CS162 - Programming Assignment 5 - Dynamic Memory Allocation, Object Pointers, Friend Functions, Enum Types

The purpose of this assignment is to give you some experience using pointers to objects, arrays of object pointers, friend functions, and enumerated types.

Overview

This program is part four of many that will implement an Asteroids-like game. Note: Asteroids is a registered trademark of Atari, Inc.

In this program you'll build upon the previous assignment.

- First, you'll add numerous asteroids to the screen using an array of asteroid pointers.
- Then you'll add collision detection and use it to determine when the ship collides with an asteroid.
- Then you'll add code to make the ship explode when it collides with an asteroid.

When the assignment's complete, you'll have one ship and many asteroids, and your ship will be able to crash into asteroids and explode. Oh the humanity!!!!

Part 1 - Bring on the Asteroids!

You currently have one (test) asteroid. Now we're going to add more. We could do this with an array of Asteroid objects, but that would make it difficult to remove or add them when the game's in play. (When we blast one, we might want to replace it with two, or simply remove it). So… we're going to create an array of pointers to asteroids, create a few asteroids to start, and leave a fair number of the array pointers null so we'll have the possibility of adding others (or deleting some) as the game goes on.

In the file that contains your main() routine:

- Define a constant integer constant named MAX_ASTEROIDS and set it equal to 20;
- In your main routine
    - comment out or delete all your references to the single asteroid.
- Near the top of main()
    - where you had created a single asteroid, add the code to declare an array of pointers to Asteroids and initialize them all to NULL (0);
    - Use a loop to dynamically create 5 new Asteroids (saving the pointers returned by new in the first 5 spots of the asteroids array)
    - Use a loop to position these asteroids randomly on the screen. You can randomly create x and y values between 0 and the max screen size and call setLocation() for each of them with those values from within main (or you could put this random-positioning code in the

asteroid constructor).
    - Randomly set their velocity (both x and y) to some reasonable values, and, ideally, their rotational-velocities.
- In main where you draw the ship (and previously drew the single test asteroid)
    - Add code that goes through the Asteroid array and calls draw() for each non-null entry.
    - You'll also probably need to call update-location so they move.

You should now have numerous asteroids flying around the screen in random directions and random speeds. Test it out and debug it as necessary.

Part 2 - Bring on the collisions!

We need to detect when the ship collides with an asteroid (and later when photons, the things the ship will be firing at the asteroids, collide with asteroids). Fortunately, since everything's derived from the SpaceObject class, this really all boils down to detecting when two SpaceObjects are intersecting. So we're going to add some code (<10 lines) to the SpaceObject source file to detect that.

The simplest solution will be to create a routine that can access the private data (location and radius) of two space objects at the same time. And this can be done using a friend function.

- In the SpaceObject.h file, add the following prototype in the SpaceObject class definition:

```
friend bool objectsIntersect(SpaceObject obj1, SpaceObject obj2);
```

- In the SpaceObject.cpp file, add the implementation of this function. Note: even though the prototype was in the class definition, this routine is NOT a member of the class (friends aren't members), so the function name is not prefaced with SpaceOject::. And even though it's a friend of the class, the word friend isn't in the implementation (it only went in the class definition). Friends are usually added at the end of the file.

```
bool objectsIntersect(SpaceObject obj1, SpaceObject obj2) {
      // Compute the distance between the two objects locations
      // If the distance is less than their combined radiuses,
      //    then they're intersecting.  Collison! Return true.
      // else return false.
}
```

- In the file with your main() routine, right before you draw the ship and asteroids, determine if the ship has run into one of them by writing a loop that goes through the Asteroid array and calls objectsIntersect for each non-null entry, passing it the asteroid and the ship (this works because both of those are derived from the spaceObject class). If true is returned, just print out "Crash!" on the console. We'll make the ship explode in the next step....

Each time your ship and an asteroid "collide" you should now be notified on the console. Test it out (crash into asteroids from all directions: top, left, right, bottom) and debug as necessary.

Part 3 - Bring on the explosion!

We now need to make the ship visually explode when it hits an asteroid.

To do this we'll first need to keep track of the "state" of the ship. Is it in good working order? Or exploding? Or gone? We need to know when we go to draw it.

Keeping track of the state of things is very common in programs. The optimal solution is to create a state variable as part of the ship class that can store the state of the ship . And to make this super-readable, we'll use an enumerated type to create a new variable type that can only hold certain values, values that clearly describe the ship's state (enums are covered in an early chapter, read/re-read it if necessary; but they're pretty simple so you might not need to).

In the ship.h file:

- At the top (above the Ship class definition), add the following statement:
  ```
  enum ShipState { SS_GOOD, SS_EXPLODING, SS_GONE };
  ```
  This creates a new datatype (ShipState) that can only have the values: GOOD, EXPLODING, GONE. (note: there are no quotes, the compiler turns these into constant names of ints)
- Add a private variable to the Ship class to store the ship's state
  ```
  ShipState state;
  ```
- Add a function prototype for a void function named explode(). (This will change the ship state from GOOD to EXPLODING)

In the ship.cpp file

- In the constructor, set the state to SS_GOOD.
- Add the implementation for the explode() function. It should change the ship state from SS_GOOD to SS_EXPLODING, and change its velocity to 0.
- In the control routines (rotateRight(), rotateLeft(), applyThrust() add if statements so if the ship's state is not SS_GOOD, nothing it done to the ship (Exploding and Gone ships aren't controllable!)
- Work on the Draw routine. We now need to draw the ship differently based on whether it's good or exploding, or not draw it at all if it's gone. Since drawing code is fairly long, it's best to create separate functions (which should be private) for drawing a good ship and drawing an exploding ship. So…
  - draw() should now call either drawGoodShip() or drawExplodingShip() based on the ship state (you'll need prototypes for these in the .h file as well, and they'll both need to get the SFML window object that was passed to draw().
  - drawGoodShip() will be just like draw() was previously.
  - drawExplodingShip() will be all new. One easy solution is to draw a solid circle that starts out small (size of radius?), and gets slightly bigger each time it's drawn. (add .1 to

the radius for each call?). When the size reaches some fixed size (30? 50?), change the ship state to GONE. Then it won't be drawn any more…. Game over.

In the source file containing main:

- Replace the line that printed "Crash" on the console with a call to the ship's explode() routine.

When your ship and asteroid "collide" the ship should now explode and be gone forever. Test it out and debug it as necessary.

Part 4 - Making it better (optional)

When creating asteroids, you could make sure they're a safe distance from the middle so the ship doesn't explode the moment it appears.

When the ship explodes, you could recreate it back in the middle when the area is clear of asteroids.

Very Important Stuff

All programs should follow the class's coding conventions.
Submit the following:

- A zip file containing all the source files you created and your executable