Programming Assignment 3 (PA3)
Sea++ Episode 1: *"Under the Sea"*
User-Defined Types, Encapsulation, Operator Overloading, Interacting Objects, Model-View-Controller

Out: October 27, 2014, Monday -- DUE: November 15, 2014, Saturday, 11:59pm
EC327 Introduction to Software Engineering – Fall 2014

Total: 200 points

- *You may use any development environment you wish, as long as it is ANSI C++ compatible. Please make sure your code compiles and runs properly under the Linux/Unix environment on the PHO 305/307 (or eng-grid) machines before submitting.*
- *Labs may be submitted up to a week late at the cost of a **30% fixed penalty** (e.g., submitting a day late and a week late is equivalent). Any submissions after the deadline will be subject to the 30% penalty. No credit will be given to solutions submitted after the 1-week late submission period following the deadline.*
- *Follow the assignment submission guidelines in this document **AND** match the provided sample output document **exactly** or you will lose points.*

Submission Format (Must Read)
- Use the **exact** file names specified in each problem for your solutions.
- Complete submissions should have **20 files.** Put all of your files in a single folder named: **<your username>_PA3** (e.g., dougd_PA3), zip it, and submit it as a single file (e.g., dougd_PA3.zip).
- Submit your .zip to the PA3 portal on Blackboard by 11:59pm on the due date.
- Please do **NOT** submit *.exe and *.o or any other files that are not required by the problem.
- **Code must compile in order to be graded. Otherwise, this is an automatic zero.**
- Comment your code (good practice!). We **may** use your comments when grading.

Coding Style (reminder from PA2)
As you become an experienced programmer, you will start to realize the importance of good programming style. There are many coding "guidelines" out there and it is important that you start to adopt your own. Naming conventions will help you recognize variableNames, functions, CONSTANTS, Classes etc. Similarly, there are many ways to elegantly format your code so that it is easy to read. All of these issues do not affect compilation and hence are easy to overlook at this stage in your development as a programmer. However, your ability (or inability) to create clean, readable code could be the difference in your career when you leave college.

Here is a link you should explore for more on good coding conventions:

http://google-styleguide.googlecode.com/svn/trunk/cppguide.html

Sea++ Overview

In this programming assignment, you will be implementing a gaming simulation similar to "Warcraft," consisting of objects located in a two-dimensional world that move around and behave in various ways. The user enters commands to tell the objects what to do, and the Objects behave in simulated time. Simulated time advances one "tick" or unit at a time. Time is "frozen" while the user enters commands. When the user commands the program to "go", time will advance one tick of time. When the user commands the program to "run", time will advance several units of time until some significant event happens (*to be described later*).

In this assignment, you will be implementing the following **8 classes**:

- `CartPoint` – represents a point on a Cartesian coordinate system.
- `CartVector` – represents a vector in the real plane.
- `GameObject` – base class for all objects in the game.
- `CoralReef` – location that contains a certain amount of food. Inherits from `GameObject`.
- `Cave` – location where fish are safe from their predators. Inherits from `GameObject`.
- `Fish` – Generic sea creature. Inherits from `GameObject`.
    - Fish can swim to a specified location and stop.
    - Fish can eat at specific locations (`CoralReef`) and return "home" (`Cave`).
- `View` - Displays game objects. More details to come (*to be described later*).
- `Model` - Holds references to game objects. Model details to come (*to be described later*).

You will also provide a set of separate, related functions combined in one .cpp and one .h file:

- `GameCommand.cpp/.h` - Handles commands from the user input.

1. Class Specifications

Each class and its members are described below by listing its name, the prototypes for the member functions, and the names and types of the member variables. You **MUST** use the prototypes, types, and names in your program as specified here, in the same upper/lower case**.** Write all classes in their own `.h` and `.cpp` files. **Failure to follow these specifications will result in lost points.**

CartPoint (10 points)

This class contains two double values, which will be used to represent a set of (x, y) Cartesian coordinates. This class, together with the `CartVector` class described below, will be used to simplify keeping track of the coordinates of each object in the game, and updating their locations as they move. All data members and functions for this class should be public.

Public Members

- double x
  - the x value of the point.
- double y
  - the y value of the point.

Constructors

- The default constructor initializes x and y to 0.0
- CartPoint(double in_x, double in_y)
  - sets x and y to in_x and in_y, respectively.

Nonmember Functions

- double cart_distance(CartPoint p1, CartPoint p2)
  - returns the Cartesian (ordinary) distance between p1 and p2.

Nonmember Overloaded Operators (assume p1 and p2 represent two CartPoint objects, and v1 represents a CartVector object)

- Stream output operator (<<): produces output formatted as (x, y)
  - Example) If p1 has x = 3.14, y = 7.07 then cout << p1 will print (3.14, 7.07)
- Addition operator (+): p1 + v1 returns a `CartPoint` object with x = p1.x + v1.x and y = p1.y + v1.y
- Subtraction operator (-): p1 - p2 returns a **`CartVector`** object with x = p1.x - p2.x and y = p1.y - p2.y

CartVector (10 points)

This class also contains two double values, but it is used to represent a vector in the real plane (a set of x and y displacements). The overloaded operators allow one to do simple linear-algebra operations to compute where an object's new location should be as it moves around. Some of the overloaded operators and other functions can be member functions, others cannot. All data members and functions for this class should be public.

