LESSON SET

13 Introduction to Classes

PURPOSE

1. To introduce object-oriented programming
2. To introduce the concept of classes
3. To introduce the concept of constructors and destructors
4. To introduce arrays of objects

PROCEDURE

1. Students should read Chapter 13 of the text.
2. Students should read the Pre-lab Reading Assignment before coming to lab.
3. Students should complete the Pre-lab Writing Assignment before coming to lab.
4. In the lab, students should complete labs assigned to them by the instructor.

Contents	Pre-requisites	Approximate completion time	Page number	Check when done
Pre-lab Reading Assignment	Chapter 13 of text	20 min.	244	
Pre-lab Writing Assignment	Pre-lab reading	10 min.	260	
LESSON 13A				
Lab 13.1 Square as a Class	Basic understanding of structures and classes	10 min.	261	
Lab 13.2 Circles as a Class	Completion of Pre-lab Reading Assignment	40 min.	263	
LESSON 13B				
Lab 13.3 Arrays as Data Members of Classes	Understanding of private data members of classes and files	20 min.	265	
Lab 13.4 Arrays of Objects	Understanding of classes	20 min.	267	
Lab 13.5 Student Generated Code Assignments	Completion of all the previous labs	30 min.	269	

PRE-LAB READING ASSIGNMENT

Introduction to Object-Oriented Programming

Up until now, we have been using the procedural programming method for writing all our programs. A procedural program has data stored in a collection of variables (or structures) and has a set of functions that perform certain operations. The functions and data are treated as separate entities. Although operational, this method has some serious drawbacks when applied to very large real-world situations. Even though procedural programs can be modularized (broken into several functions), in a large complex program the number of functions can become overwhelming and difficult to modify or extend. This can create a level of complexity that is difficult to understand.

Object-Oriented Programming (OOP) mimics real world applications by introducing **classes** which act as prototypes for **objects**. Objects are similar to nouns which can simulate persons, places, or things that exist in the real world. OOP enhances code reuse ability (use of existing code or classes) so time is not used on "reinventing the wheel."

Classes and objects are often confused with one another; however, there is a subtle but important difference explained by the following example. A plaster of Paris mold consists of the design of a particular figurine. When the plaster is poured into the mold and hardened, we have the creation of the figurine itself. A class is analogous to the mold, for it holds the definition of an object. The object is analogous to the figurine, for it is an **instance** of the class. Classes and structures are very similar in their construction. Object-oriented programming is not learned in one lesson. This lab gives a brief introduction into this most important concept of programming.

A **class** is a prototype (template) for a set of objects. An object can be described as a single instance of a class in much the same way that a variable is a single instance of a particular data type. Just as several figurines can be made from one mold, many objects can be created from the same class. A class consists of a name (its identity), its member data which describes what it is and its member functions which describe what it does.[1] Member data are analogous to nouns since they act as entities. Member functions are analogous to verbs in that they describe actions. A class is an **abstract data type (ADT)** which is a user defined data type that combines a collection of variables and operations. For example, a rectangle, in order to be defined, must have a length and width. In practical terms we describe these as its member data (`length, width`). We also describe a set of member functions that gives and returns values to and from the member data as well as perform certain actions such as finding the rectangle's perimeter and area. Since many objects can be created from the same class, each object must have its own set of member data.

As noted earlier, a class is similar to a structure except that classes encapsulate (contain) functions as well as data.[2] Functions and data items are usually designated

[1] In other object-oriented languages member functions are called methods and member data are called attributes.

[2] Although structures can contain functions, they usually do not, whereas classes always contain them

as either **private** or **public** which indicates what can access them. Data and functions that are defined as `public` can be directly accessed by code outside the class, while functions and data defined as `private` can be accessed only by functions belonging to the class. Usually, classes make data members `private` and require outside access to them through member functions that are defined as `public`. Member functions are thus usually defined as `public` and member data as `private`.

