LESSON SET

11 Structures and Abstract Data Types

PURPOSE

1. To introduce the concept of an abstract data type
2. To introduce the concept of a structure
3. To develop and manipulate an array of structures
4. To use structures as parameters
5. To use hierarchical (nested) structures

PROCEDURE

1. Students should read the Pre-lab Reading Assignment before coming to lab.
2. Students should complete the Pre-lab Writing Assignment before coming to lab.
3. 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		20 min.	196	
Pre-lab Writing Assignment	Pre-lab reading	10 min.	205	
LESSON 11 A				
Lab 11.1 Working with Basic Structures	Knowledge of previous chapters	15 min.	205	
Lab 11.2 Initializing Structures	Basic understanding of structures	15 min.	206	
Lab 11.3 Arrays of Structures	Basic understanding of arrays and structures	20 min.	208	
LESSON 11 B				
Lab 11.4 Nested Structures	Basic understanding of functions and nested logic	20 min.	209	
Lab 11.5 Student Generated Code Assignments	Completion of all the previous labs	30 min.	211	

PRE-LAB READING ASSIGNMENT

So far we have learned of data types such as `float`, `int`, `char`, etc. In some applications the programmer needs to create their own data type. A user defined data type is often an **abstract data type (ADT)**. The programmer must decide which values are valid for the data type and which operations may be performed on the data type. It may even be necessary for the programmer to design new operations to be applied to the data. We will study this style of programming extensively when we introduce *object-oriented programming* in the lesson set from Chapter 13.

As an example, suppose you want to create a program to simulate a calendar. The program may contain the following ADTs: `year`, `month`, and `day`. Note that `month` could take on values January, February, . . . , December or even 1,2, . . . ,12 depending on the wishes of the programmer. Likewise, the range of values for `day` could be Monday, Tuesday, . . . , Sunday or even 1,2, . . . ,7. There is much more flexibility in the choice of allowable values for `year`. If the programmer is thinking short term they may wish to restrict `year` to the range 1990–2010. Of course there are many other possibilities.

In this lab we study the **structure**. Like arrays, structures allow the programmer to group data together. However, unlike an array, structures allow you to group together items of *different* data types. To see how this could be useful in practice, consider what a student must do to register for a college course. Typically, one obtains the current list of available courses and then selects the desired course or courses. The following is an example of a course you may choose:

```
CHEM 310          Physical Chemistry          4 Credits
```

Note that there are four items related to this course: the course discipline (CHEM), the course number (310), the course title (Physical Chemistry), and the number of credit hours (4). We could define variables as follows:

Variable Definition	Information Held
`string discipline`	4-letter abbreviation for discipline
`int courseNumber`	Integer valued course number
`string courseTitle`	First 20 characters of course title
`short credits`	Number of credit hours

All of these variables are related because they can hold information about the same course. We can package these together by creating a structure. Here is the declaration:

```
struct course
{
     string discipline;
     int    courseNumber;
     string courseTitle;
     short  credits;
};   //note the semi-colon here
```

The **tag** is the name of the structure, `course` in this case. The tag is used like a data type name. Inside the braces we have the variable declarations that are the **members** of the structure. So the code above declares a structure named `course` which has four members: `discipline`, `courseNumber`, `courseTitle`, and `credits`.

The programmer needs to realize that the structure declaration does not define a variable. Rather it lets the compiler know what a `course` structure is composed of. That is, the declaration creates a new data type called `course`. We can now define variables of type `course` as follows:

```
course pChem;
course colonialHist;
```

Both `pChem` and `colonialHist` will contain the four members previously listed. We could have also defined these two structure variables on a single line:

```
course pChem, colonialHist;
```

Both `pChem` and `colonialHist` are called **instances** of the `course` structure. In other words, they are both user defined variables that exist in computer memory. Each structure variable contains the four structure members.

Access to Structure Members

Certainly the programmer will need to assign the members values and also keep track of what values the members have. C++ allows you to access structure members using the **dot operator**. Consider the following syntax:

```
colonialHist.credits = 3;
```

In this statement the integer 3 is assigned to the `credits` member of `colonialHist`. The dot operator is used to connect the member name to the structure variable it belongs to.

Now let us put all of these ideas together into a program. Sample Program 11.1 below uses the `course` structure just described. This interactive program allows a student to add requested courses and keeps track of the number of credit hours for which they have enrolled. The execution is controlled by a `do-while` loop.

