Object-Oriented Programming & Data Structures (CS-201)

Lab Manual

Prepared by

Engr. Fariha Atta

Table of Contents

Lab No.1 Structured Programming Refresher

1.1 Objectives of the lab

Reinstate the concepts of structured programming such as

1 decision making
2 iterations
3 functions
4 arrays

1.2 Activity

Write a function that computes the square of the elements stored in an array

Sample OUTPUT of the function

```
Enter length of the array: 5

Enter the elements of the array: 9      10    20    8    7

Input array is: 9   10    20    8    7

Square of the elements of input array is: 81    100    400    64    49
```

1.3 Activity

Write a program that takes the options from the user to perform addition, subtraction, multiplication, square root and modulus of a number(s). (Hint: use switch operator for selection of calculator's function).

Note: after selection of appropriate option your program must ask the user to enter the required data (in case of addition, subtraction and multiplication two numbers while one number for the other two options i.e. square root and modulus).

1.4 Activity

C++ code to find Multiplication Table of a number using for loop and represents it in proper format.

1.5 Activity

Write a function that reverses elements stored in an array

Sample OUTPUT of the function

```
Enter length of the array: 5

Enter the elements of the array: 11     22    33    44    55

Input array is: 11       22    33    44    55

Reverse of the input array is: 55      44    33    22    11
```

Note: Actual input array elements are to be swapped i.e. in the above example, swap array element 0 with element 4 and so forth. Temporary array for reversing the elements is not allowed.

1.6 Activity

A palindrome is a number or a text phrase that reads the same backwards as forwards. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554, and 11611. Write a program that reads in a five-digit integer and determines whether it is a palindrome. (Hint: use the division and modulus operators to separate the number into its individual digits).

1.7 Activity

Write a program that would store one hundred random numbers, ranging from 0 to 99, in an array; and displays the first repeating number in the array.

Hint: Numbers stored in the array has range [0, 99] and study the properties of % to figure out a way of getting a random number in range [0, 99].

1.8 Activity

Write a program that computes and prints both the mean and the standard deviation of the input data. The standard deviation of the n numbers ao, . . ., an, is defined by the formula

$$\sigma = \frac{\sqrt{\sum_{i=0}^{n-1}(a_i - \mu)^2}}{n-1}$$

where µ is the mean of the data. This means: square each deviation a[i] - mean; sum those squares; take the square root of that sum; divide that square root by n-l.

1.9 Activity

` Write a program to add two nxn matrixes.

1.10 References:

1 **Object-Oriented Programming in C++ by** ***Robert Lafore***

2 **How to Program C++ by** ***Deitel & Deitel***

Lab No.2 Introductory concepts of Object Oriented Programming

2.1 Objectives of the lab

Introducing the concepts of object-oriented programming and its implementation specifics such as

6 classes and objects
7 access rights
8 data members and member functions
9 constructors
10 constructor overloading

2.2 Pre-Lab

2.2.1 Class

1 A set of homogeneous objects
2 An Abstract Data Type (ADT) -- describes a new data type
3 A collection of data items or functions or both
4 Represents an entity having
    - Characteristics
        - Attributes (member data)
    - Behavior
        - Functionality (member functions)

5 Provides a plan/template/blueprint
    - Is just a declaration
    - Is a description of a number of similar objects

6 Class Definition is a Type Declaration -- No Space is Allocated

2.2.2 Syntax of Class

```
class Class_Name{
private:                //Default and Optional
        member_specification_1;
        :
        member_specification_n;
public:
        member_specification_n+1;
        :
        member_specification_n+m;
};              //Do not forget the semicolon after the closing brace.
```

2.2.3 Structure of a class

1 data members and member functions in a class
2 Data members are variables of either fundamental data types or user defined data types.

3. Member functions provide the interface to the data members and other procedures and function
4. Member access specifiers-- ***public, private, and protected*** used to specify the access rights to the members
    - ***Private***: available to member functions of the class
    - ***Protected***: available to member functions and derived classes
    - ***Public***: available to entire world
5. The default member accessibility is **private**, meaning that only class members can access the data member or member function.