The following example shows how a rectangle class can be defined in C++:

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


// Class declaration (header file)

class Rectangle  // Rectangle is the name of the class (its identity).
{

public:

   //  The following are labeled as public.
   //  Usually member functions are defined public
   //  and are used to describe what the class can do.

   void setLength(float side_l);
   //  This member function receives the length of the
   //  Rectangle object that calls it and places that value in
   //  the member data called length.

   void setWidth(float side_w);
   //  This member function receives the width of the Rectangle
   //  object that calls it and places the value in the member
   //  data called width.

   float getLength();
   //  This member function returns the length of the Rectangle
   //  object that calls it.

   float getWidth();
   //  This member function returns the width of the Rectangle
   //  object that calls it.

   double findArea();
   //  This member function finds the area of the Rectangle object
   //  that calls it.

   double findPerimeter();
   //  This member function finds the perimeter of the Rectangle
   //  object that calls it.
```

continues

```
private:

    //  The following are labeled as private.
    //  Member data are usually declared private so they can
    //  ONLY be accessed by functions that belong to the class.
    //  Member data describe the attributes of the class

    float length;
    float width;

};
```

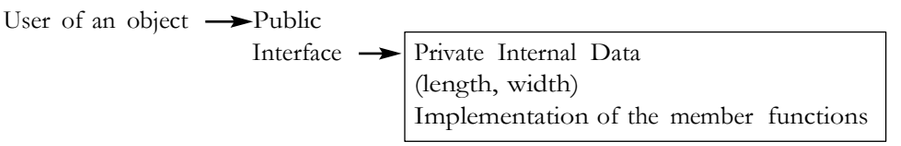
This example has six member functions. It has two member functions for each private member data: `setLength` and `getLength` for the member data `length` and `setWidth` and `getWidth` for the member data `width`. It is often the case that a class will have both a **set** and a **get** member function for each of its private data members. A set member function receives a value from the calling object and places that value into the corresponding private member data. A get member function returns the value of the corresponding private member data to the object that calls it. In addition to set and get member functions, classes usually have other member functions that perform certain actions such as finding area and perimeter in the `Rectangle` class.

Client and Implementation Files

It is not necessary for someone to understand how a television remote control works in order to use the remote to change the stations or the volume. The user of the remote could be called a **client** that only knows how to use the remote to accomplish a certain task. The details of how the remote control performs the task are not necessary for the user to operate the remote. Likewise, an automobile is a complex mechanical machine with a simple interface that allows users without any (or very little) mechanical knowledge to start, drive, and use it for a variety of functions. Drivers do not need to know what goes on under the hood. In the same way, a program that uses `Rectangle` does not need to know the details of how its member functions perform their operations. The use of an object (an instance of a class) is thus separated into two parts: the **interface** (client file) which calls the functions and the **implementation** which contains the details of how the functions accomplish their task.

An object not only combines data and functions, but also restricts other parts of the program from accessing member data and the inner workings of member functions. Having programs or users access only certain parts of an object is called **data hiding**. The fact that the internal data and inner workings can be hidden from users makes the object more accessible to a greater number of programs.

Just like an automobile or a remote control, a piece of commercial software is usually a complex entity developed by many individuals. OOP (Object-Oriented Programming) allows programmers to create objects with hidden complex logic that have simple **interfaces** which are easily understood and used. This allows more sophisticated programs to be developed. Interfacing is a major concern for software developers.



Types of Objects

Objects are either general purpose or application-specific. General purpose objects are designed to create a specific data type such as currency or date. They are also designed to perform common tasks such as input verification and graphical output. Application-specific objects are created as a specific limited operation for some organization or task. A student class, for example, may be created for an educational institution.

Implementations of Classes in C++

The class declaration is usually placed in the global section of a program or in a special file (called a **header** file). As noted earlier, the class declaration acts very much like a prototype or data type for an object. An object is defined much like a variable except that it uses the class name as the data type. This definition creates an **instance** (actual occurrence) of the class. Implementation of the member functions of a class are given either after the `main` function of the program or in a separate file called the **implementation** file. Use of the object is usually in the `main` function, other specialized functions, or in a separate program file called the **client** file.[3]

Creation and Use of Objects

`Rectangle`, previously described, is a class (prototype) and not an object (an actual instance of the class). Objects are defined in the client file, `main`, or other functions just as variables are defined:

```
Rectangle box1,box2;
```

`box1` and `box2` are objects of class `Rectangle`.

`box1` has its own `length` and `width` that are possibly different from the `length` and `width` of `box2`.

To access a member function (method) of an object, we use the dot operator, just as we do to access data members of structures. The name of the object is given first, followed by the dot operator and then the name of the member function.

The following example shows a complete `main` function (or client file) that defines and uses objects which call member functions.