Sample Program 11.1:

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

// This program demonstrates the use of structures

// structure declaration

struct course
{
       string discipline;
       int courseNumber;
       string courseTitle;
       short credits;
};
```

continues

```
int main()
{
      course nextClass;  // next class is a course structure
      int numCredits = 0;
      char addClass;
      do
      {
         cout << "Please enter course discipline area:  ";
         cin >> nextClass.discipline;
         cout << endl << "Pleae enter the course number: ";
         cin >> nextClass.courseNumber;
         cout << endl << "Please enter the course title: ";
         cin.ignore();  // necessary for the next line
         getline(cin, nextClass.courseTitle);
         // use getline because course title may have a blank space
         cout << "Please enter the number of credit hours: ";
         cin >> nextClass.credits;

         numCredits = numCredits + nextClass.credits;

         // output the selected course and pertinent information

         cout << "You have been registered for the following: " << endl;
         cout << nextClass.discipline << "    " << nextClass.courseNumber
              << "    " << nextClass.courseTitle
              << "    " << nextClass.credits << "credits" << endl;

         cout << "Would you like to add another class? (Y/N)" << endl;
         cin >> addClass;

      } while(toupper(addClass) == 'Y');

      cout << "The total number of credit hours registered for is: "
           << numCredits << endl;

      return 0;
}
```

Make sure that you understand the logic of this program and, in particular, how structures are used. Notice the line at the end of the `while` loop that reads

```
while(toupper(addclass) == 'Y');
```

What do you think the purpose of `toupper` is?

As a second example, suppose we would like a simple program that computes the area and circumference of two circles input by the user. Although we can easily do this using previously developed techniques, let us see how this can be done using structures. We will also determine which circle's center is further from the origin.

Sample Program 11.2:

```
#include <iostream>
#include <cmath>    // necessary for pow function
#include <iomanip>
using namespace std;


struct circle       // declares the structure circle
{                   // This structure has 6 members
     float centerX; // x coordinate of center
     float centerY; // y coordinate of center
     float radius;
     float area;
     float circumference;
     float distance_from_origin;
};

const float PI = 3.14159;

int main()
{
            circle circ1, circ2; // defines 2 circle structure variables

            cout << "Please enter the radius of the first circle: ";
            cin >> circ1.radius;
            cout << endl
                 << "Please enter the x-coordinate of the center: ";
            cin >> circ1.centerX;
            cout << endl
                 << "Please enter the y-coordinate of the center: ";
            cin >> circ1.centerY;

            circ1.area = PI * pow(circ1.radius, 2.0);
            circ1.circumference = 2 * PI * circ1.radius;
            circ1.distance_from_origin = sqrt(pow(circ1.centerX,2.0)
               + pow(circ1.centerY,2.0));
            cout << endl << endl;

            cout << "Please enter the radius of the second circle: ";
            cin >> circ2.radius;
            cout << endl
                 << "Please enter the x-coordinate of the center: ";
            cin >> circ2.centerX;
            cout << endl
                 << "Please enter the y-coordinate of the center: ";
            cin >> circ2.centerY;

            circ2.area = PI * pow(circ2.radius, 2.0);
            circ2.circumference = 2 * PI * circ2.radius;
            circ2.distance_from_origin = sqrt(pow(circ2.centerX,2.0)
               + pow(circ2.centerY,2.0));
```

continues

```
    cout << endl << endl;

    // This next section determines which circle's center is
    // closer to the origin

    if (circ1.distance_from_origin > circ2.distance_from_origin)
    {
       cout << "The first circle is further from the origin"
            << endl << endl;
    }
    else if (circ1.distance_from_origin < circ2.distance_from_origin)
    {
       cout << "The first circle is closer to the origin"
            << endl << endl;
    }
    else
       cout << "The two circles are equidistant from the origin";
    cout << endl << endl;

    cout << setprecision(2) << fixed << showpoint;

    cout << "The area of the first circle is : ";
    cout << circ1.area << endl;
    cout << "The circumference of the first circle is: ";
    cout << circ1.circumference << endl << endl;

    cout << "The area of the second circle is : ";
    cout << circ2.area << endl;
    cout << "The circumference of the second circle is: ";
    cout << circ2.circumference << endl << endl;

    return 0;
}
```

Arrays of Structures

In the previous sample program we were interested in two instances of the `circle` structure. What if we need a much larger number, say 100, instances of this structure? Rather than define each one separately, we can use an **array of structures**. An array of structures is defined just like any other array. For example suppose we already have the following structure declaration in our program:

```
struct circle
{
     float centerX;    // x coordinate of center
     float centerY;    // y coordinate of center
     float radius;
     float area;
     float circumference;
     float distance_from_origin;  // distance of center from origin
};
```

Then the following statement defines an array, `circn`, which has 100 elements. Each of these elements is a `circle` structure variable:

```
circle circn[100];
```

Like the arrays encountered in previous lessons, you can access an array element using its subscript. So `circn[0]` is the first structure in the array, `circn[1]` is the second, and so on. The last structure in the array is `circn[99]`. To access a member of one of these array elements, we still use the dot operator. For instance, `circn[9].circumference` gives the `circumference` member of `circn[9]`. If we want to display the center and distance from the origin of the first 30 cir- cles we can use the following:

