school of computing, informatics, & decision systems engineering

CSE 100, CIDSE, Fall B 2013
Lab 7

LAB 7 : Please Submit Your Program Under Lab 7 basket in the DropBox

Pre-Lab: Please read the pre-lab and answer the accompanying questions before your lab session.

Learning to use classes in C++ is a lot like learning to drive a stick-shift car: at first it can be annoying and painful, and it takes quite some time before you really feel comfortable doing it, but once you get used to it, it all makes sense and you'd gladly pick it over the alternative. Unfortunately, right now we're at the annoying and painful stage, so grit your teeth, keep on trying, and trust me when I say that in the end you'll be glad you learned how to use classes.

Classes are the basis, the cornerstone, and the core of ***object-oriented programming*** (OOP). Basically, OOP means structuring your code around things (cars, people, etc.) rather than tasks (driving, talking, etc.). Classes help accomplish this by allowing you to split up your code into pieces. Much like functions, classes allow you to write reusable code that has a specific application or purpose. What's fundamentally different (& cool) about classes is that they also allow you to store variables and data specific and unique to each instance of a class. An ***instance*** of a class is a unique instantiation of that class. Assuming that didn't help at all, basically you can think of this relation: classes are to instances as types are to variables. Actually, instances of classes work a lot like variables. You declare them, give them a name, and then can use them for input, output, function calls, and pretty much anything else you can do with variables. First, we'll go over the basics of writing classes, then briefly cover how to use them in a program.

Say we wanted to write a class that could describe a puppy. Fortunately, puppies are simple objects, so writing a class to describe one shouldn't be too hard. We'd start with something like this:

```
class Puppy
{

};
```

This code says that we're making a class called Puppy. Doesn't sound like much, but once we start putting things between those curly braces it'll get more interesting. Note that I called the class "Puppy", and not "Spot", "Princess", or any other specific name. Generally, convention says you're supposed to name classes after the category they represent, and save the special names for specific instances of the classes. Also, note that I capitalized the first letter of the class' name. That's another convention that helps distinguish variables and functions (which we've been writing with lowercase first letters) from classes. See, I told you there was a point to writing code a certain way!

Now, here comes a big, super-important part: check out that semi-colon at the end of the class declaration. ***Classes must have a semi-colon after their closing curly brace!*** This is one of the *only* times you'll see a semi-colon after a curly brace (remember, thus far we haven't done this for functions, loops, or any other block of code), but it has to be there, or else you will get a bunch of very cryptic compilation errors. Your best bet is to put it there as soon as you declare the class, so you won't forget to do it later.

Now that we've declared the class, it's time to put some things in it. For starters, puppies all have names, so let's give all instances of `Puppy` a name:

```
class Puppy
{
private:
	string name;
};
```

Now, every instance of `Puppy` will have its own name. Note that "`private:`" part. That's a special keyword that says, "From this point on, stuff declared in the class can only be accessed and changed by the class itself." This is an important part of classes, and one of the main reasons we use them in the first place. Making class data and functions private allows us to restrict how much the rest of the program can change, and how it is allowed to change and access that data. In our case, it means that we can control how the puppy gets its name, and if (& how) it can be changed and accessed. After all, you wouldn't want random people renaming your puppy, would you? In our case, we do want to be able to change and access the puppy's name, so we'll need to add a couple of functions:

```
class Puppy
{
public:
	string getName()
	{
		return name;
	}
	void setName(string newName)
	{
		name = newName;
	}
private:
	string name;
};
```

That "`public:`" part is `private`'s opposite. It says, "From here on, anything declared can be accessed and used by anything, whether or not it's part of the class." In our case, it allows us to call `getName` and `setName` from our main program (more on how to do that later). Let's check out `getName` and `setName` for a moment. These are a classic pair of functions that often come along with private class variables. `getName` is called a ***getter function***, and `setName` is called a ***setter function***. True to their names, `getName` "gets" the current value of name and returns it to whomever asked for it, and `setName` takes a new name as a parameter and assigns it to the private variable `name` accordingly. Getters and setters are very common and very useful, because they allow us to create a "middleman" between programs and private data, letting us carefully control how much access programs have to instance data.

In case you're wondering, class functions don't always have to be tied to class variables; they can (and often do) serve other purposes besides just accessing and changing class variables. For instance, anyone who has ever owned a puppy knows that they like to play a lot, so we should probably implement a function that lets our puppies play. That should make the `Puppy` class complete enough for our purposes.

```
class Puppy
{
public:
	void play()
	{
		cout << name << " romps around excitedly, barking happily and licking
everything in sight." << endl;
	}
	string getName()
	{
		return name;
	}
	void setName(string newName)
	{
		name = newName;
	}
private:
	string name;
};
```

There's one last thing we need to add to this class to make it ready for use. Classes, like variables, need to be initialized before they are used. However, they don't get initialized in the same way that variables do. Instead of setting them equal to some value, a special function is called in the class whose job it is to initialize the class and make sure everything is ready for use. This function is called the ***constructor***. It is only called once (when the instance of the class is declared), and really shouldn't do anything other than set default values of variables. Constructor function headers have a special syntax that sets them apart from other functions and signifies that they are constructors. Constructor functions do not have any return value at all (not even `void`), and their name is *always* the name of the class verbatim (same capitalization and everything). They can take any number of parameters, but a "typical" constructor doesn't take any parameters. Here's what our `Puppy` class looks like after adding constructors to it:

```
class Puppy
{
public:
	Puppy()
	{
		name = "";
	}
	Puppy(string initName)
	{
		name = initName;
	}
	void play()
	{
		cout << name << " romps around excitedly, barking happily and licking
everything in sight." << endl;
	}
	string getName()
	{
		return name;
	}
	void setName(string newName)
	{
		name = newName;
	}
private:
	string name;
};
```