Public Members

- double x
  - the x displacement value of the vector.
- double y
  - the y displacement value of the vector.

Constructors

- The default constructor that initializes x and y to 0.0
- CartVector(double in_x, double in_y)
  - sets x and y to in_x and in_y, respectively.

Nonmember Overloaded Operators (assume v1 represents a CartVector object and d represents a ***non-zero*** *double value)*

- Multiplication operator (*): v1 * d returns a `CartVector` object with x = v1.x *d and y = v1.y * d
- Division operator (/): v1 / d returns a `CartVector` object with x = v1.x / d and y = v1.y / d
- Stream output operator (<<): produce output formatted as <x, y>
    - Example) If v1 has x = 5.3, y = 2.4 then cout << v1 will print <5.3, 2.4>

CHECKPOINT I
Start by writing the `CartPoint` and `CartVector` classes and a couple of their functions. Add the additional functions one at a time, and test each one. Write a `TestCheckpoint1.cpp` file with a `main` function to test them. Therefore, create multiple `CartPoint` and `CartVector` objects in order to test their constructors, the `cart_distance` method, and the overloaded operators (<<, +, -, *, /). Getting the overloaded output operator to work soon will make testing the remaining functions more fun.

DO NOT MOVE ON UNTIL YOU ARE 100% SURE YOU UNDERSTAND THIS CHECKPOINT

GameObject (20 points)

This class is the base class for all the objects in the game. It is responsible for the member variables and functions that they all have in common. It has the following members:

Private members:

- int id_num;

Protected members:

- CartPoint location;
    - The location of the object
- char display_code;
    - How the object is represented in the View
- char state;
    - State of the object; more information provided in each derived class

Public members:

- GameObject(char in_code, int in_id);
    - Initializes the display_code, id_num, and location to (0,0). It outputs the message: “GameObject default constructed”.
- GameObject(char in_code, int in_id, CartPoint in_loc);
    - Initializes the display_code, id_num, and location. It outputs the message: “GameObject constructed”.

- CartPoint get_location();
    - Returns the location for this object.
- int get_id();
    - Returns the id_num for this object.
- void show_status();
    - Outputs the information contained in this class: display_code, id_num, location. i.e. "<display_code> with ID <id_num> at location <location>". See sample output for exact formatting.

CoralReef (20 points)

This class is derived from `GameObject`. It contains food, which fish eat and in turn gain in size. It has a limited amount of food which will be consumed throughout the process of the game.

Private Members

- double amount
    - The amount of food currently in the reef.
    - Initial value should be set to 100.

Inherited Members from GameObject with CoralReef-specific initialization values

- char display_code
    - Initial value should be set to 'R'. Change to 'r' when reef is depleted.
- char state
    - Initial value is 'f' for "full of food". Changes to 'e' when reef is depleted.

Constructors

- The default constructor that initializes the member variables to their initial values (display_code = 'R' and state = 'f').
- Prints out the message "CoralReef default constructed".
- CoralReef(int in_id, CartPoint in_loc)
    - Initializes the id number to in_id, and the location to in_loc, and display_code and state should be set to their initial values listed directly above.
    - Prints out the message "CoralReef constructed".

Public Member Functions

- bool is_empty()
    - Returns true if this CoralReef contains 0 food. Returns false otherwise.

- double provide_food(double amount_to_provide = 20.0)
    - If the amount of food in the reef is greater than or equal to amount_to_provide, it subtracts amount_to_provide from CoralReef's amount and returns amount_to_provide. If the amount of food in the reef is less, it returns the CoralReef's current amount, and amount is set to 0. **NOTICE THIS HAS A DEFAULT ARGUMENT VALUE; Chapt 6.7.**

- bool update()
    - If the reef is empty, it sets the state to ‘e’ for “empty”, change display_code to ‘r’, prints the message “CoralReef <id_num> has been depleted” and lastly, returns true. Returns false if it is not depleted.
    - This function shouldn’t keep returning true if the CoralReef is depleted. It should return true ONLY at the time when the CoralReef becomes empty, and return false for later “update()” function calls.

- void show_status()
    - Prints out the status of the object (display_code, id_num, location, and the amount of food). *See the sample output for the format.*

CHECKPOINT II
Write the `GameObject` and `CoralReef` classes. To test these classes, also write a `TestCheckpoint2.cpp`. Instantiate multiple objects of these classes in the `main` function and test out their functions (e.g. `update()`, `provide_food()`, etc.) in order to ensure their proper behavior. For example, call each object’s `show_status()` method after calling their `update()` method.

DO NOT MOVE ON UNTIL YOU ARE 100% SURE YOU UNDERSTAND THIS CHECKPOINT

Cave (20 points)

A Cave object represents a home for `Fish`. It is derived from `GameObject` and has a location and a certain amount of space. It has the following members:

Private Members

- double space
    - The amount of space left in the cave.
    - Initial value should be set to 100.

