Recitation 9 - Classes

CSCI-1300
Instructor: Dr. David Knox
Fall 2017

What is an Object? What is a Class?

Forget programming for a second. If you think about the questions above, you'll realize how deceptively complex they are. At first, they seem straightforward. An object is… a "thing". It's an object in our world. At the same time, this means so much. For our purposes, let's define an object as any entity in the real world. Some examples include a person, a dog, a book, a hat, a boat, a state, a continent, a government, a planet, a galaxy, an atom, a poster, a wall, a building… you get the idea. Usually we define these objects as *instances* of a particular type of object or class of objects.

Now what's a class? That's a bit trickier. Basically, a class is the core representation of an object. Any given object is an instance of a *class.* To give an example, "Dog" is a class. Any given dog that you encounter is an object, an instance of a dog. Each object has specific features that distinguish it from other objects of the same class. Such as the type of dog or color, height, and weight attributes. Objects are a way to *encapsulate* information, meaning that a single variable contains all the needed data to describe something, rather than having to assign each piece of data to a different variable.

Now think about programming again. What does this mean for us as programmers? A programming "class" is simply the abstract representation of a real world object. So we'll really have the same kinds of things in programming (a person, a dog, a book, a state, a planet… you get the idea). However, in order to create a digital representation of a physical object, you have to break it down into what's important. In general, you have two different concepts to worry about.

1) *Attributes* - What does an object know/have? These are facts about your object. These can change (sometimes), but they are intended to hold information about your object. They are typically kept as variables within your class.

2) *Methods* - What does an object do? These are abilities of your object, or tasks it can complete. These are built into your object, but often act based on attributes. In C++, they are methods in your class.

Let's think about an example: a Dog. Yes, a dog is pretty complicated, but we're just making a representation of a dog, so it only needs to be as complicated as we need.

First, what *attributes* does a dog have? It could have a name, breed, height, weight, fur color, eye color, collar color, and many other attributes. However, for our purposes, the only one we'll worry about is a dog's name.

Second, what *methods* does a dog have? Dogs bark, eat, run, walk, fetch, sit, lay down, play dead, shake, and do many other tasks. However, for our purposes, the only one we'll worry about is a dog's ability to bark.

So, a dog *has* a name, and a dog *can* bark.

TIP: Look for words that highlight ownership (has, have, etc.) in a description to find *attributes*. Look for action verbs to find *methods*.

More on Classes and Objects in C++

Objects need a method to be created. This is a special method known as a *constructor*.

Classes are the templates for objects, they define the *attributes* and *methods* an object has. The particular values for *attributes* and the *methods* called are defined for each instance of the object.

Here is a simple class for a Dog, represented in C++. This simple class defines a Dog. Every dog has a name and every dog can bark.

-- Make sure to include the header file (Dog.h) in the implementation file (Dog.cpp)!

// Dog.h

```
#ifndef DOG_H
#define DOG_H
#include <iostream>
using namespace std;

class Dog
{
   public:
      Dog(string);          //constructor
      ~Dog();               //destructor
      void bark();
   protected:
   private:
      string name;
};

#endif // DOG_H
```

//Dog.cpp

```
#include "Dog.h"
#include <iostream>
using namespace std;

Dog::Dog(string dogName){
   //constructor
   name = dogName;
}

Dog::~Dog(){
   //destructor
}

void Dog::bark(){
   cout << "My dog's name is: " << name << endl;
}
```

//main.cpp

```
#include <iostream>
#include "Dog.h"
using namespace std;

int main(){
   Dog DogOne("Scruffy");
   DogOne.bark();

   return 0;
}
```

After build and run is selected this will produce:

My dog's name is: Scruffy

Notice a few things here. First, *Dog* is now a class (i.e. a valid type) and an object (i.e. variable) of this class called DogOne. C++ knows what a *Dog* is, because we defined it above. To call one of the *Dog* object's methods, simply type the variable name (indicating *which* object instance you want to bark) followed by a dot, followed by the method name.

Public vs. Private

Notice in the Dog class that the attribute *name* has the keyword *private* in its declaration. Often (though not always), an object doesn't want other objects or methods to be able to access these attributes directly. By applying the *private* keyword to an attribute or method, that attribute can *only* be accessed by the object the attribute belongs to, e.g. only methods in the Dog class can access private Dog attributes.

Recap:

- private members of a class are accessible only from within other members of the same class (or from their *"friends"*).
- public members are accessible from anywhere where the object is visible.

Getters and Setters

Typically, a class may provide functions to get or set the values of an attribute (called *getters* and *setters*). One reason for doing this is to ensure that the assignment is a valid one for the attribute. Let's modify the Dog class to have a getter and a setter for the variable.

-- For the header we have to add in the methods to the public section.

-- For the implementation file we have to add the getter and setter methods.

