Laboratory 7: *Getting Classy*

Overview: The concept of a *class* facilitates the object-oriented features of C++. A class defines an object type, or abstraction, from which individual objects can be declared, or instantiated. A class encapsulates both the attributes (data members) and behavior (member functions) of a type of object.

Objectives: Before you leave lab today you should have a better understanding of:

1. how to declare a class in a header file (including how and why the access specifications "public" and "private" are used),

2. how to define a class in an implementation file, and

3. how to use a class to instantiate an object.

You should understand how classes use information hiding and *encapsulation*. You should understand what *data members* and *member functions* are, including *constructors*, *inspectors*, *mutators* and *facilitators*. You should understand how and when to use the member access operator (".").

In general, after this lab you should be well on your way to understanding the purpose of classes and how to put them to use appropriately.

1. Setting up for the lab

Please perform the following activities as you wait for the lab to formally begin:

a. Boot up the machine and log on to the CS Department server.

b. Put your lab diskette in the *A:* drive and make a subdirectory LAB07 in the *LABS* directory.

c. Download the files from the class Web site to your new directory. The URL is http://s9000.furman.edu/chealy/cs11/lab07.

2. The basic elements of classes

Objects are the fundamental unit of programming in object-oriented languages like C++. Objects are models of information and are implemented as software packages that contain or *encapsulate* both *attributes* and *behavior*. A representation of information and the operations to be performed on it is a *data abstraction*.

A fundamental-type object is an object that is provided by the programming language. For example, in C++ the type **int** and the operations on it are part of the language definition. That is, all C++ compilers must provide **int** objects. In addition to the fundamental types, C++ provides several mechanisms to define other types. These other types are called *programmer*- or *user-defined types.*

The **class** construct is the most important mechanism for defining new types. With this construct, software engineers can define encapsulated objects. When defining a class, we typically use the *information-hiding principle:*

> All interaction with an object should be constrained to use a well-defined interface that allows underlying object implementation details to be safely ignored. (i.e. Always use inspectors to view properties and always use mutators to set property values).

By following the information-hiding principle, classes tend to be more reliable and more easily reused.

To consider the basics of creating a new object type using the **class** construct, we first examine the declaration and definition of a programmer-defined type—the Account class.

a. Open the file *account.h* to see the class definition for **`account`**. Examine the code. Identify the different components of a class definition, including:

- *data members (attributes)*
  These are normally private to allow the enforcement of the information-hiding principle.

- *constructors*
  These have the same name as the class and and declare an object of the class. Note that there are 3 constructors (default, initial-value and copy). Be sure you can tell which one is which.

- *inspectors*
  These provide methods to access representations of the data members of an object. (They return the value of some attribute of an object.)
- *mutators*

These provide methods to modify the representation of the data members of an object. (They change, or mutate, an object.)

- *facilitators*
  These perform some action or service on or for an object.

b. Note that inspectors, mutators and facilitators are simply different kinds of class member functions. If you are unsure about any aspects of the definition of the `account` class, ask me before proceeding. You should understand all parts of the definition.

By convention, when defining a class, we give the **public** section first as the members in this section can be used by any function. The **protected** section (if it exists) is then given. The **private** section comes last.

3. Experimenting with classes

a. Open the project `bank.ide`. Then open the file `bank.cpp` from that project. Examine the code carefully.

Note that to access a member we use the *member access operator*, which is the period (.).

b. Build and run the project and examine the results.

c. Modify the program by defining a new `account` object called **betty.** Make a second call to the appropriate member function to also display the Betty's balance. Demonstrate your modified program. √

We will now test whether the access permissions associated with the `account` class are truly in effect. Rather than using the inspectors, we will attempt to directly access the data members.

d. Modify the implementation of the main program as follows. The object **jason** is of the `account` class, and so has the property `balance`. To get Jason's interest rate, assign a local variable with the value jason.rate. This is the syntax necessary to obtain the value of a public property.

e. Try to compile your modified file. Examine the compiler messages. Do you understand the error message you get from the compiler?

f. Modify `account.cpp` so that is implements an inspector get_rate() to provide the interest rate attribute of an object. Comment out the line of code in bank.cpp that caused the syntax error, and instead use the get_rate() member function to find out Jason's interest rate. √

Note that the names of the member functions are preceded by the class name and the scope resolution operator, which is a double colon (`::`). The member functions must be identified this way, otherwise the compiler would not be able to tell that they belonged to a class.

The first function defined in the file is a *constructor*. A constructor for a class is invoked when an object of that class is instantiated.

g. To verify that the `account` constructor is invoked when an account object is declared, add a statement in each of the three constructors to display the following message, as appropriate:

```
default constructor called
initial-value constructor called
copy constructor called
```

Execute the modified program. How many times was each constructor called in the program? Do you understand the flow of control? √

h. Remove (or comment out) the display statements you added to the constructors.

Keep in mind that a member function of a class, even a public member function, can access the private members of the class.

i. Add a member function `interest()` to the file `account.cpp`. This function adds the appropriate amount of interest to the account balance.

Note that none of the member functions in your `interest()` function need to use the selection operator (`.`). Why is that? Is `interest()` an inspector, mutator or facilitator? Should the qualifier **`const`** be used?

j. Modify the class definition of `account` in `account.h` to include the prototype of the member function `interest()`.

k. Modify the main program so that the account object **jim** has interest added after the initial call to get_balance`()`. Re-display its balance after the interest is added. Run the project `bank.ide` again. √

l. Close the `bank.ide` project. (Select *Close Project* from the **Project** menu.) Close Borland C++ and log out.

Post-Lab activity:

We did not have a pre-lab assignment for this lab, so I am replacing it with this post-lab question. Please turn in your response tomorrow at the beginning of class. Late submissions will not be accepted.

Look again at the withdraw() member function. It returns –1 if we try to withdraw more money than we have. Here –1 really represents an error condition rather than an amount of money. In the main program, when we try to take $60 from Judy's account, the output is misleading. Each time we call the withdraw() member function, we must test to see if the return value is –1. If it is, we should print an overdraft message; otherwise let the withdrawal proceed. Show on paper how you would modify the main program so that it uses the return value of withdraw() properly. In particular, in the case of taking $60 from Judy, it should print an error message and tell the user why there is a problem (i.e. tell the user how much is in the account so that it is clear there is an overdraft.) Note: you should not modify the account class implementation to accomplish this modification.