Inherited Members from GameObject with Cave-specific initialization

- char display_code
    - Initial value should be set to ‘c’.
- char state
    - Initial value is ‘e’ for “empty”.

Constructors

- The default constructor initializes the member variables to their initial values listed directly above.
    - Prints out the message “Cave default constructed”.

- Cave(int in_id, CartPoint in_loc)
  - Initializes the id number to in_id, and the location to in_loc, and remainder of the member variables to their default initial values as listed above.
  - Prints out the message “Cave constructed”.

Public Member Functions

- bool hide_fish(Fish* fish_to_hide)
  - Attempts to hide the fish if there is space based on the fish’s size. Updates the available space if the fish hides in the cave. Must check if the fish is not hidden yet. Returns true if it has enough space to hide the fish, false otherwise

- bool release_fish(Fish* fish_to_release)
  - Attempts to release fish if the fish is hidden. Updates the available space once a specified fish leaves. Returns true if the fish is successfully released. Returns false if the fish is not originally hidden.

- bool update()
  - If the Cave has space equal to 0, it sets the state to ‘p’ for “packed”, change display_code to ‘C’, prints the message “Cave <id_num> is packed” and lastly, returns true. Returns false if space is greater than 0.
  - This function shouldn’t keep returning true if the Cave is packed. It should return true ONLY at the time when the Cave gets packed, and return false for later “update()” function calls.

- void show_status()
  - Prints out the status of the object (display_code, id_num, location, and the amount of space, and a message about whether the Cave is packed or not). *See the sample output for the format.*

How the CoralReef and Cave Behave

These objects change state, but do so very simply. The `CoralReef` update function simply checks to see if the reef is empty; if so, it changes the state to 'e' for empty, and signifies a deteriorated reef by changing the display_code letter from 'R' to 'r', and returns true to signify that an event has happened. Similarly, the `Cave` update function checks to see if the amount of space is 0; if so, it changes state to 'p' for "packed" and changes the display_code letter from 'c' to 'C' to show that the cave capacity is diminished.

Fish (30 points)

The `Fish` class represents a simulated fish character in the game. It inherits from the `GameObject` class. It can be told to move to a specified location, and it does so at a certain speed. When it arrives to the location, it stops. It can also be told to start eating at a specified `CoralReef` and hide in a specified `Cave`. The `Fish` object keeps track of the `CoralReef` and the `Cave` by keeping **pointers** to the two objects, which allows it to access their locations, extract food from the reef, and hide in a cave. The behavior of the `Fish` class is mainly represented in the update() function, and the functions and variables associated with making it move around.

Private Members

- double energy
    - The health of the fish.
    - Initial value is 15.

- double size
    - The size of the fish.
    - Initial value is 10.

- double prefeast_size
    - The size of the fish before eating.
    - Initial value is initial size of fish.

- CartPoint destination
    - This object's current destination coordinates in the real plane.
    - CartPoint's default constructor will initialize this to (0.0, 0.0).

- CartVector delta
    - Contains the x and y amounts that the object will move on each time unit.

- CoralReef* reef
    - A pointer to the CoralReef to be eaten from.
    - Initial value should be NULL.

- Cave* home
    - A pointer to the Cave where the fish will hide.
    - Initialized to the birthplace of the fish.
    - Can be changed by the `start_hiding` function
    - Default constructor sets this to NULL.

- bool update_location()
    - Updates the object's location while it is moving. Print "I'm here" or "Just keep swimming" depending on whether it has arrived at its destination. *See sample output for exact format*.
    - Returns true when the fish arrives.

- void setup_destination(CartPoint dest)
    - Sets up the object to start moving to dest. (Setting destination pointer and calculating delta (explained in "How fish behaves"))

Inherited Members from GameObject with Fish-specific initialization

- char display_code
    - Initial value should be set to ‘F’.
- char state
    - Initial value is ‘h’ for “hidden” inside Cave.

Constructors

- The default constructor that initializes a Fish object using the initial values listed above.
    - Prints the message “Fish default constructed”.

- Fish(int in_id, Cave* home)
    - Sets the ID number and initial location of the object using the supplied values and the remainder of the member values using the values listed above.
    - Hide the Fish inside the home Cave (decreasing Cave space by size).
    - Prints the message “Fish constructed”.

Public Member Functions

- bool update()
    - Returns true if the state has changed.
    - Updates the status of the object at each time unit.
    - See Section entitled “How the Fish behaves” for more information.

