Fish/Shark Game Using Polymorphism

50 points - Due Friday, November 17, 11:59 PM

Assignment

The goal of this project is to build a simple grid-based shark/fish simulation game. This program should be written using polymorphism: create an **Animal** base class and derive **Fish** and **Shark** classes from it. There should be a 2-D virtual game board of a decent size: I used 60x20, but this is up to you. Each "slot" in the game should hold an **Animal*** pointer which is either NULL, or pointing to a **Shark** or a **Fish**. At the beginning, the game should randomly populate the board with equal numbers of each: say, 10 **Fish**es, 10 **Shark**s.

The rules of the game are as follows.

What Should Happen In Each Round:

1. Print the board on the screen. I'd suggest using using one character (like 'S') for sharks, another for fish, and leaving empty spaces blank.

2. Move around (or eat) each animal on the board, according to the rules below.

3. Wait for the user to press a key - if it's "Q", quit the game, otherwise do another round; go back to Step #1 (display the board again).

Sharks:

1. When it is a shark's turn to move, if there is an adjacent fish and the shark is hungry (hunger > 25), the shark moves to the spot where the fish is, and eats the fish. Whenever a shark eats a fish, that shark's hunger value goes down by 80; going below zero is fine.

2. Otherwise, the shark randomly moves to an open, adjacent position, and its hunger value increases by 1.

3. The shark's hunger should start at 50. If the shark's hunger goes above 100, the shark dies.

Fish:

1. When it is the fish's turn to move, the fish randomly moves to an open, adjacent position. There are no other rules; the fish dies when it gets eaten by a shark, but at no other time. Also, fish don't eat anything.

Polymorphism:

Here's the tricky part: the game logic is ***only*** allowed to use **Animal*** pointers: your logic code shouldn't even have to know that the **Shark** or **Fish** classes exist! In other words, your Animal base class will need to be well-designed and flexible enough so that the game logic can fulfill the above rules just by using the method in the **Animal** base class. This means that if we were later to add a **Turtle** class, for example, then we'd be able to do so without touching the game logic. This is what makes polymorphism cool. :-)

Something We Haven't Learned Yet:

If you are given a base class pointer (like an **Animal***), you might need to know whether this **Animal*** is actually a **Fish*.** You can do this using something called a **dynamic_cast**: this lets you try and transform an **Animal*** into a **Fish***. If C++ can't safely do this - because, for example, the **Animal*** is actually pointing to a **Shark** - then **dynamic_cast** will return NULL.

For example (and a heavy hint on how you might want to structure your project):

```
bool Shark::canEat(  Animal* a  )
{
    // see if a is pointing to an object of type Fish
    Fish* ptr = dynamic_cast<Fish*>( a );
    if( ptr )
       return true;   // we can eat Fish!
    return false;
}
```

We will go over dynamic_cast-ing in class on Wednesday, November 8.

Assignment Notes

- There are many hidden subtleties to this sort of project. Don't wait long to get started on it.
- Be careful when your fish/shark are on the borders of your grid: if you try and access pointers 'off the edge' of the grid, you'll very likely get crashes.
- You get a considerable amount of flexibility in how you design this project. Again, both **Fish** and **Shark** must derive from **Animal**, and you can only use **Animal*** in your game logic. Make good use of virtual functions. Otherwise, it's up to you, but for full credit, make good use of polymorphism and virtual functions.
- The specific numbers (80, 25, 100, etc.) in this project are only guidelines - if you want, play around with these and see what values give you a more stable simulation. With randomness involved, you'll never get anything that's perfectly stable, different numbers will make a big difference.

What to turn in: a directory containing...

- Your code, of course!
- A README file containing any relevant info, such as your name and compilation instructions (include the exact command you used to compile your code). Again, please include in the README the approximate amount of time you spent on this assignment. Also, the better you document your program/design in the README, the easier it will be for me to see what you were doing. This is always a good thing. :-)

Extra Credit

Add breeding: when two animals of any species end up in an adjacent grid (aka two **Fish**es or two **Shark**s) add some code so that there's a 20% chance they will breed, resulting in a new animal of that species randomly placed on the game board. Otherwise, the simulation will go on as normal. (Again, feel free to play around with that 20% number; you can even make it different for different species if you want.)

Non-specific Notes

- Start early; ask questions.
- Refer back to the syllabus for hints about writing good C++ programs (or programs in any language): only work on small pieces of code at a time, and test them well before moving on. Compile lots, and fix the warnings and errors. Write lots of comments.
- **Be sure your code compiles on the CS department's Linux machines before turning it in**. Even if you write it elsewhere (on a different OS, a different IDE, etc), give it a quick compile and test on a lab machine before you turn it in. Remember, if your code does not compile, it will get (at most) 50% credit.
- You must submit your code using the department's electronic submit procedure. A link to instructions for doing this is on the class website. Our course number (for the purposes of the submit program) is **c109sb**.
- Just a reminder: the CLAS academic (dis)honesty policies will be enforced. You're allowed to discuss the assignment with others, but not to collaborate or share code. Nor are you allowed to find or use code off the Internet.