2.2.4 A simple program using a class

```
#include <iostream.h>
class complex
{
private:
    double  re, im;
public:
    void set_val(double r, double i)
    {
            re=r;
            im=i;
    }
    void show()
    {
            cout<<"complex number: "<<re<<"+"<<im<<"i"<<endl;
    }
};

void main()
{
    complex         c1;
    c1.set_val(4,3);
    c1.show();
}
```

2.2.5 Constructor

- Special member function -- same name as the class name
    - initialization of data members
    - acquisition of resources
    - opening files
- Does NOT return a value
- Implicitly invoked when objects are created
- Can have arguments
- Cannot be explicitly called
- Multiple constructors are allowed
- If no constructor is defined in a program, default constructor is called
    - no arguments

- constructors can be overloaded (two constructors with same name but having different signatures)

2.2.6 A simple program demonstrating the use of constructor

```
#include <iostream.h>
class complex
{
private:
    double  re, im;
public:
    complex()                                   // simple method
    {       re=0;
                                            im=0;
        }
    complex(double r, double i):re(r), im(i)   //initializer list
    {}

    void set_val(double r, double i)
    {       re=r;
            im=i;
    }
    void show()
    {       cout<<"complex number: "<<re<<"+"<<im<<"i"<<endl;
    }
};
void main()
{
    complex         c1(3, 4.3);
        complex     c2;
    c1.show();
        c2.set_val(2, 5.5);
        c2.show();
}
```

2.2.7 In the above program, which constructors will be called for objects c1 and c2?

2.3 Post-Lab

2.3.1 Activity

Create a class, **Heater** that contains a single integer field, **temperature**. Define a constructor that takes no parameters. The **temperature** field should be set to the value 15 in the constructor. Define the mutators **warmer** and **cooler**, whose effect is to increase or decrease the value of the temperature by 5 respectively. Define an accessor method to return the value of **temperature**.

2.3.2 Activity

Create a class called ***BankAccount*** that models a checking account at a bank. The program creates an account with an opening balance, displays the balance, makes a deposit and a withdrawal, and then displays the new balance.

2.3.3 Activity

Modify your **Heater** class to define three new integer fields: **min**, **max**, and **increment**. The values of **min** and **max** should be set by parameters passed to the constructor. The value of **increment** should be set to 5 in the constructor. Modify the definitions of **warmer** and **cooler** so that they use the value of **increment** rather than an explicit value of 5. Before proceeding further, check that everything works as before. Now modify the **warmer** method so that it will not allow the temperature to be set to a value larger than **max**. Similarly modify **cooler** so that it will not allow **temperature** to be set to a value less than **min**. Check that the class works properly. Now add a method, **setIncrement** that takes a single integer parameter and uses it to set the value of **increment**. Once again, test that the class works as you would expect it to, by creating some **Heater** objects. Do things still work as expected if a negative value is passed to the **setIncrement** method? Add a check to this method to prevent a negative value from being assigned to **increment**.

2.3.4 Activity

Imagine a tollbooth at a bridge. Cars passing by the booth are expected to pay a 50 cent toll. Mostly they do, but sometimes a car goes by without paying. The tollbooth keeps track of the number of cars that have gone by, and of the total amount of money collected. Model this tollbooth with a class called **tollBooth**. The two data items are a type **unsigned int** to hold the total number of cars, and a type **double** to hold the total amount of money collected. A constructor initializes both of these to 0. A member function called **payingCar()** increments the car total and adds 0.50 to the cash total. Another function, called **nopayCar(),** increments the car total but adds nothing to the cash total. Finally, a member function called **display()** displays the two totals. Make appropriate member functions const.

Include a program to test this class. This program should allow the user to push one key to count a paying car, and another to count a nonpaying car. Pushing the [Escape] key should cause the program to print out the total cars and total cash and then exit.

2.3.5 Activity

Create a class called Complex for performing arithmetic with complex numbers. Complex numbers have the form realPart + imaginaryPart * i

where i is $\sqrt{-1}$

Write a program to test your class. Use floating-point variables to represent the private data of the class. Provide a constructor that enables an object of this class to be initialized when it is declared. Provide a no-argument constructor with default values in case no initializers are provided. Provide public methods that perform the following operations:

a) **Add two Complex numbers:** The real parts are added together and the imaginary parts are added together.

b) **Subtract two Complex numbers:** The real part of the right operand is subtracted from the real part of the left operand, and the imaginary part of the right operand is subtracted from the imaginary part of the left operand.

c) **Print** Complex numbers in the form (a, b), where a is the real part and b is the imaginary part.

d) In class Complex, define a **multiply** method that returns the product of two Complex numbers. Suppose you are trying to compute the product of two complex numbers a + bi and c + di. The real part will be ac – bd, while the imaginary part will be ad+ bc. Modify Complex Test to test your solution.