- double get_size()
    - Returns the size for this fish

- void start_swimming(CartPoint dest)
    - Tells the Fish to start swimming towards a given destination.
    - Calls the setup_destination() function.
    - Sets the state to ‘s’ for “Swimming.”
    - Prints the message “Swimming (id) to (destination)” and another message “<display_code><id_num>): On my way” (See sample output)

- void start_eating(CoralReef* destReef)
    - Tells the Fish to start eating at the CoralReef given. After it has doubled in size it will return back to its Cave home.
    - Calls setup_destination() function.
    - Sets the state to ‘o’ for “Outbound”
    - Prints “Fish <id_num> eating at CoralReef <id_num> and returning back to Cave <id_num>.” and another message “<display_code><id_num>: Food here I come”

- bool is_hidden()
    - Returns true if the state of this fish is hidden. Otherwise it returns false.

- void start_hiding(Cave* destCave)

    - Tells fish to go hide at the Cave given
    - Sets its Cave home to destCave
    - Calls setup_destination() function
    - Sets the state to ‘z’ for “Zooming”
    - Prints “Fish <id_num> swimming to Cave <id_num>” and another message “<display_code)><id_num>: Off to safety”.

- void float_in_place()
    - Tells the Fish to stop moving, eating, or hiding.
    - Sets the state to ‘f’ for “Floating”.
    - Prints “Stopping <id_num>.” and another message “<display_code)><id_num>: Resting my fins”.

- void show_status()
    - Prints out the status of the object (display_code, id_num, current location, and additional description depending on the state of the object as described below).

- double get_speed()
    - Returns the speed of the fish which is: (1/size)(energy)*5

How the Fish Moves

The main function of the program accepts commands from the user and will be explained later. Simulated time is stopped while the user enters commands. The user can command individual objects to move to specified destination coordinates. When the user tells the program to “go”, one step of simulated time then happens, and the program calls the update function on every object. The program then pauses to let the user enter more commands, and when the user commands “go” again, another step of simulated time happens, and every object is updated again. So each “go” command corresponds to one “tick” of the clock, one step of the simulated time. The “run” command conveniently makes the program run until an important event happens (*to be defined*).

A Fish is commanded to move by calling its `start_swimming` function and supplying the destination. The `start_swimming` function does the following: Call the `setup_destination` function to save the destination and calculate the delta value. Then set the object in the moving state. The delta value contains the amount that the object’s x and y coordinates will change on each update step. We calculate it once, and then apply it on each step. This is the purpose of the overload operators for `CartPoint` and `CartVector`. To calculate the value of delta, use:

delta = (destination – location) * (speed / cart_distance(destination, location))

In other words, the object will move on a straight line, moving a distance equal to its speed on each step. The change in the x and y values of the location on each step are thus proportional to the ratio of the speed to the distance. So the `setup_destination` function calculates the delta value to be used in the updating steps.

The `update` function for Fish does a variety of things, but if the object is moving, it calls the `update_location` function. This function first checks to see if the object is within one step of its destination (see below). If it is, `update_location` sets the object's location to the destination, prints an "arrived" message (see sample output), and then returns true to indicate that the object arrived. If the object is not within a step of destination, `update_location` adds the delta to the location, prints a "moved" messaged, and returns false to indicate that the object has not yet arrived. Thus the object will take a "speed-sized" step on each update "tick" until it gets within one step of the destination, and then on the last step, goes exactly to the destination. On each step of simulated time, the Model will call each object's `update` function *(more details to be provided later*).

Finally, notice that the user can command all of the Fish objects to move to a destination, and then tell the program to "go", and all of the objects will start moving, and each will stop when it arrives at its own destination. The objects are responsible for themselves!

***An object is within a step of the destination if the absolute value of both the x and y components of fabs(destination - location) are less than or equal to the x and y components of delta (use the `fabs()` function in `math.h` library). By checking our distance with very simple computations using the delta value, we don't have to calculate the remaining `cart_distance` and compare it to the speed using a slow square root function on every update step.*

*** If the fish is currently hidden in a cave and is asked to swim away, the cave's space should increase and change its status from packed to normal if applicable. (Hint, use the pointer to home to release the fish)*

How the Fish Behaves

The behavior of the `Fish` class is programmed using an approach called a "state machine". A state machine is a system that can be in one of several states and behaves depending on what state it is in. It can either stay in the same state, or change to a different state and behave differently. The neat feature of this approach is that you can easily specify a complicated behavior pattern by simply listing the possible states, and then with each state, describe the input/output behavior of the machine and whether it will change state (See http://en.wikipedia.org/wiki/Finite-state_machine for detail).

In the `Fish` class, the update function will do something different depending on the state of the `Fish` object. The state is represented with a simple char variable that contains a code letter for the particular state. A good way to program this is to use a switch statement that switches on the state variable and has a case for each possible state. In each case, perform the appropriate action for the state, and if needed, change the state by setting the state variable to a different value. Then the next time the update function is called, the `Fish` will do the appropriate thing for the current state. Thus the update function contains nothing but a big switch statement.