```
int main()

{
        Rectangle box1;      // box1 is defined as an object of Rectangle class
        Rectangle box2;      // box2 is defined as another Rectangle class object
```

[3] More will be given on header, implementation, and client files later in the lesson.

```
    box1.setLength(20); // This instruction has the object box1 calling the
                        // setLength member function which sets the member data
                        // length associated with box1 to the value of 20
    box1.setWidth(5);

    box2.setLength(9.5);// This instruction has the object box2 calling the
                        // setLength member function which sets the member data
                        // length associated with box2 to the value of 9.5
    box2.setWidth(8.5);

    cout << "The length of box1 is "  << box1.getLength() << endl;
    cout << "The width of box1 is " << box1.getWidth() << endl;
    cout << "The area of box1 is "  << box1.findArea() << endl;
    cout << "The perimeter of box1 is " << box1.findPerimeter() << endl;

    cout << "The length of box2 is "  << box2.getLength() << endl;
    cout << "The width of box2 is " << box2.getWidth() << endl;
    cout << "The area of box2 is "  << box2.findArea() << endl;
    cout << "The perimeter of box2 is " << box2.findPerimeter() << endl;

    return 0;

}
```

Since `findArea` and `findPerimeter` must have `length` and `width` before they can do the calculation, an object must call `setLength` and `setWidth` first. The user must remember to initialize both `length` and `width` by calling both set functions. It is not good programming practice to assume that a user will do the necessary initialization. Constructors (discussed later) solve this problem.

Implementation of Member Functions

As previously noted, the implementation of the member function can be hidden from the users (clients) of the objects. However, they must be implemented by someone, somewhere. The following shows the implementation of the `Rectangle` member functions.

```
//*****************************************************************
//                                    setLength
//
//  task:     This member function of the class Rectangle receives
//            the length of the Rectangle object that calls it and
//            places that value in the member data called length.
//  data in:  the length of the rectangle
//  data out: none
//
//*****************************************************************
```

```
void Rectangle::setLength(float side_l)
{

   length = side_l;
}

//*********************************************************************
//                                      setWidth
//
//  task:     This member function of the class Rectangle receives the
//            the width of the Rectangle object that calls it and
//            places that value in the member data called width.
//  data in:  the width of the rectangle
//  data out: none
//
//*********************************************************************

void Rectangle::setWidth(float side_w)
{

   width = side_w;
}


//*********************************************************************
//                                      getLength
//
//  task:     This member function of the class Rectangle returns
//            the length of the Rectangle object that calls it.
//  data in:  none
//  data returned: length
//
//*********************************************************************

float Rectangle::getLength()

{
   return length;

}

//*********************************************************************
//                                      getWidth
//
//  task:     This member function of the class Rectangle returns
//            the width of the Rectangle object that calls it.
//  data in:         none
//  data returned:   width
//
//*********************************************************************
```

continues

```
float Rectangle:: getWidth()
{
   return width;
}

//*****************************************************************
//                                      findArea
//
//  task:            This member function of the class Rectangle
//                   calculates the area of the object that calls it.
//  data in:         none (uses the values of member data length &
//                   width)
//  data returned:   area
//
//*****************************************************************


double Rectangle::findArea()

{
   return length * width;
}


//*****************************************************************
//                                      findPerimeter
//
//  task:            This member function of the class Rectangle
//                   calculates the perimeter of the object that calls it
//  data in:         none  (uses the values of member data length &
//                          width)
//  data returned:   perimeter
//
//*****************************************************************