Now there are two more functions in the class `Puppy`. The first one is called the ***default constructor***. This constructor is called if someone creates an instance of Puppy and doesn't pass any parameters to it (we'll see an example of that in the code below). Usually, a default constructor should just set all class variables to their equivalent of empty or zero (in this case, it sets `name` to `""`, meaning an empty string). The other constructor takes a string called `initName` as a parameter, and sets `name` equal to the value of `initName`. This way, we could create an instance of `Puppy` and give it a name right away, instead of having to wait to call `setName` (again, an example of this can be seen in the code below). Note that neither constructor returns anything; each just does their job and ends. Yes, a class can have multiple constructors, the only restriction is that the headers all have to be different (either by the types of the parameters, the number of parameters, or both).

Now that the `Puppy` class is done, it's time to use it in a program. Obviously, to use a class in a program, its code has to show up in the code file. Like any other code, the definition of the class has to appear before it is used. For now, your best bet is to implement all classes before your `main` function. There is a way to declare a "prototype" of sorts for classes, then implement them later (similar to how function prototypes work), but you don't have to worry about it at this point.

So, assuming we've already put all our include file declarations and the class Puppy into our code file, here's what code demonstrating the Puppy class might look like:

```
int main()
{
	Puppy myPuppy;
	Puppy yourPuppy("Reginald");

	myPuppy.setName("Sparky");
	myPuppy.play();
	cout << myPuppy.getName() << " likes to play with " << yourPuppy.getName() << "!"
<< endl;

	return 0;
}
```

This code would produce the following output:

```
Sparky romps around excitedly, barking happily and licking everything in sight.
Sparky likes to play with Reginald!
```

As you can see, we first declare an instance of `Puppy` called `myPuppy`. Even though we didn't put any parentheses after `myPuppy`, we still "secretly" call the default constructor for `Puppy` which sets `myPuppy`'s `name` to `""` by default (the line "`Puppy myPuppy();`" would do the same thing). Next, we give you a puppy (joy!), and by passing `"Reginald"` as a parameter, we call the special `Puppy` constructor that sets `yourPuppy`'s `name` to `"Reginald"`. Then, since `myPuppy` still doesn't have a name, we give it the name Sparky by calling the function `setName` and passing `"Sparky"` to it. This is important, so pay attention: ***when calling a class function, you have to call it using the syntax "`instanceName.classFunction(parameters, go, here);`"*** (i.e., you have to put the instance name and a period before the call to the function). Calling a class function without an instance name will make C++ mad at you, so don't do it.

The last part of the demonstration calls `myPuppy`'s `play` function (producing the first line of output), followed by a `cout` statement that uses both puppies' `getName` functions. Note that even though we're calling the same function, we get different outputs because we referenced different instances when we called `getName` each time. That's one of the coolest parts about classes (an instance's data remains separate from all other instances of that class), and that's why you need to preface any class function calls with an instance name. This concludes our introduction to classes, hope you enjoyed the puppies!

Lab 7
Classes

Lab Exercise: This portion of the lab should be completed during your lab session.

The exercise for this week is to write a class that simulates managing a simple bank account. The account is created with an initial balance. It is possible to deposit and withdraw funds, to add interest, and to find out the current balance. This should be implemented in class named `Account` that includes:

- A default constructor that sets the initial balance to zero.
- A constructor that accepts the initial balance as a parameter.
- A function `getBalance` that returns the current balance.
- A method `deposit` for depositing a specified amount.
- A method `withdraw` for withdrawing a specified amount.
- A method `addInterest` for adding interest to the account.

The `addInterest` method takes the interest rate as a parameter and changes the balance in the account to balance*(1+interestRate).

The UML diagram for the `Account` class is shown in figure 1.

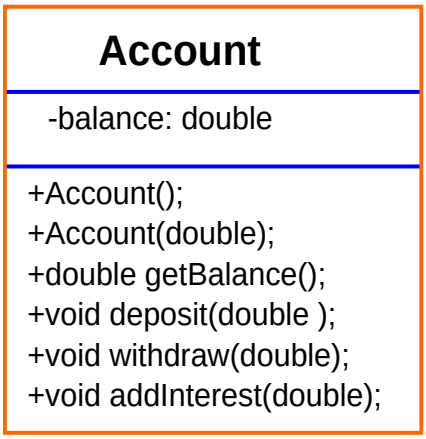

Figure 1: The Account Class

Your class must work with the code given below and display the output .

240
450

```
#include<iostream>
using namespace std;
#include "Account.h"
int main()
{
   Account  a1;
   Account a2(500);
   a1.depost(200);
   a2.withdraw(50);
   a1.addInterest(0.2);
cout<<a1.getBalance();
cout<<"\n";
cout<<a2.getBalance();
system("pause");
return 0;

}
```

Submissions:

Please Submit Your ***Account.h*** and ***Account.cpp*** Under Lab 7 basket in the DropBox