Here are the states of the Fish and what the update and `show_status` functions do for each state (note that update function also does some printing). Generally, the update function should return true whenever the state is changed and return false if it stays in the same state:

- 'f' for "floating."
  - The Fish does nothing and stays in this state.
  - show_status() prints "Is Floating. Has a size of: <size> And energy of: <energy>"

- 's' for "swimming to a destination"
  - Call update_location to take a step; if the object has arrived, set the state to 'f' and return true. Otherwise, stay in the swimming state.
  - show_status() prints "Has a speed of: <speed> And is heading to (x,y)"

- 'o' for "outbound swimming to eat"
  - Call update_location to take a step; if it has arrived, set the state to 'e'', and return true; otherwise stay in the outbound state.
  - show_status() prints "Is outbound to Reef: <id_num> With a speed of <speed>"

- 'e' for "eating"
  - Store the current size to prefeast_size.
  - Call the CoralReef provide_food function.
    - If the Reef runs out of food, set the state to 'f' and print "<display_code><id_num>: This reef has no more food for me"
    - If there is enough food, the fish grows in size proportional to a fourth (1/4) of the eaten amount.
  - Print "<display_code><id>: Grew to size <size>"
  - Call setup_destination once it has grown to at least twice its size to go back to its Cave *home.
  - Set the state to 'z' and return true.
    - show_status() prints "is eating at Reef <id_num>".

  .
- 'z' for "zooming to home"
  - Call update_location.
    - If it has arrived call Cave hide_fish function to check if there is space.
      - If there is space set the state to 'h'
        - Print "<display_code><id_num>: I am hidden and safe"
      - If there is no space set the state to 'p'
        - Print "<display_code><id_num>: Help! My home is full"
  - show_status() prints "Is swimming to Cave <id_num> With a speed of <speed>"

- 'h' for "hiding"

    - Print “<display_code><id_num>: Is hidden in Cave <id_num>”.
    - **Fish will need to check if it is hidden itself before doing anything else.**

- ‘p’ for “panicked”
    - update() does nothing for this case.
    - show_status() prints “<display_code><id>: Is panicked”

Thus if the `Fish` is commanded by `start_moving` to move to a destination, it goes into the moving state, starts moving, and then stops when it arrives and does nothing until commanded again. If the Fish is commanded by `start_eating` at `CoralReef`, the following happens: The start eating function sets the `CoralReef` and `Cave` pointer member variables, calls `setup_destination` to prepare to move to the reef, and sets the state to 'o'. The fish goes to the reef, eats until it is at least twice its original size and then calls `setup_destination` to prepare to move to a cave (If the Reef is out of food, the fish simply stops). When it gets there, it checks to see if it can fit in the cave. It hides in the cave if it can fit or it panics if it cannot. Note that it spends a whole update time "tick" in the cave, and at the `CoralReef`, before setting out again.

Checkpoint III
Iteratively develop the `Cave` and `Fish` classes. Start implementing the classes by only writing the constructors and a partial form of the `show_status` function; leave everything else out. To test the correct behavior, write another simple `TestCheckpoint3.cpp` file. In the `main` function, create a Fish and Cave object, and call its `show_status` function. It should display the correct initial state of both objects, and so you can verify if both the constructors and the `show_status` function work properly. Only then start implementing the `start_swimming`, `setup_destination`, and `update_location` functions, and the 's' and 'f' parts of the update and `show_status` functions. Change your trivial main function to call the `start_swimming` function on your one Fish object, showing its status, calling its `update` function, and showing its status again - it should be in a different location! Check that amount moved on the step is correct. Put in a few more calls of update and see if the Fish stops like it should. With the help of CoralReef and Cave objects, you are able to get your fish to eat and hide as well.

DO NOT MOVE ON UNTIL YOU ARE 100% SURE YOU UNDERSTAND THIS CHECKPOINT

The Model-View-Controller (MVC) Pattern

Model (15 points)

The Model is a central component in the MVC pattern and stores all game objects in-memory. Hence, it contains various arrays of pointers to the instances of the GameObject class. Also it offers multiple methods to interact with the Controller and View components. Here, it has the following structure:

Private members:

- `int time;`
    - The simulation time.

We have a number of arrays of pointers and a variable for the number in each array:

- `GameObject** object_ptrs;`
- `int num_objects;`
- `Fish** fish_ptrs;`
- `int num_fish;`
- `Cave** cave_ptrs;`
- `int num_caves;`
- `CoralReef** reef_ptrs;`
- `int num_reefs;`

Each object will have a pointer in the `object_ptrs` array and also in the appropriate other arrays. For example, a Fish object will have a pointer in the `object_ptrs` array and in the `fish_ptrs` array.

Note: make the copy constructor private to prevent a Model object from being copied

Public members:

- `Model();`

This initializes the time to 0, then creates the objects using **new**, and stores the pointers to them in the arrays as follows:

The list shows the object type, its id number, initial location, and subscript in object_ptrs, and the subscript in the other array.

```
Cave: id: 1, display_code: ‘c’, location: (5, 1) - object_ptrs[0], cave_ptrs[0]
Cave: id: 2, display_code: ‘c’, location: (6, 2) - object_ptrs[1], cave_ptrs[1]
Fish: id: 1, display_code: ‘F’, birthPlace: Cave 1 - object_ptrs[2], fish_ptrs[0]
Fish: id: 2, display_code: ‘F’, birthPlace: Cave 2 - object_ptrs[3], fish_ptrs[1]
CoralReef: id: 1, display_code: ‘R’, location: (1, 20) - object_ptrs[4], reef_ptrs[0]
CoralReef: id: 2, display_code: ‘R’, location: (20, 1) - object_ptrs [5], reef_ptrs [1]
```

```
CoralReef: id: 3, display_code: ‘R’, location: (20, 20) - object_ptrs [6], reef_ptrs [2]
```

Set `num_objects` to 7, `num_fish` to 2, `num_reefs` to 3, `num_caves` to 2;

Finally, output a message: "Model default constructed";

```
~Model();
```

The destructor deletes each object, and outputs a message: "Model destructed."