```
for (int count = 0; count < 30; count++)
{
     cout << circn[count].centerX << endl;
     cout << circn[count].centerY << endl;
     cout << circn[count].distance_from_origin;
}
```

When studying arrays you may have seen two-dimensional arrays which allow one to have "a collection of collections" of data. An array of structures allows one to do the same thing. However, we have already noted that structures permit you to group together items of different data type, whereas arrays do not. So an array of structures can sometimes be used when a two-dimensional array cannot.

Initializing Structures

We have already seen numerous examples of initializing variables and arrays at the time of their definition in the previous labs. Members of structures can also be initialized when they are defined. Suppose we have the following structure declaration in our program:

```
struct course
{
     string  discipline;
     int courseNumber;
     string courseTitle;
     short credits;
};
```

A structure variable `colonialHist` can be defined and initialized:

```
course colonialHist = {"HIST",302,"Colonial History",3};
```

The values in this list are assigned to `course`'s members in the order they appear. Thus, the string `"HIST"` is assigned to `colonialHist.discipline`, the integer `302` is assigned to `colonialHist.courseNumber`, the string `"Colonial History"` is assigned to `colonialHist.courseTitle`, and the short value `3` is assigned to `colonialHist.credits`. It is not necessary to initialize all the members of a structure variable. For example, we could initialize just the first member:

```
course colonialHist = {"HIST"};
```

This statement leaves the last three members uninitialized. We could also initialize only the first two members:

```
course colonialHist = {"HIST",302};
```

There is one important rule, however, when initializing structure members. If one structure member is left uninitialized, then all structure members that follow it must be uninitialized. In other words, we cannot skip members of a structure during the initialization process.

It is also worth pointing out that you cannot initialize a structure member in the declaration of the structure. The following is an illegal declaration:

```
// illegal structure declaration
struct course
{
     string discipline = "HIST";                 // illegal
     int courseNumber = 302;                     // illegal
     string courseTitle = "Colonial History";    // illegal
     short credits = 3;                          // illegal
};
```

If we recall what a structure declaration does, it is clear why the above code is illegal. A structure declaration simply lets the compiler know what a structure is composed of. That is, the declaration creates a new data type (called `course` in this case). So the structure declaration does not define any variables. Hence there is nothing that can be initialized there.

Hierarchical (Nested) Structures

Often it is useful to nest one structure inside of another structure. Consider the following:

Sample Program 11.3:

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

struct center_struct
{
       float x;    // x coordinate of center
       float y;    // y coordinate of center
};

struct circle
{
       float radius;
       float area;
       float circumference;
       center_struct coordinate;
};

const float PI = 3.14159;

int main()
{
       circle circ1, circ2; // defines 2 circle structure variables
```

```
    cout << "Please enter the radius of the first circle: ";
    cin >> circ1.radius;
    cout << endl
         << "Please enter the x-coordinate of the center: ";
    cin >> circ1.coordinate.x;
    cout << endl
         << "Please enter the y-coordinate of the center: ";
    cin >> circ1.coordinate.y;

    circ1.area = PI * pow(circ1.radius, 2.0);
    circ1.circumference = 2 * PI * circ1.radius;

    cout << endl << endl;

    cout << "Please enter the radius of the second circle: ";
    cin >> circ2.radius;
    cout << endl
         << "Please enter the x-coordinate of the center: ";
    cin >> circ2.coordinate.x;
    cout << endl
         << "Please enter the y-coordinate of the center: ";
    cin >> circ2.coordinate.y;

    circ2.area = PI * pow(circ2.radius, 2.0);
    circ2.circumference = 2 * PI * circ2.radius;

    cout << endl << endl;

    cout << setprecision(2) << fixed << showpoint;

    cout << "The area of the first circle is : ";
    cout << circ1.area << endl;
    cout << "The circumference of the first circle is: ";
    cout << circ1.circumference << endl;
    cout << "Circle 1 is centered at (" << circ1.coordinate.x
         << "," << circ1.coordinate.y << ")." << endl << endl;

    cout << "The area of the second circle is : ";
    cout << circ2.area << endl;
    cout << "The circumference of the second circle is: ";
    cout << circ2.circumference << endl ;
    cout << "Circle 2 is centered at (" << circ2.coordinate.x
         << "," << circ2.coordinate.y << ")." << endl << endl;

    return 0;
}
```

Note that the programs in this lesson so far have not been modularized. Everything is done within the `main` function. In practice, this is not good structured programming. It can lead to unreadable and overly repetitious code. To solve this problem, we need to be able to pass structures and structure members to functions. In this next section, you will see how to do this.

