eat` would return `false`). But if the `Vulture` gets into a fight or a series of fights (if `fight` is called on it one or more times), it becomes hungry again. When a `Vulture` is hungry, the next call to `eat` should return `true`. Eating once causes the `Vulture` to become "full" again so that future calls to eat will return `false`, until the `Vulture`'s next fight or series of fights.

5. Cat

constructor	`Cat()` (must not accept any parameters)
all behavior	you decide (see below)

You will decide the behavior of your `Cat` class. Part of your grade will be based upon writing creative and non-trivial `Cat` behavior. The following are some guidelines and hints about how to write an interesting Cat.

Your `Cat`'s fighting behavior may want to utilize the `opponent` parameter to the `fight` member function, which tells you what kind of critter you are fighting against (such as `"%"` if you are fighting against an `Ant`). Your `Cat` can return any text you like from `toString` (besides `nullptr`) and any color from `getColor`. Each critter's `getColor` and `toString` are called on each simulation round, so you can have a `Cat` that displays differently over time. The `toString` text is also passed to other animals when they fight you; you may want to try to fool other animals.

Unlike on most assignments, your `Cat` **can use any advanced material** you happen to know in C++. If your `Cat` uses additional classes you have written, let me know to make sure it will be compatible with our system.

Each critter class has some **additional member functions** that it receives by inheritance from `Critter`. Your `Cat` may want to use these member function to guide its behavior. None of the member functions below are needed for `Ant`, `Bird`, `Hippo`, or `Vulture`.

- `int getX(), int getY()`
  Returns your critter's current x and y coordinates. For example, to check whether your critter's x-coordinate is greater than 10, you would write code such as: `if (getX() > 10) {`
- `int getWidth(), int getHeight()`
  Returns the width and height of the simulation grid.
- `string getNeighbor(Direction direction)`
  Returns a `string` representing what is next to your critter in the given direction. `" "` means an empty square. For example, to check if your neighbor to the west is a `"Q"`, you could write this in your `getMove` member function: `if (getNeighbor(WEST) == "Q") {`
- `void onWin(), void onSleep(), void onMate(), void onReset(), void onLose(), void onWakeup(), void mateEnd()`
  The simulator calls these member functions on your critter to notify you when you have won/lost a fight, been put to sleep/ wake up, start/end mating, or when the game world has reset, respectively.

Development Strategy:

You can run the simulator even if you haven't completed all the critters classes. You will see that `Nursery.cpp` initially has several lines of code commented out. This is to allow the simulator to run with just the `Stone` when you first download it and try it. Uncomment the lines that refer to each other critter type as you implement it.

The classes increase in difficulty from `Ant` to `Bird` to `Hippo` to `Vulture`. We suggest doing `Ant` first. Look at `Stone` and the lecture/lab example to see the general structure.

It will be impossible to implement each behavior if you don't have the right state in your object. As you start writing each class, spend some time thinking about what **state** will be needed to achieve the desired behavior.

One thing that students in the past have found particularly difficult to understand is the various **constructors** for each type of animal. Some of the constructors accept parameters that guide the behavior of later member functions of the animal. It is your job to **store data from these parameters into member variables** of the animal as appropriate, so that it will "remember" the proper information and will be able to use it later when the animal's other member functions are called by the simulator.

Test your code **incrementally**. A critter class will compile even if you have not written all of the member functions (although you must always implement `getType`). The unwritten ones will use default behavior. So, add one member functions, run the simulator to see that it works, then add another.

Style Guidelines:

Since this assignment is largely about using inheritance with classes and objects, much of the style grading will be about how well you follow proper **object-oriented** programming style. You should **encapsulate** the data inside your objects, and you should not declare unnecessary data members to store information that isn't vital to the state of the object. Style points will also be awarded for expressing each critter's behavior elegantly.

Style aspects of this program related to **inheritance** will be a grading focus as well. Your critter classes should properly extend the `Critter` superclass as described. Inheritance can also be used to remove redundancy between classes that are similar, and you should make use of this concept in your classes as appropriate. In other words, if two of your critter classes *A* and *B* are very much alike, you should have *B* extend *A* rather than having both simply extend `Critter`.

Some of the style points for this assignment will be awarded on the basis of how much **energy and creativity** you put into defining an interesting `Cat` class. These points allow us to reward the students who spend time writing an interesting critter definition. Your `Cat`'s behavior should not be trivial or closely match that of an existing animal shown in class.

Follow past style guidelines about indentation, spacing, identifiers, and localizing variables. Place comments at the beginning of each class documenting that critter's behavior, on top of each member function header, and on any complex code. Remember that your main public comments should be in your header files not in your cpp files.

Your critters should not produce any console output. It may be useful to output to the console while debugging and testing but all of this code should be removed before submission.