2.3.6 Activity

Create a class **Rectangle**. The class has attributes length and width, each of which defaults to 1. Provide methods that calculate the perimeter and the area of the rectangle. Provide set and get methods for both length and width. The set methods should verify that length and width are each floating-point numbers greater than or equal to 0.0 and less than 20.0. Write a program to test class Rectangle.

2.3.7 Activity

Create a class called **Employee** that includes three pieces of information as instance variables—a first name, a last name, and a monthly salary. Your class should have a constructor that initializes the three instance variables. Provide a set and a get method for each instance variable. If the monthly salary is not positive, set it to 0.0.Writea test application named **EmployeeTest** that demonstrates class Employee's capabilities. Create two Employee objects and display the yearly salary for each Employee. Then give each Employee a 10% raise and display each Employee's yearly salary again.

2.3.8 Activity

Create a class called **Invoice** that a hardware store might use to represent an invoice for an item sold at the store. An Invoice should include four pieces of information as instance variables—a part number, a part description, a quantity of the item being purchased (type int), and a price per item (double).Your class should have a constructor that initializes the four instance variables. Provide a set and a get method for each instance variable. In addition, provide a method named **getInvoiceAmount** that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as a double value. If the quantity is not positive, it should be set to 0. If the price per item is not positive, it should be set to 0.0.Writea test application named **InvoiceTest** that demonstrates class Invoice's capabilities.

2.3.9 Activity

Create a class called Date that includes three pieces of information as instance variables—a month (type int), a day (type int), and a year (type int). Your class should have a constructor that initializes the three instance variables and assumes that the values provided are correct. Provide a set and a get method for each instance variable. Provide a method **displayDate** that displays the month, day, and year separated by forward slashes (/). Write a test application named **DateTest** that demonstrates class Date's capabilities.

2.4 References:

3 **Class notes**

4 **Object-Oriented Programming in C++ by *Robert Lafore* (Chapter 6)**

5 **How to Program C++ by *Deitel & Deitel* (Chapter 6)**

Lab No.3 Using the global scope resolution operator, passing and returning objects to & from member functions

3.1 Objectives of the lab:

The purpose of this lab is to make you acquainted with the concepts such as

11 Defining member functions outside a class
12 Passing objects in function arguments
13 Returning objects from functions etc

3.2 Pre-Lab

3.2.1 Member functions defined outside a class

1 Member functions defined inside a class are ***inline*** by default
2 Declaration and definition can be separated from each other
   - Declaration inside the class
   - Definition outside the class

   Syntax:

   ```
   Return_type     class_name  ::   ftn_name   (list of parameters)
   {
                   Body of function
   }
   ```

1 **::** operator is called ***scope resolution operator***
   - specifies what class something is associated with

3.2.2 Objects as function arguments

1 A member function has direct access to all the other members of the class – **public** or **private**
2 A function has indirect access to other objects of the same class that are passed as arguments
   - Indirect access: using object name, a dot, and the member name

3.2.3 Example code

```
#include <iostream.h>
class complex
{
private:
    double re, im;
public:
    complex();
    complex(double r, double i);
```

```
    void add(complex c1, complex c2);
    void show();
};

complex::complex(): re(0), im(0)
{}
complex::complex(double r, double i): re(r), im(i)
{}

void complex::add(complex c1, complex c2)
{
    re=c1.re + c2.re;
    im=c1.im + c2.im;
}

void complex::show()
{
    cout<<"Complex no: "<<re<<"+"<<im<<"i"<<endl;
}

void main()
{
    complex        c1(3, 4.3), c2(2, 4);
    c3.add(c1, c2);
    c3.show();
}
```

3.2.4 Make changes in the program to allow for a call to add function as c3=c1.add(c2).

3.2.5 Returning objects from functions

Objects can be returned from functions just like normal primitive type variables.