Structures and Functions

Just as we can use other variables as function arguments, structure members may be used as function arguments. Consider the following structure declaration:

```
struct circle
{
     float centerX;        // x coordinate of center
     float centerY;        // y coordinate of center
     float radius;
     float area;
};
```

Suppose we also have the following function definition in the same program:

```
float computeArea(float r)
{
      return PI * r * r;  // PI must previously be defined as a
                          // constant float
}
```

Let `firstCircle` be a variable of the `circle` structure type. The following function call passes `firstCircle.radius` into `r`. The return value is stored in `firstCircle.area`:

```
firstCircle.area = computeArea(firstCircle.radius);
```

It is also possible to pass an entire structure variable into a function rather than an individual member.

```
struct course
{
      string discipline;
      int courseNumber;
      string courseTitle;
      short credits;
};

course pChem;
```

Suppose the following function definition uses a `course` structure variable as its parameter:

```
void displayInfo(course c)
{
     cout << c.discipline << endl;
     cout << c.courseNumber << endl;
     cout << c.courseTitle << endl;
     cout << c.credits << endl;
}
```

Then the following call passes the `pChem` variable into `c`:

```
displayInfo(pChem);
```

There are many other topics relating to functions and structures such as returning a structure from a function and pointers to structures. Although we do not have time to develop these concepts in this lab, the text does contain detailed coverage of these topics for the interested programmer.

PRE-LAB WRITING ASSIGNMENT

Fill-in-the-Blank Questions

1. The name of a structure is called the _____.
2. An advantage of structures over arrays is that structures allow one to use items of _____data types, whereas arrays do not.
3. One structure inside of another structure is an example of a _____.
4. The variables declared inside the structure declaration are called the _____of the structure.
5. When initializing structure members, if one structure member is left uninitialized, then all the structure members that follow must be _____.
6. A user defined data type is often an _____.
7. Once an array of structures has been defined, you can access an array element using its _____.
8. The _____allows the programmer to access structure members.
9. You may not initialize a structure member in the _____.
10. Like variables, structure members may be used as _____ arguments.

LESSON 11 A

LAB 11.1 Working with Basic Structures

Bring in program `rect_struct.cpp` from the Lab 11 folder. The code is shown below.

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


// This program uses a structure to hold data about a rectangle
// PLACE YOUR NAME HERE

// Fill in code to declare a structure named rectangle which has
// members length, width, area, and perimeter all of which are floats

int main()
{
    // Fill in code to define a rectangle variable named box

    cout << "Enter the length of a rectangle: ";

    // Fill in code to read in the length member of box

    cout << "Enter the width of a rectangle: ";
```

continues

```
    // Fill in code to read in the width member of box

    cout << endl << endl;

    // Fill in code to compute the area member of box
    // Fill in code to compute the perimeter member of box

    cout << fixed << showpoint << setprecision(2);

    // Fill in code to output the area with an appropriate message
    // Fill in code to output the perimeter with an appropriate message

    return 0;
}
```

Exercise 1: Fill in the code as indicated by the comments in bold.

Exercise 2: Add code to the program above so that the modified program will determine whether or not the rectangle entered by the user is a square.

Sample Run:

```
Enter the length of a rectangle: 7
Enter the width of a rectangle: 7
The area of the rectangle is 49.00
The perimeter of the rectangle is 28.00
The rectangle is a square.
```

LAB 11.2 Initializing Structures

Bring in program `init_struct.cpp` from the Lab 11 folder. The code is shown below.

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


// This program demonstrates partially initialized structure variables

// PLACE YOUR NAME HERE

struct taxPayer
{
      string name;
      long  socialSecNum;
      float taxRate;
      float income;
      float taxes;
};

int main()
{
```

```
   // Fill in code to initialize a structure variable named citizen1 so that
   // the first three members are initialized.  Assume the name is Tim
   // McGuiness, the social security number is 255871234, and the tax rate is .35

   // Fill in code to initialize a structure variable named citizen2 so that
   // the first three members are initialized.  Assume the name is John Kane,
   // the social security number is 278990582, and the tax rate is .29

   cout << fixed << showpoint << setprecision(2);

   // calculate taxes due for citizen1

   // Fill in code to prompt the user to enter this year's income for the citizen1
   // Fill in code to read in this income to the appropriate structure member

   // Fill in code to determine this year's taxes for citizen1

   cout << "Name: " << citizen1.name << endl;
   cout << "Social Security Number: " << citizen1.socialSecNum << endl;

   cout << "Taxes due for this year: $" << citizen1.taxes << endl << endl;

   // calculate taxes due for citizen2

   // Fill in code to prompt the user to enter this year's income for citizen2
   // Fill in code to read in this income to the appropriate structure member

   // Fill in code to determine this year's taxes for citizen2

   cout << "Name: " << citizen2.name << endl;
   cout << "Social Security Number: " << citizen2.socialSecNum << endl;

   cout << "Taxes due for this year: $" << citizen2.taxes << endl << endl;

   return 0;
}
```

Exercise 1: Fill in the code as indicated by the comments in bold.