Note: For purposes of demonstration, add to `GameObject` a destructor (`~GameObject()`) that does nothing except output a message: "GameObject destructed" Define similar destructors for `Fish, CoralReef`, and `Cave`. This is so that you can see the order of destructor calls for an object.

Important: Make the destructors in GameObject virtual. Think about why this is important!

In the Model, there are three functions that provide a lookup and validation services to the main program and GameCommand (Controller). They return a pointer to the object identified by an id number, depending on the type of object we are interested in. The functions search the appropriate array for an object matching the supplied id, and either return the pointer if found, or 0 if not.

```
Fish* get_Fish_ptr(int id_num);
```

- This function iterates over the fish_ptrs array and it returns the pointer to the Fish object that has the ID id_num. If there’s no Fish object with the id_num ID, then it returns NULL.

- `CoralReef* get_CoralReef_ptr(int id_num);`

This function iterates over the reef_ptrs array and it returns the pointer to the CoralReef object that has the ID id_num. If there’s no CoralReef object with the id_num ID, then it returns NULL.

- `Cave* get_Cave_ptr(int id_num);`

This function iterates over the cave_ptrs array and it returns the pointer to the Cave object that has the ID id_num. If there’s no Cave object with the id_num ID, then it returns NULL.

- `bool update();`

This function provides a service to the main program. It increments the time, and iterates through the object_ptrs array and calls update() for each object. Since GameObject::update() will be made virtual, this will update each object.

- `void display(View& view);`

Likewise it provides a service to the main program. It outputs the time and generates the view display for all of the GameObjects.

- `void show_status();`

It outputs the status of all of the GameObjects by calling their show_status() function.

Implement the polymorphism in the class hierarchy as follows:

- Declare bool update() to be a **pure virtual function** in GameObject. This makes GameObject an abstract base class and ensures that each of the derived classes will have defined an update function, or we get a linker error to tell us we have left it out.
- Declare show_status() to be virtual in GameObject.

In your final main function declare a single Model object.

GameCommand.h and .cpp (15 pts)

The `GameCommand` represents the Controller of the MVC pattern and provides multiple functions that interpret user input in order to perform the appropriate actions.

You should create a set of functions that can be called from main to handling the processing of user provided commands.

Here is a description of the commands and their input values:

- **s** ID x y
  - “swim": command Fish with ID to move to location (x, y)
  - Runs the `start_swimming` function of Fish ID1
- **e** ID1 ID2
  - "eat food": command Fish with ID1 to start eating at CoralReef with ID2
  - Runs the `start_eating` function of Fish ID1
- **f** ID
  - "float in place": command fish with ID to stop doing whatever it is doing and just floats at its location
  - Runs the `float_in_place` function of Fish ID1

- **z** ID1 ID2
    - "zoom to Cave": command fish with ID1 to start hiding at Cave with ID2
    - Runs the `start_hiding` function of Fish ID1
- **g**
    - "go": advance one time step by updating each object once.
- **r**
    - "run": advance one time step and update each object, and repeat until either the update function returns true for at least one of the objects, or 5 time steps have been done.

- **q**
    - "quit": terminate the program

You must have a separate command-handling function for each command that collects the input required for the command and calls the appropriate object member functions. We recommend using the switch statement in main to pick out the function for each command; this is the simplest and cleanest way to do this sort of program branching. For example, to handle the "move" command, the case would look like this:

```
case 's':
        do_swim_command(model);
        break;
```

All input for this program must be done with the stream input operator >>. **You do NOT need to do error checking. This will be done in PA4.**

Important*: Your program must "echo" each command to confirm it and to provide a record in the output of what the input command was. For example, if the user enters "m 1 10 15" the program should output something like "moving Fish 1 to (10, 15)". This way, if output redirection is used to record the behavior of the program, the output file will contain a record of the input. If there is an error in the input, there should be an informative error message, but your program is not responsible for trying to output an exact copy of the erroneous input. So the echo does not have to be present or complete if there is an error in the input.*

Each command function should be given just the `Model` object (by reference), as in:

- `void do_swim_command(Model& model);`
- `void do_go_command(Model& model);`
- `void do_run_command(Model& model);`
- `void do_eat_command(Model& model);`
- `void do_float_command(Model& model);`
- `void do_zoom_command(Model& model);`

The command functions should use the Model member functions, such as `get_Fish_ptr`, to check that an input ID number is valid and get a pointer for the object if it is. The `do_go_command` and `do_run_command` functions can just call the Model's update method to update the status of all the objects. After Checkpoint 4, the Model's display method can also be called to display the current View of the game. Move the prototypes of these do_***_command functions into the `GameCommand.h` file, and define the functions in the corresponding `GameCommand.cpp` file.