3.2.6 Example code

```
#include <iostream.h>
class complex
{
private:
    double re, im;
public:
    complex();
    complex(double r, double i);
    complex negate();
    void show();
};
complex::complex(): re(0), im(0)
{}
complex::complex(double r, double i): re(r), im(i)
{}

complex complex::negate()
{
    complex temp;
    temp.re= - re;
    temp.im= - im;
    return temp;
}
void complex::show()
{
    cout<<"Complex no: "<<re<<"+"<<im<<"i"<<endl;
}

void main()
{
    complex        c1(3, 4.3), c2(2, 4);
    c3=c1.negate();
    c3.show();
}
```

3.3 Post-Lab

3.3.1 Activity

Create a class **RationalNumber** that stores a fraction in its original form (i.e. without finding the equivalent floating pointing result). This class models a fraction by using two data members: an integer for numerator and an integer for denominator. For this class, provide the following functions:

a) A **no-argument constructor** that initializes the numerator and denominator of a fraction to some fixed values.

b) A **two-argument constructor** that initializes the numerator and denominator to the values sent from calling function. This constructor should prevent a 0 denominator in a fraction, reduce or simplify fractions that are not in reduced form, and avoid negative denominators.

c) A function **AddFraction** for addition of two rational numbers.

Two fractions a/b and c/d are added together as:

$$\frac{a}{b}+\frac{c}{d}=\frac{(a*d)+(b*c)}{b*d}$$

d) A function **SubtractFraction** for subtraction of two rational numbers.

Two fractions a/b and c/d are subtracted from each other as:

$$\frac{a}{b}-\frac{c}{d}=\frac{(a*d)-(b*c)}{b*d}$$

e) A function **MultiplyFraction** for subtraction of two rational numbers.

Two fractions a/b and c/d are multiplied together as:

$$\frac{a}{b}*\frac{c}{d}=\frac{a*c}{b*d}$$

f) A function **DivideFraction** for division of two rational numbers.

If fraction a/b is divided by the fraction c/d, the result is

$$\frac{a}{b}\Big/\frac{c}{d}=\frac{a*d}{b*c}$$

g) Provide the following functions for comparison of two fractions

a. **isGreater**: should return a variable of type **bool** to indicate whether 1st fraction is greater than 2nd or not.

b. **isSmaller**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than 2nd or not.

c. **isGreaterEqual**: should return a variable of type **bool** to indicate whether 1st fraction is greater than or equal to 2nd or not.

d. **isSmallerEqual**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than or equal to 2nd or not.

h) Provide the following functions to check the equality of two fractions

e. **isEqual**: should return a variable of type **bool** to indicate whether 1st fraction is equal to the 2nd fraction or not.

f. **isNotEqual:** should a **true** value if both the fractions are not equal and return a **false** if both are equal.

NOTE: Define all the member functions outside the class.

3.3.2 Activity

Create a class called **Time** that has separate int member data for hours, minutes, and seconds. Provide the following member functions for this class:

a. A **no-argument constructor** to initialize hour, minutes, and seconds to 0.

b. A **3-argument constructor** to initialize the members to values sent from the calling function at the time of creation of an object. Make sure that valid values are provided for all the data members. In case of an invalid value, set the variable to 0.

c. A member function **show** to display time in 11:59:59 format.

d. A function **AddTime** for addition of two **Time** objects. Each time unit of one object must add into the corresponding time unit of the other object. Keep in view the fact that minutes and seconds of resultant should not exceed the maximum limit (60). If any of them do exceed, subtract 60 from the corresponding unit and add a 1 to the next higher unit.

   NOTE: Define all the member functions outside the class.

   A **main()** programs should create two initialized **Time** objects and one that isn't initialized. Then it should add the two initialized values together, leaving the result in the third **Time** variable. Finally it should display the value of this third variable.