Sample Run:

```
Please input the yearly income for Tim McGuiness: 30000
Name: Tim McGuiness
Social Security Number: 255871234
Taxes due for this year: $10500.00

Please input the yearly income for John Kane: 60000
Name: John Kane
Social Security Number: 278990582
Taxes due for this year: $17400.00
```

LAB 11.3 Arrays of Structures

Bring in program `array_struct.cpp` from the Lab 11 folder. The code is shown below.

```
#include <iostream>
#include <iomanip>

using namespace std;


// This program demonstrates how to use an array of structures
// PLACE YOUR NAME HERE

// Fill in code to declare a structure called taxPayer that has three
// members:  taxRate, income, and taxes — each of type float


int main()
{
      // Fill in code to define an array named citizen which holds
      // 5 taxPayers structures

      cout << fixed << showpoint << setprecision(2);

      cout << "Please enter the annual income and tax rate for 5 tax payers: ";
      cout << endl << endl << endl;

      for(int count = 0;count < 5;count++)
      {

         cout << "Enter this year's income for tax payer " << (count + 1);
         cout << ": ";

         // Fill in code to read in the income to the appropriate place

         cout << "Enter the tax rate for tax payer # " << (count + 1);
         cout << ": ";

         // Fill in code to read in the tax rate to the appropriate place

         // Fill in code to compute the taxes for the citizen and store it
         // in the appropriate place


         cout << endl;
      }
```

```
      cout << "Taxes due for this year: " << endl << endl;

      // Fill in code for the first line of a loop that will output the
      // tax information
      {
            cout << "Tax Payer # " << (index + 1) << ": " << "$ "
                 << citizen[index].taxes << endl;
      }

      return 0;
}
```

Exercise 1: Fill in the code as indicated by the comments in bold.

Exercise 2: In the previous code we have the following:

```
cout << "Tax Payer # " << (index+1) << ": " << "$ "
     << citizen[index].taxes << endl;
```

Why do you think we need `(index+1)` in the first line but `index` in the second?

Sample Run:

```
Enter this year's income for tax payer 1: 45000
Enter the tax rate for tax payer # 1: .19
Enter this year's income for tax payer 2: 60000
Enter the tax rate for tax payer # 2: .23
Enter this year's income for tax payer 3: 12000
Enter the tax rate for tax payer # 3: .01
Enter this year's income for tax payer 4: 104000
Enter the tax rate for tax payer # 4: .30
Enter this year's income for tax payer 5: 50000
Enter the tax rate for tax payer # 5: .22

Tax Payer # 1: $ 8550.00
Tax Payer # 2: $ 13800.00
Tax Payer # 3: $ 120.00
Tax Payer # 4: $ 31200.00
Tax Payer # 5: $ 11000.00
```

LESSON 11 B

LAB 11.4 Nested Structures

Bring in program `nestedRect_struct.cpp` from the Lab 11 folder. This code is very similar to the rectangle program from Lab 11.1. However, this time you will complete the code using nested structures. The code is shown below.

```
#include <iostream>
#include <iomanip>

using namespace std;
```

continues

```
// This program uses a structure to hold data about a rectangle
// It calculates the area and perimeter of a box

// PLACE YOUR NAME HERE

// Fill in code to declare a structure named dimensions that
// contains 2 float members, length and width

// Fill in code to declare a structure named rectangle that contains
// 3 members, area, perimeter, and sizes. area and perimeter should be
// floats, whereas sizes should be a dimensions structure variable

int main()
{
   // Fill in code to define a rectangle structure variable named box.

   cout << "Enter the length of a rectangle: ";

   // Fill in code to read in the length to the appropriate location

   cout << "Enter the width of a rectangle: ";

   // Fill in code to read in the width to the appropriate location

   cout << endl << endl;

   // Fill in code to compute the area and store it in the appropriate
   // location
   // Fill in code to compute the perimeter and store it in the
   // appropriate location

   cout << fixed << showpoint << setprecision(2);
   cout << "The area of the rectangle is " << box.attributes.area << endl;
   cout << "The perimeter of the rectangle is " <<  box.attributes.perimeter
        << endl;

   return 0;
}
```

Exercise 1: Fill in the code as indicated by the comments in bold.

Exercise 2: Modify the program above by adding a third structure named `results` which has two members `area` and `perimeter`. Adjust the `rectangle` structure so that both of its members are structure variables.

Exercise 3: Modify the program above by adding functions that compute the area and perimeter. The structure variables should be passed as arguments to the functions.

Sample Run:

```
Enter the length of a rectangle: 9
Enter the width of a rectangle: 6
The area of the rectangle is 54.00
The perimeter of the rectangle is 30.00
```

LAB 11.5 Student Generated Code Assignments

Option 1: Re-write Sample Program 11.2 so that it works for an array of structures. Write the program so that it compares 6 circles. You will need to come up with a new way of determining which circle's center is closest to the origin.