double Rectangle::findPerimeter()
{

   return ((2 * length) + (2 * width));
}
```

Notice that in the heading of each member function the name of the function is preceded by the name of the class to which it is a member followed by a double colon. In the above example each name is preceded by **`Rectangle::`**. This is necessary to indicate in which class the function is a member. There can be more than one function with the same name associated with different classes. The `::` symbol is called the **scope operator**. It acts as an indicator of the class association.

Usually classes are declared in a header file, while member functions are stored in an implementation file and objects are defined and used in a client file. These files are often bound together in a project. Various development environments have different means of creating and storing related files in a project. All could be located in three different sections of the same file.

Complete Program

The following code shows the class declaration, member functions (methods), implementations and use (client) of the `Rectangle` class:

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

//_____________________________________________________________________
// Class declaration (header file)

class Rectangle  // Rectangle is the name of the class
{
public:

   //  The member functions are labeled as public.

   void setLength(float side_l);
   //  This member function receives the length of the
   //  Rectangle object that calls it and places that value in
   //  the member data called length.

   void setWidth(float side_w);
   //  This member function receives the width of the Rectangle
   //  object that calls it and places the value in the member
   //  data called width.

   float getLength();
   //  This member function returns the length of the Rectangle
   //  object that calls it.

   float getWidth();
   //  This member function returns the width of the Rectangle
   //  object that calls it.

   double findArea();
   //  This member function finds the area of the rectangle object
   //  that calls it.

   double findPerimeter();
   //  This member function finds the perimeter of the rectangle
   //  object that calls it.


private:

   //  The following are labeled as private.
   //  Member data are usually declared private so they can
   //  ONLY be accessed by functions that belong to the class.
   //  Member data describe the attributes of the class
```

continues

```
    float length;
    float width;

};

// _____________________________________________________________________
// Client file
int main()

{
    Rectangle box1;         // box1 is defined as an object of Rectangle class
    Rectangle box2;         // box2 is defined as another Rectangle class object

    box1.setLength(20);     // This instruction has the object box1 calling the
                            // setLength member function which sets the member
                            // data length associated with box1 to the value
                            // of 20
    box1.setWidth(5);

    box2.setLength(30.5);   // This instruction has the object box2 calling the
                            // setLength member function which sets the member
                            // data length associated with box2 to the value
                            // of 30.5

    box2.setWidth(8.5);

    cout << "The length of box1 is "  << box1.getLength() << endl;
    cout << "The width of box1 is " << box1.getWidth() << endl;
    cout << "The area of box1 is "  << box1.findArea() << endl;
    cout << "The perimeter of box1 is " << box1.findPerimeter() << endl;

    cout << "The length of box2 is "  << box2.getLength() << endl;
    cout << "The width of box2 is " << box2.getWidth() << endl;
    cout << "The area of box2 is "  << box2.findArea() << endl;
    cout << "The perimeter of box2 is " << box2.findPerimeter() << endl;

    return 0;

}
// _____________________________________________________________________
// Implementation file

//*********************************************************************
//                                      setLength
//
//  task:     This member function of the class Rectangle receives the
//            the length of the Rectangle object that calls it and
//            places that value in the member data called length.
//  data in:  the length of the rectangle
//  data out: none
//
//*********************************************************************
```

```
void Rectangle::setLength(float side_l)
{

   length = side_l;
}

//*********************************************************************
//                                      setWidth
//
//  task:     This member function of the class Rectangle receives the
//            the width of the Rectangle object that calls it and
//            places that value in the member data called width.
//  data in:  the width of the rectangle
//  data out: none
//
//*********************************************************************

void Rectangle::setWidth(float side_w)
{

   width = side_w;
}

//*********************************************************************
//                                      getLength
//
//  task:            This member function of the class Rectangle returns
//                   the length of the Rectangle object that calls it.
//  data in:         none
//  data returned:   length
//
//*********************************************************************

float Rectangle::getLength()

{
   return length;

}

//*********************************************************************
//                                      getWidth
//
//  task:            This member function of the class Rectangle returns
//                   the width of the Rectangle object that calls it.
//  data in:         none
//  data returned:   width
//
//*********************************************************************
```

continues

```
float Rectangle::getWidth()
{
   return width;
}

//*********************************************************************
//                                      findArea
//
//  task:            This member function of the class Rectangle
//                   calculates the area of the object that calls it.
//  data in:         none  (uses the values of member data length &
//                   width)
//  data returned:   area
//
//*********************************************************************


double Rectangle::findArea()

{
   return length * width;
}


//*********************************************************************
//                                      findPerimeter
//
//  task:            This member function of the class Rectangle
//                   calculates the perimeter of the object that calls it.
//  data in:         none  (uses the values of member data length & width)
//  data returned:   perimeter
//
//*********************************************************************

double Rectangle::findPerimeter()
{

   return ((2 * length) + (2 * width));
}
```