3.3.3 Activity

Create a class called **Distance** containing two members feet and inches. This class represents distance measured in feets and inches. For this class, provide the following functions:

a) A **no-argument constructor** that initializes the data members to some fixed values.

b) A **2-argument constructor** to initialize the values of feet and inches to the values sent from the calling function at the time of creation of an object of type Distance.

c) **AddDistance** function to add two distances: Feet and inches of both objects should add in their corresponding members. 12 inches constitute one foot. Make sure that the result of addition doesn't violate this rule.

d) **isGreater**: should return a variable of type **bool** to indicate whether 1st distance is greater than 2nd or not.

e) **isSmaller**: should return a variable of type **bool** to indicate whether 1st distance is smaller than 2nd or not.

f) **isGreaterEqual**: should return a variable of type **bool** to indicate whether 1st distance is greater than or equal to 2nd or not.

g) **isSmallerEqual**: should return a variable of type **bool** to indicate whether 1st distance is smaller than or equal to 2nd or not.

h) **isEqual:** should return a variable of type **bool** to indicate whether 1st distance is equal to the 2nd distance or not.

i) **isNotEqual**: should a **true** value if both the distances are not equal and return a **false** if both are equal.

NOTE: Define all the member functions outside the class.

3.3.4 Activity

Create a class called **Martix** that represents a 3x3 matrix. This matrix contains a two-dimensional integer array of size 3x3. Provide the following member functions for this class:

a) a **no-argument constructor** for initializing the matrix with 0 values.

b) A **one-argument constructor** to initialize the member matrix to the matrix sent as an argument from the calling function.

c) An **AddMatrix** function for addition of two matrices

d) A **MultiplyMatrix** method for finding the product of the two matrices.

e) An **isEqual** function for checking the equality of two matrices

Note: Define all the member functions outside the class.

3.3.5 Activity

Create a class called **IntegerSet**. Each object of class **IntegerSet** can hold integers in the range 0 through 49. A set is represented internally as an array of ones and zeros. Array element a[i] is 1 if integer i is in the set. Array element a[j] is 0 if integer j is not in the set. The default constructor initializes a set to the so-called “empty set,” i.e., a set whose array representation contains all zeros.

Provide member functions for the common set operations. For example,

1. Provide a **unionOfIntegerSets** member function that creates a third set which is the set-theoretic union of two existing sets (i.e., an element of the third set’s array is set to 1 if that

element is 1 in either or both of the existing sets, and an element of the third set's array is set to 0 if that element is 0 in each of the existing sets).

2. Provide an **intersectionOfIntegerSets** member function that creates a third set which is the set-theoretic intersection of two existing sets (i.e., an element of the third set's array is set to 0 if that element is 0 in either or both of the existing sets, and an element of the third set's array is set to 1 if that element is 1 in each of the existing sets).
3. Provide an **insertElement** member function that inserts a new integer k into a set (by setting a[k] to 1).
4. Provide a **deleteElement** member function that deletes integer m (by setting a[m] to 0).
5. Provide a **setPrint** member function that prints a set as a list of numbers separated by spaces. Print only those elements that are present in the set (i.e., their position in the array has a value of 1).
6. Provide an **isEqualTo** member function that determines if two sets are equal.

Now write a driver program to test your **IntegerSet** class. Instantiate several **IntegerSet** objects. Test that all your member functions work properly.

3.4 References:

6 **Class notes**

7 **Object-Oriented Programming in C++ by** ***Robert Lafore*** **(Chapter 6)**

8 **How to Program C++ by** ***Deitel & Deitel*** **(Chapter 6)**

Lab No.4 Operator Overloading

4.1 Objectives of the lab:

Introducing the concepts of operator overloading and overloading different operators such as

14 Arithmetic operators
15 Relational operators
16 Logical operators
17 Unary operators

4.2 Pre-Lab

4.2.1 Operator overloading

- Allows to use operators for ADTs
- Most appropriate for math classes e.g. matrix, vector, etc.
- Gives operators class-specific functionality
- Analogous to function overloading -- *operator@* is used as the function name
- Operator functions are not usually called directly
- They are automatically invoked to evaluate the operations they implement
- Assume that + has been overloaded

  actual C++ code becomes
  c1+ c2 + c3 +c4
  The resultant code is very easy to read, write, and maintain

4.2.2 Syntax of operator function for binary operators

```
TYPE   operator    OP     (TYPE      rhs)
{
        body of function………
}
```