Option 2: Write a program that uses a structure to store the following information for a particular month at the local airport:

Total number of planes that landed
Total number of planes that departed
Greatest number of planes that landed in a given day that month
Least number of planes that landed in a given day that month

The program should have an array of twelve structures to hold travel information for the entire year. The program should prompt the user to enter data for each month. Once all data is entered, the program should calculate and output the average monthly number of landing planes, the average monthly number of departing planes, the total number of landing and departing planes for the year, and the greatest and least number of planes that landed on any one day (and which month it occurred in).

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;

};
```

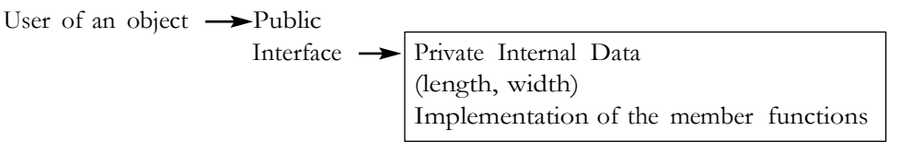
This example has six member functions. It has two member functions for each private member data: `setLength` and `getLength` for the member data `length` and `setWidth` and `getWidth` for the member data `width`. It is often the case that a class will have both a **set** and a **get** member function 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.

LESSON 13A

Lab 13.1 Squares as a Class

Retrieve program `square.cpp` from the Lab 13 folder. The code is as follows:

```
// This program declares the Square class and uses member functions to find
// the perimeter and area of the square

#include <iostream>
using namespace std;

// FILL IN THE CODE TO DECLARE A CLASS CALLED Square. TO DO THIS SEE
// THE IMPLEMENTATION SECTION.

int main()
{
    Square  box;   // box is defined as an object of the Square class
    float   size;  // size contains the length of a side of the square

   // FILL IN THE CLIENT CODE THAT WILL ASK THE USER FOR THE LENGTH OF THE
   // SIDE OF THE SQUARE. (This is stored in size)

   // FILL IN THE CODE THAT CALLS SetSide.

   // FILL IN THE CODE THAT WILL RETURN THE AREA FROM A CALL TO A FUNCTION
   // AND PRINT OUT THE AREA TO THE SCREEN.

   // FILL IN THE CODE THAT WILL RETURN THE PERIMETER FROM A CALL TO A
   // FUNCTION AND PRINT OUT THAT VALUE TO THE SCREEN.

   return 0;
}

//_____________________________________________________________________
//Implementation section     Member function implementation

//************************************************
//                  setSide
//
// task:    This procedure takes the length of a side and
//          places it in the appropriate member data
// data in: length of a side
//************************************************

void Square::setSide(float length)

{
   side = length;
}
```

continues

```
//************************************************
//                 findArea
//
// task:    This finds the area of a square
// data in: none (uses value of data member side)
// data returned:  area of square
//************************************************

float Square::findArea()
{
   return side * side;
}
//************************************************
//                 findPerimeter
//
// task:    This finds the perimeter of a square
// data in: none (uses value of data member side)
// data returned:  perimeter of square
//************************************************
float Square::findPerimeter()
{
   return 4 * side;
}
```

Exercise 1: This program asks you to fill in the class declaration and client code based on the implementation of the member functions. Fill in the code so that the following input and output will be generated:

Please input the length of the side of the square
8
The area of the square is 64
The perimeter of the square is 32

Exercise 2: Add two constructors and a destructor to the class and create the implementation of each. One constructor is the default constructor that sets the side to 1. The other constructor will allow the user to initialize the side at the definition of the object. The destructor does not have to do anything but reclaim memory space. Create an object called `box1` that gives the value of 9 to the constructor at the definition. Add output statements so that the following is printed in addition to what is printed in Exercise 1.

The area of box1 is 81
The perimeter of box1 is 36

Lab 13.2 `Circles` as a Class

Retrieve program `circles.cpp` from the Lab 13 folder. The code is as follows:

```
#include <iostream>
using namespace std;
//_____________________________________________________________________
// This program declares a class for a circle that will have
// member functions that set the center, find the area, find
// the circumference and display these attributes.
// The program as written does not allow the user to input data, but
// rather has the radii and center coordinates of the circles
// (spheres in the program) initialized at definition or set by a function.

//class declaration section   (header file)

class Circles
{
public:
   void setCenter(int x, int y);
   double findArea();
   double findCircumference();
   void printCircleStats(); // This outputs the radius and center of the circle.
   Circles (float r);       // Constructor
   Circles();               // Default constructor
private:
   float  radius;
   int    center_x;
   int    center_y;
};

const double PI = 3.14;

//Client section

int main()
{
   Circles sphere(8);
   sphere.setCenter(9,10);
   sphere.printCircleStats();

   cout << "The area of the circle is " << sphere.findArea() << endl;
   cout << "The circumference of the circle is "
        << sphere.findCircumference() << endl;

   return 0;
}

//_____________________________________________________________________
```

continues

```
//Implementation section     Member function implementation

Circles::Circles()
{
   radius = 1;
}
// Fill in the code to implement the non-default constructor

// Fill in the code to implement the findArea member function

// Fill in the code to implement the findCircumference member function

void Circles::printCircleStats()
// This procedure prints out the radius and center coordinates of the circle
// object that calls it.

