# *EECS 211 Lab 6*

## *Classes and Abstractation*

## *Winter 2017*

In this week's lab, we will be introducing classes, and reviewing encapsulation and abstraction.

If you have any lingering questions during the lab, don't hesitate to ask your peer mentor!

### *Getting the code*

Download the zip file from the course site:

http://users.eecs.northwestern.edu/~jesse/course/eecs211/lab/eecs211-lab06.zip

After you have downloaded the zip file onto your laptop, extract the zip file into its own folder. Make sure you keep track of which folder it's in! Next, open up CLion and Click on File –> Open Project, and click on the Lab 6 project that you just unzipped.

Once you open the project, try building the lab and then running the lab6 executable. You should see some output printed in your output subwindow. If you need a reminder on how to build and run code in CLion, consult lab 3/4 or ask your TA. Once this works, you're ready to start the lab!

### *Encapsulation and Abstraction*

Encapsulation is the process of binding associated data, and abstraction is the process of hiding details. You already have seen the idea of encapsulation and abstraction throughout the quarter. One place that you seen encapsulation is through structs. When you create a struct, you are creating a type that encapsulates data members. For example, the circle struct had x,y coordinates as well as a radius. One place that you've seen abstraction is through creating libraries, like linked_lib or circle_lib, which allow a consumer to use certain functions about circles or linked lists, without having to understand every little inner working under the hood.

### *Classes*

So that brings us to classes. Classes allow you to create your own types, just like structs. Additionally, with classes, we usually create member functions, which allow you to do things that you may have

already noticed from other classes, such as call myObject.area(); for instance. This is similar to a data member, as the functions would work on the instances of the classes themselves.

### *Private vs. Public*

One key difference between structs and classes is that struct data members and functions are public by default, but in classes, they are private by default. But what does that mean? Basically, public vs private determines what an instance of a class can access. First, examine a class with a definition something like this:

We usually refer to instances of classes as *objects*.

```
class Person
{
private:
    std::string name_;
    int ssn_;
    double bank_balance_;
public:
    Person(const std::string& n, int s);
    Person(const std::string& n, int s, double b);

    bool canIBuyThis(double itemCost) const;
    bool canIBuyThis(int itemCost) const;

    double withdrawFromBank(double amount);

    const std::string& name() const;
    int ssn() const;
    double bank_balance() const;
};
```

Notice here, that our Person class has a name_, ssn_, and bank_balance_ that are private. This means that if in our *main* or any other function creates an instance of a Person, it won't be able to access the Person's name_, ssn_ or bank_balance. For example, this code wouldn't work, as you can't access the ssn_ or bank_balance_.

```
int main()
{
    Person myPerson{"Wyatt", 102349783, 0};
    cout << myPerson.name_ << '\n';
    cout << myPerson.ssn_ << '\n';
}
```

However, what you CAN do, is create public functions that use the private data members or private functions. Let's say a Person is at

a store, and is trying to buy a soda pop. While you don't want the register to be able to see how much money is on the Person's bank account, you still want to be able to tell whether or not the Person has enough money to purchase the soda pop. So, you're able to run code something like this:

```
void buySoda(Person& p, double sodaPrice)
{
    if (p.canIBuyThis(sodaPrice)) {
        p.withdrawFromBank(sodaPrice);
    } else {
        cerr << "You don't have enough money!\n";
    }
}
```

Notice, that we are able to compare the Person's bank_balance_ and the sodaPrice, without being able to look at the Person's bank_balance_, through the use of our public *canIBuyThis* function. This allows us to abstract away the bank_balance_, while maintaining its functionality!

### *Constructors*

The constructor function for a given class does not have a return type and must have the same name as the class.

When you create an instance of a class, you use what's called a Constructor, which allows you to do something upon the creation of an object. Typically, we'll use a constructor to set the private data members of your class. For example, let's look at the definition of the constructor for Person.

```
Person::Person(const string& n, int s, double b)
        : name_(n), ssn_(s), bank_balance_(b)
{ }
```

Now, when you create a Person using the following syntax:

```
    Person myPerson("Jesse", 1234567, 100.0);
```

myPersons's name_, ssn_, and bank_balance_ are assigned to be Jesse, 1234567, and 100.0, respectively. Note that the following syntax would also be acceptable:

```
    Person myPerson = Person("Jesse", 1234567, 100.0);
```

You probably noticed that this constructor allows for 3 parameters. You can actually create different constructors for different numbers of parameters as inputs, allowing you to set default values easier, or do different things when a class is instantiated with a different number of arguments. In Person.cpp, we've defined another constructor for Person which only takes in 2 parameters:

```
Person::Person(const string& n, int s)
        : name_(n), ssn_(s), bank_balance_(0.0)
{ }
```

This constructor will automatically assign bank_balance_ to be 0.0. This is called ***overloading*** a constructor. When you call these constructors, C++ will call the correct constructor by looking at the number and type of the arguments it's given.

Another way to write a constructor is to have it *delegate* to another constructor:

```
Person::Person(const string& n, int s)
        : Person(n, s, 0.0)
{ }
```

In this case, the two-argument Person constructor calls the three-argument Person constructor, passing along its two arguments and and additional 0 for the balance.

### *Overloading Classes*

We can also overload functions! This works in the exact same way. In Person.cpp, note that *canIBuyThis* has 2 definitions. One of them takes in an int and the other takes in a double. Each of these definitions behaves in a slightly different way. Notice that both functions take in the same number of arguments but they are different types! C++ looks at the parameter supplied and decides which function to call based on the type of that parameter.

## *Practicing with classes*

To get a little practice with classes, implement the function *printPerson*. In order to do that, you'll need to implement the get functions for the Person class in Person.cpp. Once you've done that, use the get functions in the *printPerson* function in order to print a string with the following format:

The get functions are public functions that return the values of their respective private data members.

Hint: All 3 get functions can (and should) be implemented in one line.

"This Person is named (name_), has Social Security Number (ssn_), and has (bank_balance_) dollars in their bank account."

For more practice, go Bank.cpp and implement the function called *stealMoney*. Note that a Bank is another class that we've created for you. A bank has a private vector of Person objects called accounts_. It also has a public constructor and a get function for accounts_ already implemented. Your job is to write the *stealMoney* function, which should iterate through each Person in accounts_ and withdraw all of their money so that their bank_balance_ is 0. Add up

all of the money you've stolen and return that value. The functions *bank_balance* and *withdrawFromBank* in the Person class will be very helpful for this!