Inline Member Functions

Sometimes the implementation of member functions is so simple that they can be defined inside a class declaration. Such functions are called inline member functions. In the `Rectangle` class, `findArea` and `findPerimeter` are so simple that they can be defined in the class declaration as follows:

```
double findArea(){ return length * width; }
double findPerimeter()  { return 2 * length + 2 * width; }
```

Introduction to Constructors

As noted earlier, the methods (member functions) `findArea` and `findPerimeter` must have the `length` and `width` before they can do any calculation. The user must remember to initialize both `length` and `width` by calling both of these functions. What happens if the user forgets? Suppose we call `findArea` without first calling both `setLength` and `setWidth`. The function will try to find the area of a rectangle that has no length or width. Thus, the creator of a class should never rely on the user to initialize essential data.

C++ provides a mechanism, called a **constructor**, to guarantee the initialization of an object. A constructor is a member function that is *implicitly* invoked whenever a class instance is created (whenever an object is defined). A constructor is unique from other member functions in two ways:

1. It has the same name as the class itself.
2. It does not have a data type (or the word void) in front of it. The only purpose of the constructor is to initialize an object's member data.

The following shows the `Rectangle` class using two constructors that set the values of `length` and `width`.

```
class Rectangle
{
public:
   Rectangle(float side_l, float side_w);
   // Constructor allowing a user to input the length and width
   Rectangle();
   // Constructor using default values for both length and width

   void setLength(float side_l);
   void setWidth(float side_w);
   float getLength();
   float getWidth();
   double findArea();
   double findPerimeter() ;
private:
   float length;
   float width;

};
```

This class includes two constructors, differentiated by their parameter lists. Recall from Lesson Set 6.2 that two or more functions can have the same name as long as their parameters differ in quantity or data type. The parameter-less constructor (the second constructor in the above example) is the **default constructor**. Like all member functions, constructors are defined in the implementation file (or function definition section of a program). The reason for a default constructor is explained in the next section.

Constructor Definitions

The function definitions of the two constructors for the `Rectangle` class are as follows:

```
Rectangle::Rectangle(float side_l, float side_w)
{
    length = side_l;
    width = side_w;
}

Rectangle::Rectangle()
{
    length = 1;
    width = 1;
}
```

The first constructor allows the user to input a value for both `length` and `width` at the same time that the object is defined (shown later in the lab). The second constructor (the default constructor) sets both `length` and `width` to 1 whenever the object is defined. Actually they could be set to anything that the creator of the class wants to use as a default for an object of the class that is not initialized by the user. With the use of these constructors, every object of class `Rectangle` will have a value for both `length` and `width`. We still keep the two member functions `setLength` and `setWidth` to allow the user to change the values of `length` and `width`. We could create a third constructor that has just one parameter which gives the value of `length` and uses the default value for `width`. If we create this third constructor, however, we can not create a fourth constructor that gives the value of `width` and use the default value for `length`. Why? We would have two member functions with the same name and an identical parameter list in both data type and number.

Invoking a Constructor

Although a constructor is a member function, it is never invoked (called) using the dot notation. It is invoked when an object is defined.

Example:
```
Rectangle box1(12,6);
Rectangle box2;
```

In this example, `box1` is an object of `Rectangle` class that has `length` set to 12 and `width` set to 6. Since it has two parameters, `box1` activates the constructor that has two parameters. The object `box2` is defined with both `length` and `width` set to 1. Since `box2` has no parameters, it activates the default constructor.

Destructors

A **destructor** is a member function that is automatically called to destroy an object. Just like constructors, a destructor has the same name as the class; however, it is preceded by a tilde (~). Destructors are used to free up memory when the object is no longer needed. The destructor is automatically called when an object of the class goes out of scope. This occurs when the function (such as `main`), where the object is defined, ends. The following example shows how constructors and destructors operate.