CHECKPOINT IV
Implement the Model and the GameCommand.cpp/.h and add them to your program. You now should be able to actively have a user manually enter commands and have them play your game. Write a TestCheckpoint4.cpp file and in the main function command the fish to swim around, eat, etc, and list the status of all game objects. Make sure you comment out parts that reference View as that does not exist yet. When the program terminates, you should see the correct sequence of destructor messages.

DO NOT MOVE ON UNTIL YOU ARE 100% SURE YOU UNDERSTAND THIS CHECKPOINT

All what is missing at this point is a graphical view of the game. So, we set up a simple display using "ASCII graphics." This display will be like a game board – a grid of squares. Each object will be plotted in the grid corresponding to its position in the plane. As the object moves around its position on the grid will be changed. The object will be identified on the board by its display_code letter and its id_num value. **We will be assuming that the id_nums are all one digit in size.** We will provide sample output to show what the display should look like.

The code for the display will be encapsulated in a class of object called a "View" whose function is to provide a view of some objects. The View object has a member function plot() that puts each object into display grid; it will ask GameObject to provide the two characters to identify itself. Member function draw() will output the completed display grid, and clear() will reset the grid to empty in preparation for a new round of plotting.

Thanks to inheritance we can add the following public member function to the GameObject class:

- `void drawself(char* grid);.`
    - The function puts the display_code at the character pointed to by grid, and then the ASCII character for the id_num in the next character.

View (10 points)

Constant defined in the header file

- `const int view_maxsize=20;`
  - The maximum size of the grid array

Private members:

- `int size;`
  - The current size of the grid to be displayed; not all of the grid array will be displayed in this project.

- `double scale;`
  - The distance represented by each cell of the grid.

- `CartPoint origin;`
  - The coordinates of the lower left-hand corner of the grid.

- `char grid[view_maxsize][view_maxsize][2];`
  - An array to hold the characters that make up the display grid.
- `bool get_subscripts(int &ix, int &iy, CartPoint location);`
  - This function calculates the column and row subscripts of the grid array that correspond to the supplied location. Note that the x and y values corresponding to the subscripts can be calculated by (location - origin) / scale. Assign these to integers to truncate the fractional part to get the integer subscripts, which are returned in the two reference parameters. The function returns true if the subscripts are valid, that is within the size of the grid being displayed. If not, the function prints a message: "An object is outside the display" and returns false.

Public members:

- `View()`
  - This sets the size to 11, the scale to 2, and lets the origin default to (0, 0). No constructor output message is needed.
- `void clear();`
  - This sets all the cells of the grid to the background pattern shown in the sample output.
- `void plot(GameObject* ptr);`
  - This plots the pointed-to object in the proper cell of the grid. It calls `get_subscripts` and if the subscripts are valid, it calls the drawself() method for the object to insert the appropriate characters into the cell of the grid. However, if there is already an object plotted in that cell of the grid, the characters are

replaced with ‘*’ and ' ' to indicate that there is more than one object in that cell of the grid.

Tip about base class pointers. C++ normally refuses to convert one pointer type to another. But it will allow a conversion from a derived class pointer to a base class pointer (upcasting). This allows you to plot all kinds of GameObjects with a single function.

Tip about CIC++ arrays: If a is a three-dimensional array, then a[i][j][k] is the i, j, k element in the array, and a[i][j] is a pointer to a one-dimensional array starting a[i][j][0].

- `void draw();`
  - Outputs the grid array to produce the display like that shown in the sample output The size, scale, and origin are printed first, then each row and column, for the current size of the display. Note that the grid is plotted like a normal graph: larger x values are to the right, and larger y values are at the top. The x and y axes are labeled with values for every alternate column and row. Use the output stream formatting facilities to save the format settings, set them for neat output of the axis labels on the grid, and then restore them to their original settings.

Specifications: Allow two characters for each numeric value of the axis labels, with no decimal points. The axis labels will be out of alignment and rounded off if their values cannot be represented well in two characters. This distortion is acceptable in the name of simplicity for this project

Overall Structure of the Program (main.cpp)

(50 points for behavioral tests; update will be posted on Blackboard regarding this requirement and sample output)

The main program should include a loop that reads a command, executes it, and then asks for another command. You do NOT have to detect errors or bad input. PA4 will add these features.

The program will declare two Fish objects, three CoralReef object, and two Cave objects as outlined in the Model class description. These seven objects will persist throughout the execution of the program.

The program starts by displaying the current time and the status of each object using the show status function. The main function then asks for a command and the user inputs a single character for the command, and main calls a function for the appropriate command. The function for the command inputs requests any required additional information from the user, and then carries out the command. If the user entered the "go" or "run" commands, the program repeats the main loop by displaying the time and current status and prompting for a new command. Otherwise, it continues to prompt for new commands. This enables the user to command more than one object to move before starting the simulation running again.