4.2.3 Example code for overloaded operator functions

```
#include <iostream.h>
class complex
{
private:
    double re, im;
public:
    complex():re(0), im(0)
    {}
    complex(double r, double i):re(r), im(i)
    {}
    void show()
    {
        cout<<"complex number: "<<re<<"+"<<im<<"i"<<endl;
    }

    complex operator + (complex rhs)
    {
        complex temp;
        temp.re=re + rhs.re;
        temp.im=im + rhs.im;
        return temp;
    }
  };

  void main()
  {
    complex     c1(3, 4.3), c2(2, 4);
    c1.show();
    c2.show();
    complex c3;
    c3=c1+ c2;         //Invocation of "+" Operator -- direct
  //or
  //c3=c1.operator+ (c2);   //Invocation of "+" Operator -- Function
    c3.show();
  }
```

4.2.4 Syntax of operator function for unary operators

```
    TYPE   operator    OP ()
    {
        body of function………
    }
```

4.2.5 Example code for overloaded operator functions

```
class complex
{
private:
    float re, im;
public:

    complex():re(0), im(0)
    {}
```

```
		complex(double r, double i):re(r), im(i)
		{}

		void show()
		{
			cout<<"complex number: "<<re<<"+"<<im<<"i"<<endl;
		}

		void operator ++()
		{
			++re;
			++im;
		}
};
void main()
{

		complex		c1(3, 4.3), c2;
		c2.set_val(4, 2.3);
		c1.pre_inc();
		c1.show();
		++c2;
		c2.show();
}
```

4.2.6 How can the above program be modified to allow for a statement like c2=++c1;?

4.3 Post-Lab

4.3.1 Activity

Create a class **RationalNumber** that stores a fraction in its original form (i.e. without finding the equivalent floating pointing result). This class models a fraction by using two data members: an integer for numerator and an integer for denominator. For this class, provide the following functions:

a) A **no-argument constructor** that initializes the numerator and denominator of a fraction to some fixed values.

b) A **two-argument constructor** that initializes the numerator and denominator to the values sent from calling function. This constructor should prevent a 0 denominator in a fraction, reduce or simplify fractions that are not in reduced form, and avoid negative denominators.

c) A **display** function to display a fraction in the format a/b.

d) An overloaded **operator +** for addition of two rational numbers.

Two fractions a/b and c/d are added together as:

$$\frac{a}{b}+\frac{c}{d}=\frac{(a*d)+(b*c)}{b*d}$$

e) An overloaded **operator -** for subtraction of two rational numbers.

Two fractions a/b and c/d are subtracted from each other as:

$$\frac{a}{b}-\frac{c}{d}=\frac{(a*d)-(b*c)}{b*d}$$

f) An overloaded **operator *** for subtraction of two rational numbers.

Two fractions a/b and c/d are multiplied together as:

$$\frac{a}{b}*\frac{c}{d}=\frac{a*c}{b*d}$$

g) An overloaded **operator /** for division of two rational numbers.

If fraction a/b is divided by the fraction c/d, the result is

$$\frac{a}{b}\Big/\frac{c}{d}=\frac{a*d}{b*c}$$

h) Overloaded relational operators

a. **operator >**: should return a variable of type **bool** to indicate whether 1st fraction is greater than 2nd or not.

b. **operator <**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than 2nd or not.

c. **operator >**=: should return a variable of type **bool** to indicate whether 1st fraction is greater than or equal to 2nd or not.

d. **operator <=**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than or equal to 2nd or not.

i) Overloaded equality operators for **RationalNumber** class

e. **operator==**: should return a variable of type **bool** to indicate whether 1st fraction is equal to the 2nd fraction or not.

f. **Operator!=:** should a **true** value if both the fractions are not equal and return a **false** if both are equal.

4.3.2 Activity

Create a class called **Time** that has separate int member data for hours, minutes, and seconds. Provide the following member functions for this class:

a) A **no-argument constructor** to initialize hour, minutes, and seconds to 0.

b) A **3-argument constructor** to initialize the members to values sent from the calling function at the time of creation of an object. Make sure that valid values are provided for all the data members. In case of an invalid value, set the variable to 0.

c) A member function **show** to display time in 11:59:59 format.

d) An overloaded **operator+** for addition of two Time objects. Each time unit of one object must add into the corresponding time unit of the other object. Keep in view the fact that minutes and seconds of resultant should not exceed the maximum limit (60). If any of them do exceed, subtract 60 from the corresponding unit and add a 1 to the next higher unit.

e) Overloaded operators for **pre- and post- increment**. These increment operators should add a 1 to the **seconds** unit of time. Keep track that **seconds** should not exceed 60.

f) Overload operators for **pre- and post- decrement**. These decrement operators should subtract a 1 from **seconds** unit of time. If number of seconds goes below 0, take appropriate actions to make this value valid.