//Dog.h

```
#ifndef DOG_H
#define DOG_H
#include <iostream>
using namespace std;

class Dog{
    public:
        Dog(string);
        ~Dog();
        void bark();
        string getName();
        void setName(string);

    protected:
    private:
        string name;
};

#endif // DOG_H
```

//Dog.cpp

```
#include "Dog.h"
#include <iostream>
using namespace std;

Dog::Dog(string dogName){
    name = dogName;
}

Dog::~Dog(){
    //dtor
}

void Dog::bark(){
    cout << "My dog's name is: " << name << endl;
}

//getter -- gets the desired value
string Dog::getName(){
    return name;
}

//setter -- sets the value as requested
void Dog::setName(string newName){
    name = newName;
}
```

//main.cpp -- for this main, we still make the instance of the Dog class, but then get the method and print it to the terminal

```
#include <iostream>
#include "Dog.h"
using namespace std;

int main(){
    Dog DogOne("Scruffy");
    cout << DogOne.getName() << endl;

    DogOne.setName("Roscoe");
    cout << DogOne.getName() << endl;

    return 0;
}
```

After build and run is selected this will produce:

Scruffy
Roscoe

Scruffy is from the first “getter” and Roscoe is from the “setter” and printed with the getter.

Recitation Activity

For today’s recitation activity you are going to implement three separate classes.

Part 1: Battle Ship

The first class you will implement is called *battleShip*. The ***battleShip.h*** file has been made available on Moodle. You are also given the the beginning of a main file for Battle Ship: ***BattleShip_main.cpp***. Download these files and place them into the same folder on your computer. Open them with CodeBlocks. You will implement the battleShip class and fill out the main file for part 1 of the recitation.

You will create your class file ***battleShip.cpp***. In battleShip.cpp you will define the methods. (Remember: methods define what your object *does*). How many methods do you see in your header file? It will be your job to fill in these methods so that they work.The constructor always has the same name as your class file, and it never has a data type. Make sure to import the proper class file “battleShip.h” and libraries <string> etc. in the header of your ***battleShip.cpp*** file**.** See the BattleShip_main.cpp for further instructions. You will need to create 3 instances of your class, set up some of the methods, and expand on the logic of the while loop.

In the game of battleship, a ship is sunk once the number of hits it has received is equal to its size. So if a ship is size 3 and has received 3 hits, it has been sunk.

When you have finished, implementing these methods, show your TA to check off your work. Then move onto the next part.

Part 2: Robot

Go to the coding activity for this week and implement the Robot class. Unlike part 1, you will *not* have separate implementation and header files. Take a look at the Dog class implemented in a single file:

```
#include <iostream>
using namespace std;

class Dog{
   public:
          Dog(string dogName){
                 name = dogName;
          }
```

```
        ~Dog() {
                //destructor
        }

        void bark(){
                cout << "My Dog's name is " << name << endl;
        }

        string getName(){
                return name;
        }

        void setName(string newName){
                name = newName;
        }

    private:
        string name;

};
```

Part 3: Planet Class (optional)

Parts 3 and 4 of this recitation activity have a shared main file ***SolarSystem_main.cpp***, provided to you on Moodle.

The next class you will implement is called *planet*. This time we have given you the class file ***planet.cpp***. It is your job to create the header file for the planet class. Create the file ***planet.h*** in CodeBlocks in the same directory as the main file. The planet class has two constructors, a *default* constructor (which takes no arguments) and a constructor which takes the arguments:

string planetName,
float planetRadius (the radius of the planet, in km),
float planetDist (the distance the planet orbits from the sun, in Astronomical Units)

Once you have implemented the header file for the planet class, you will need to modify main to test your class. Use the table of planet data provided to you in the main file to create four planet objects of the Solar System. It will be helpful to use an array for this task, as the code to test your planets assumes an array of planet objects.

Part 4: Solar System (optional)

The last class you will implement is called *solarSystem*. The ***solarSystem.h*** file has been made available on Moodle. You will create your class file ***solarSystem.cpp***. There is one constructor, one destructor, and five methods to create for the solarSystem class. One of the setter methods is an *Add function.* Use this method to populate the solar system arrays, one planet at a time. Use the planet class getter methods to access the data needed for the addPlanet() method of solarSystem.

Code is provided in main to test the implementation of your solarSystem class. The Part 4 code is inside of a *comment block*. Uncomment the comment block to unlock the Part 4 main code. When you have

completed the implementation of the solarSystem class and included the solarSystem header file to the main file, the output of main should look like this:

```
The orbit of Mercury takes 0.24075 years.
The diameter of Mercury is 4880 km.
Mercury is 0.387 AU from the Sun.
 ---
The orbit of Venus takes 0.614763 years.
The diameter of Venus is 12104 km.
Venus is 0.723 AU from the Sun.
 ---
The orbit of Earth takes 1 years.
The diameter of Earth is 12742 km.
Earth is 1 AU from the Sun.
 ---
The orbit of Mars takes 1.88138 years.
The diameter of Mars is 6780 km.
Mars is 1.524 AU from the Sun.
 ---
System name: Sol
Sol has 4 planets.
The biggest planet is Earth.
The diameter of Sol is 3.048 AU.
```

When you are done, zip *all* of your files into one file called **Recitation9.zip** and submit to Moodle by Sunday at 5pm.