Example:

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

class Demo
{
public:
    Demo();  // Default constructor
    ~Demo(); // Destructor

};

int main()
{
    Demo demoObj;  // demoObj is defined and invokes
                   // the default constructor that
                   // prints the message "The constructor has
                   // been invoked"

    cout << "The program is now running" << endl;
    return 0;
}

// Now that the main program is over, the object demoObj is no
// longer active.  The destructor is invoked and the message
// "The destructor has been invoked" is printed.

//*********************************************************
//                    The Default Constructor Demo
// Notice that constructors do not have to set member data
// This constructor prints a message that the constructor
// has been invoked.
//*********************************************************

Demo::Demo()
{
    cout << "The constructor has been invoked"  << endl;
}

//*********************************************************
//                   The Destructor Demo
// Notice that destructors do not have to print anything but
// this destructor prints the message "The destructor has been
// invoked."  The primary purpose of destructors is to free
// memory space once an object is no longer needed.
//*********************************************************

Demo::~Demo()
{
    cout << "The destructor has been invoked"  << endl;
}
```

What order do you think the three `cout` statements will be executed? Note that a class can have only one default constructor and one destructor.

Arrays of Objects

Arrays can also contain objects of a class. For example, we could have an array of `Rectangle` objects.

Example:

```
Rectangle box[4]; // box is defined as an array of Rectangle objects
```

This statement makes an array of 4 elements, each consisting of an object of the `Rectangle` class.

Since this class has a default constructor, the default values are assigned to each element (object) of the array. The length and width for each of the objects in the box array are equal to 1 since these are the default values assigned by the default constructor.

The following program demonstrates the use of an array of objects:

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

class Rectangle
{
public:

   // Constructor allowing a user to input the length and width
    Rectangle(float side_l, float side_w);
    Rectangle();      // Default constructor
    ~Rectangle();     // Destructor

    void setLength(float side_l);
    void setWidth(float side_w);
    float getLength();
    float getWidth();
    double findArea();
    double findPerimeter() ;

private:

    float length;
    float width;

};

const int NUMBEROFOBJECTS = 4;

int main()

{
```

```
    Rectangle box[NUMBEROFOBJECTS];  // Box is defined as an array of
                                     // Rectangle objects

    for (int pos = 0; pos < NUMBEROFOBJECTS; pos++)

    {
       cout << "Information for box number  " << pos + 1 << endl << endl;

       cout << "The length of the box is "  << box[pos].getLength()
            << endl;
       cout << "The width of the box is " << box[pos].getWidth() << endl;
       cout << "The area of the box is "  << box[pos].findArea() << endl;
       cout << "The perimeter of the box is " << box[pos].findPerimeter()
            << endl << endl;

    }

    return 0;

}

void Rectangle::setLength(float side_l)
{
    length = side_l;
}

void Rectangle::setWidth(float side_w)
{
    width = side_w;
}


float Rectangle::getLength()
{
    return length;
}

float Rectangle:: getWidth()
{
    return width;
}

double Rectangle::findArea()
{
    return length * width;
}

double Rectangle::findPerimeter()
{
```

continues

```
    return ((2 * length) + (2 * width));
}

Rectangle::Rectangle(float side_l, float side_w)
{
    length = side_l;
    width = side_w;
}

Rectangle::Rectangle()
{
    length = 1;
    width = 1;
}

Rectangle::~Rectangle()
{
}
```

The output will be the same for each `box` because each has been initialized to the default values for `length` and `width`.

PRE-LAB WRITING ASSIGNMENT

Fill–in-the-Blank Questions

1. A(n)_____is used in C++ to guarantee the initialization of a class instance.
2. A constructor has the_____name as the class itself.
3. Member functions are sometimes called _____in other object-oriented languages.
4. A(n) _____is a member function that is automatically called to destroy an object.
5. To access a particular member function, the code must list the object name and the name of the function separated from each other by a _____.
6. A _____constructor has no parameters.
7. A _____precedes the destructor name in the declaration.
8. A(n) _____member function has its implementation given in the class declaration.
9. In an array of objects, if the default constructor is invoked, then it is applied to _____object in the array.
10. A constructor is a member function that is _____invoked whenever a class instance is created.