A **main()** programs should create two initialized **Time** objects and one that isn't initialized. Then it should add the two initialized values together, leaving the result in the third **Time** variable. Finally it should display the value of this third variable. Check the functionalities of ++ and -- operators of this program for both pre- and post-incrementation.

4.3.3 Activity

Create a class called **Distance** containing two members feet and inches. This class represents distance measured in feets and inches. For this class, provide the following functions:

f) A **no-argument constructor** that initializes the data members to some fixed values.

g) A **2-argument constructor** to initialize the values of feet and inches to the values sent from the calling function at the time of creation of an object of type Distance.

h) Overloaded arithmetic operators

   a. **operator+** to add two distances: Feet and inches of both objects should add in their corresponding members. 12 inches constitute one feet. Make sure that the result of addition doesn’t violate this rule.

b. **operator+=** for addition of two distances.

i) overloaded relational operators

a. **operator >**: should return a variable of type **bool** to indicate whether 1st distance is greater than 2nd or not.

b. **operator <**: should return a variable of type **bool** to indicate whether 1st distance is smaller than 2nd or not.

c. **operator >**=: should return a variable of type **bool** to indicate whether 1st distance is greater than or equal to 2nd or not.

d. **operator <=**: should return a variable of type **bool** to indicate whether 1st distance is smaller than or equal to 2nd or not.

j) Overloaded equality operators

a. **operator==:** should return a variable of type **bool** to indicate whether 1st Distance is equal to the 2nd distance or not.

b. **Operator!=**: should a **true** value if both the distances are not equal and return a **false** if both are equal.

4.3.4 Activity

Create a **Point** class has two coordinates x and y. Provide the following member functions:

a) Two constructors: one for initializing to fixed values and one for initializing to the values sent from outside.

b) A function for displacing the object Point. Provide two values to this function which are added to the x and y coordinates as displacements.

c) Functions for overloading the addition and subtraction operators: +=, +, -=, and unary and binary-.

d) Function for overloading the equality operator.

Now use the overloaded operator + for adding displacement to a Point. In this case, the displacement will be represented by an object of type Point. So

P2=P1 + Disp; where P2, P1, and Disp all are instances of class **Point**.

4.3.5 Activity

Create a class called **Martix** that represents a 3x3 matrix. Create a constructor for initializing the matrix with 0 values. Create another overloaded constructor for initializing the matrix to the values sent from outside. Overload the + and += operators for addition of two matrices, == operator for checking the equality of two matrices, and *operator for finding the product of the two matrices. Define all the member functions outside the class.

4.3.6 Activity

Create a class called **Complex** for performing arithmetic with complex numbers. Complex numbers have the form realPart + imaginaryPart * i

where i is $\sqrt{-1}$

Write a program to test your class. Use floating-point variables to represent the private data of the class. Provide a constructor that enables an object of this class to be initialized when it is declared. Provide a no-argument constructor with default values in case no initializers are provided. Provide public methods that perform the following operations:

a) **Add two Complex numbers**: Overload the operators +, += for addition of two complex numbers.

b) **Subtract two Complex numbers**: overload the operators -, -= for subtraction of two complex numbers.

c) **Increment a complex number**: incrementing a complex number results in addition of 1 to the real portion. Overload both the pre- and post-incrementation operators.

d) **Decrement a complex number**: decrementing a complex number results in subtraction of 1from the real portion. Overload both the pre- and post-decrementation operators.

e) **Print** Complex numbers in the form (a, b), where a is the real part and b is the imaginary part.

f) **Multiply two complex number**: overload * and *= operators for multiplication of complex numbers.