{
   cout << "The radius of the circle is " << radius << endl;
   cout << "The center of the circle is (" << center_x
        << "'" << center_y << ")" << endl;
}

void Circles::setCenter(int x, int y)
// This procedure will take the coordinates of the center of the circle from
// the user and place them in the appropriate member data.

{
   center_x = x;
   center_y = y;
}
```

Exercise 1: Alter the code so that setting the center of the circle is also done during the object definition. This means that the constructors will also take care of this initialization. Make the default center at point (0, 0) and keep the default radius as 1. Have `sphere` defined with initial values of 8 for the radius and (9, 10) for the center. How does this affect existing functions and code in the `main` function?

The following output should be produced:

The radius of the circle is 8
The center of the circle is (9, 10)
The area of the circle is 200.96
The circumference of the circle if 50.24

Exercise 2: There can be several constructors as long as they differ in number of parameters or data type. Alter the program so that the user can enter either just the radius, the radius and the center, or nothing at the time the object is defined. Whatever the user does NOT include (radius or center) must be initialized somewhere. There is no `setRadius` function and there will no longer be a `setCenter` function. You can continue to assume that the default radius is 1 and the default center is (0, 0). Alter the client portion (`main`) of the program by defining an object `sphere1`, giving just

the radius of 2 and the default center, and `sphere2` by giving neither the radius nor the center (it uses all the default values). Be sure to print out the vital statistics for these new objects (area and circumference).

In addition to the output in Exercise 1, the following output should be included:

The radius of the circle is 2
The center of the circle is (0, 0)
The area of the circle is 12.56
The circumference of the circle is 12.56

The radius of the circle is 1
The center of the circle is (0, 0)
The area of the circle is 3.14
The circumference of the circle is 6.28

Exercise 3: Alter the program you generated in Exercise 2 so that the user will be allowed to enter either nothing, just the radius, just the center, or both the center and radius at the time the object is defined. Add to the client portion of the code an object called `sphere3` that, when defined, will have the center at (15, 16) and the default radius. Be sure to print out this new object's vital statistics (area and circumference).

In addition to the output in Exercise 1 and 2, the following output should be printed:

The radius of the circle is 1
The center of the circle is (15, 16)
The area of the circle is 3.14
The circumference of the circle is 6.28

Exercise 4: Add a destructor to the code. It should print the message **This concludes the Circles class** for each object that is destroyed. How many times is this printed? Why?

LESSON 13B

Lab 13.3 Arrays as Data Members of Classes

Retrieve program `floatarray.cpp` and `temperatures.txt` from the Lab 13 folder. The code is as follows:

```
// This program reads floating point data from a data file and places those
// values into the private data member called values (a floating point array)
// of the FloatList class. Those values are then printed to the screen.
// The input is done by a member function called GetList. The output
// is done by a member function called PrintList. The amount of data read in
// is stored in the private data member called length. The member function
// GetList is called first so that length can be initialized to zero.

#include <iostream>
#include <fstream>
#include <iomanip>
```

continues

```
using namespace std;

const int MAX_LENGTH = 50;  // MAX_LENGTH contains the maximum length of our list
class FloatList             // Declares a class that contains an array of
                            // floating point numbers
{
public:
   void getList(ifstream&);   // Member function that gets data from a file
   void printList() const;    // Member function that prints data from that
                              // file to the screen.
   FloatList();               // constructor that sets length to 0.
   ~FloatList();              // destructor

private:
   int length;                // Holds the number of elements in the array
   float values[MAX_LENGTH];  // The array of values

};

int main()
{
   ifstream tempData;         // Defines a data file

   // Fill in the code to define an object called list of the class FloatList

   cout << fixed << showpoint;
   cout << setprecision(2);

   tempData.open("temperatures.txt");

   // Fill in the code that calls the getList function.
   // Fill in the code that calls the printList function.

return 0;
}
 FloatList::FloatList()
 {
   // Fill in the code to complete this constructor that
   // sets the private data member length to 0
 }

// Fill in the entire code for the getList function
// The getList function reads the data values from a data file
// into the values array of the class FloatList

// Fill in the entire code for the printList function
// The printList function prints to the screen the data in
// the values array of the class FloatList