Additional Specifications

You MUST:

- Make use of the classes and their members; you may not write non-object oriented code to do things that the classes can do.
- Declare function prototypes for the functions that are not part of a class, such as those called by main(), and list the function definitions after main. This will improve the readability of your program.
- Make .h and .cpp files for each of your class with data fields and member functions as specified above.
- **Make sure your code compiles!**

Programming Guideline

Unless you have so much experience that you shouldn't be in this course, trying to write this program all at once is the **hard** way to do it! Objected oriented programming is easy to do in chunks! That's the idea! The individual classes can be written and tested pretty much one at a time, and a piece at a time. Follow the checkpoints provided throughout this document. **Course staff will expect you to have completed these checkpoints before asking questions about future checkpoints.**

Makefile

Makefiles automate and facilitate the compilation process of software projects that consists of a large number of source code (`.cpp`) files (such as PA3). Instead of specifying each `.cpp` file in the compilation process (g++) of PA3, we will create makefiles to automate the compilation process and the achievement of the checkpoints.

In general, a makefile consists of multiple named targets ("rules") that can depend on each other. Every target has an associated action that is usually a UNIX command, such as `g++`. The structure of a named target looks as follows:

```
target_name : dependencies
	command_to_execute
```

IMPORTANT!

The `command_to_execute` line MUST be indented using a Tab space. This can be achieved by using the Tab key on your keyboard.

The UNIX `make` command interprets makefiles by executing the targets and their associated actions in the order of the targets' dependencies. Per default, the `make` command executes a makefile that is literally called `Makefile` (case-sensitive!). In case of naming your Makefile differently (e.g. Makefile_Checkpoint1), then you must use the –f option for the make command. Also check out the manual page of the make command (man make).

To demonstrate makefiles, we provide a makefile for Checkpoint I. Let's name the makefile Makefile_Checkpoint1.

```
# in EC327, we use the g++ compiler
# therefore, we define the GCC variable
GCC = g++

# a target to compile the Checkpoint1 which depends on all object-files
# and which links all object-files into an executable
Checkpoint1: TestCheckpoint1.o CartPoint.o CartVector.o
        $(GCC) TestCheckpoint1.o CartPoint.o CartVector.o -o Checkpoint1

# a target to compile the TestCheckpoint1.cpp into an object-file
TestCheckpoint1.o: TestCheckpoint1.cpp
        $(GCC) -c TestCheckpoint1.cpp

# a target to compile the CartPoint.cpp into an object-file
CartPoint.o: CartPoint.cpp
        $(GCC) -c CartPoint.cpp

# a target to compile the CartVector.cpp into an object-file
CartVector.o: CartVector.cpp
        $(GCC) -c CartVector.cpp

# a target to delete all object-files and executables
clean:
      rm TestCheckpoint1.o CartPoint.o CartVector.o Checkpoint1
```

As demonstrated, makefiles only support single-line comments that start with the '#' character. Furthermore, makefiles support the definition of variables, such as GCC, which get a specific value assigned. In our case, the GCC variable holds the value g++. You can access the value of the variables using the $() notation. For example, $(GCC) returns the value of the GCC variable, which is g++.

In the file `Makefile_Checkpoint1` we specify five targets (denoted in bold text). The `Checkpoint1` target depends on three targets, namely `TestCheckpoint1.o, CartPoint.o,` and `CartVector.o`. Each `*.o` target depends on the `.cpp` file, which should be compiled to an object file (using the `-c` option of the `g++` compiler). We define those dependencies in order to avoid re-compilation of `.cpp` files that have not been changed during the last compilation process. We also define one target `clean`, which deletes all object-files and the executable `Checkpoint1` (using the `rm` UNIX command).

Per default, the `make` command executes the first target, which is in our case `Checkpoint1`. You can, however, instruct what target to execute by passing the name of the target to the `make` command. For example, the following command will execute the `clean` target of the `Makefile_Checkpoint1` makefile.

```
make –f Makefile_Checkpoint1 clean
```

The EC327 staff will provide a final makefile for PA3. However, it is your task to formulate the makefiles for each checkpoint.

For further information on makefiles, please consult https://www.gnu.org/software/make/

Submission

Please use the file names **CartPoint.h, CartPoint.cpp, CartVector.h, CartVector.cpp, GameObject.h, GameObject.cpp, CoralReef.h, CoralReef.cpp, Cave.h, Cave.cpp, Fish.h, Fish.cpp, View.h, View.cpp, Model.h, Model.cpp, GameCommands.h, GameCommands.cpp and PA3.cpp**. Put all your cpp files **AND the makefile** (Makefile) in a folder named <your username>_PA3 (e.g., dougd_PA3), zip it, and submit a single file (e.g., dougd_PA3.zip) following the submission guidelines on Blackboard. Do **NOT** submit your executable files (a.out or others) or any other files in the folder. Make sure to **comment** your code.