// Fill in the code for the implementation of the destructor
```

This program has an array of floating point numbers as a private data member of a class. The data file contains floating point temperatures which are read by a member function of the class and stored in the array.

Exercise 1: Why does the member function `printList` have a `const` after its name but `getList` does not?

Exercise 2: Fill in the code so that the program reads in the data values from the temperature file and prints them to the screen with the following output:

78.90
87.40
60.80
70.40
75.60

Exercise 3: Add code (member function, call and function implementation) to print the average of the numbers to the screen so that the output will look like the output from Exercise 2 plus the following:

The average temperature is 74.62

Lab 13.4 Arrays of Objects

Retrieve program `inventory.cpp` and `inventory.dat` from the Lab 13 folder. The code is as follows:

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

// This program declares a class called Inventory that has itemnNumber (which
// contains the id number of a product) and numOfItem (which contains the
// quantity on hand of the corresponding product)as private data members.
// The program will read these values from a file and store them in an
// array of objects (of type Inventory).  It will then print these values
// to the screen.

// Example: Given the following data file:
//     986 8
//     432 24
// This program reads these values into an array of objects and prints the
// following:
//     Item number 986 has 8 items in stock
//     Item number 432 has 24 items in stock


const NUMOFPROD = 10;   // This holds the number of products a store sells

class Inventory
{
public:
```

continues

```
   void getId(int item);     // This puts item in the private data member
                             // itemNumber of the object that calls it.
   void getAmount(int num);  // This puts num in the private data member
                             // numOfItem of the object that calls it.
   void display();           // This prints to the screen
                             // the value of itemNumber and numOfItem of the
                             // object that calls it.


private:

   int  itemNumber;          // This is an id number of the product
   int  numOfItem;           // This is the number of items in stock

};


int main()
{

   ifstream infile;          // Input file to read values into array
   infile.open("Inventory.dat");

   // Fill in the code that defines an array of objects of class Inventory
   // called products. The array should be of size NUMOFPROD

   int pos;                  // loop counter
   int id;                   // variable holding the id number
   int total;                // variable holding the total for each id number

   // Fill in the code that will read inventory numbers and number of items
   // from a file into the array of objects. There should be calls to both
   // getId and getAmount member functions somewhere in this code.
   // Example: products[pos].getId(id); will be somewhere in this code

   // Fill in the code to print out the values (itemNumber and numOfItem) for
   // each object in the array products.
   // This should be done by calling the member function display within a loop

   return 0;

}

// Write the implementations for all the member functions of the class.
```

Exercise 1: Complete the program by giving the code explained in the commands in bold. The data file is as follows:

```
986  8
432  24
132  100
```

123 89
329 50
503 30
783 78
822 32
233 56
322 74

The output should be as follows:

Item number 986 has 8 items in stock
Item number 432 has 24 items in stock
Item number 132 has 100 items in stock
Item number 123 has 89 items in stock
Item number 329 has 50 items in stock
Item number 503 has 30 items in stock
Item number 783 has 78 items in stock
Item number 822 has 32 items in stock
Item number 233 has 56 items in stock
Item number 322 has 74 items in stock

LAB 13.5 Student Generated Code Assignments

Exercise 1: Give a C++ class declaration called `SavingsAccount` with the following information:

Operations (Member Functions)

1. Open account (with an initial deposit). This is called to put initial values in `dollars` and `cents`.
2. Make a deposit. A function that will add value to `dollars` and `cents`
3. Make a withdrawal. A function that will subtract values from `dollars` and `cents`.
4. Show current balance. A function that will print `dollars` and `cents`.

Data (Member Data)

1. `dollars`
2. `cents`

Give the implementation code for all the member functions.

NOTE: You must perform normalization on `cents`. This means that if `cents` is 100 or more, it must increment `dollars` by the appropriate amount. Example: if cents is 234, then dollars must be increased by 2 and cents reduced to 34.

Write code that will create an object called `bank1`. The code will then initially place $200.50 in the account. The code will deposit $40.50 and then withdraw $100.98. It will print out the final value of `dollars` and `cents`.

The following output should be produced:

```
Dollars = 140   cents = 2.
```

Part 2: Change the program to allow the user to input the initial values, deposit and withdrawal.

Example:

```
Please input the initial dollars
402

Please input the initial cents
78

Would you like to make a deposit? Y or y for yes
y
Please input the dollars to be deposited
35
Please input the cents to be deposited
67
Would you like to make a deposit? Y or y for yes
y
Please input the dollars to be deposited
35
Please input the cents to be deposited
67
Would you like to make a deposit? Y or y for yes
n
Would you like to make a withdrawal Y or y for yes
y
Please input the dollars to be withdrawn
28
Please input the cents to be withdrawn
08
Would you like to make a withdrawal Y or y for yes
y
Please input the dollars to be withdrawn
75
Please input the cents to be withdrawn
78
Would you like to make a withdrawal Y or y for yes
n

Dollars = 370 Cents = 26
```

Exercise 2: Replace the initial member function by two constructors. One constructor is the default constructor that sets both `dollars` and `cents` to 0. The other constructor has 2 parameters that set `dollars` and `cents` to the indicated values.

Have the code generate two objects: `bank1` (which has its values set during definition by the user) and `bank2` that uses the default constructor. Have the code input deposits and withdrawals for both `bank1` and `bank2`.