4.3.7 Activity

Let us model digital storage. Digital data is stored in the form of bits. 8 bits constitute one byte. Create a class **Storage** that specifies the size of a file. This class has two integer data members: bits and bytes.

g) Provide a **no-argument constructor** to initialize the size of some file to 0 bits and 0 bytes.

h) Provide a **two-argument constructor** to initialize the size of a file to values specified at the time of creation of an object.

i) Provide an overloaded **operator +** that is used to merge two files together in a third file.

j) Provide an overloaded **operator +=** that is used to concatenate one file at the end of the other.

k) Provide overloaded **post-increment and pre-increment** operators to increment the size of a file by one bit. (You must write the functions to accommodate statements like f2=++f1; and f2=f1++; where f1 and f2 are instances of the class Storage)

l) Provide an overloaded **operator >** to determine whether one file is larger in size than the other.

Write a test program to test the functionality of this class.

4.4 References:

9 **Class notes**

10 **Object-Oriented Programming in C++ by** ***Robert Lafore*****(Chapter 8)**

11 **How to Program C++ by** ***Deitel & Deitel*** **(Chapter 8)**

Lab No.5 Inheritance

5.1 Objectives of the lab:

Introducing the concepts of inheritance

18 Base and derived classes
19 Public inheritance
20 Private inheritance
21 Protected inheritance

5.2 Pre-Lab

5.2.1 Inheritance

1 New classes created from existing classes
2 Absorb attributes and behaviors from base class
3 Classes are often closely related
   - "Factor out" common attributes and behaviors and place these in a base class
   - Use inheritance to form derived classes
4 Derived class
   - Class that inherits data members and member functions from a previously defined base class
   - Derived class *extends* the behavior of the base class.
5 The class from which another class is derived is called ***base class*** or ***parent class*** or ***super-class***
6 The class which is derived from another class is called a ***derived class*** or ***child class*** or ***subclass***
7 The derived class is a superset of the base class

5.2.2 Syntax for Inheritance

```
class ChildClass: access_specifier BaseClass
{
...
};
```

5.2.3 Protected access specifier

1 Intermediate level of protection between **public** and **private** inheritance
2 Protected member can't be accessed from outside the base & derived classes
3 Derived-class members can refer to **public** and **protected** members of the base class simply by using the member names

5.2.4 Example of inheritance

```
#include <iostream.h>
class A
```

```
{
private:
      int       num1;
protected:
      int       num2;
public:
      int       num3;
      void      display()
      {
                cout<<"Number 1: "<<num1<<endl;
                cout<<"Number 2: "<<num2<<endl;
                cout<<"Number 3: "<<num3<<endl;
      }
};
class B: public A
{
private:
      int       num4;
public:
      void      print()
      {
                cout<<"Number 1: "<<num1<<endl;      //error: can't access private members
                cout<<"Number 2: "<<num2<<endl;
                cout<<"Number 3: "<<num3<<endl;
                cout<<"Number 4: "<<num4<<endl;
      }
};

void main()
{
      B         obj;
      obj.print();
}
```

Q. In case of private and protected inheritance, would the public member num3 of class B be accessible to obj in main i.e. can we write obj.num3=2; ? If not, why?

5.3 Post-Lab

5.3.1 Activity

Create a class called **Point** that has two data members: **x**- and **y**-coordinates of the point. Provide a no-argument and a 2-argument constructor. Provide separate get and set functions for the each of the data members i.e. **getX**, **getY**, **setX**, **setY**. The getter functions should return the corresponding values to the calling function. Provide a **display** method to display the point in (x, y) format. Make appropriate functions **const**.

Derive a class **Circle** from this **Point** class that has an additional data member: **radius** of the circle. The point from which this circle is derived represents the center of circle. Provide a no-argument constructor to initialize the radius and center coordinates to 0. Provide a 2-argument constructor: one argument to initialize the radius of circle and the other argument to initialize the center of circle (provide an object of point class in the second argument). Provide a 3-argument constructor that takes three floats to initialize the radius, x-, and y-coordinates of the circle. Provide setter and getter functions for radius of the circle. Provide two functions to determine the radius and circumference of the circle.

Write a main function to test this class.

5.3.2 Activity

Derive a class **Cylinder** from the **Circle** class of activity 1. This class contains an additional data member: height of cylinder. Provide appropriate constructors to initialize the center, radius, and height of the cylinder. Provide functions to determine the area and volume of the cylinder. Area of a cylinder is 2πr*r+ 2πr*h. Volume of cylinder is 2πr*r*h. Use the **clac_area** of circle class where required.

5.3.3 Activity

Imagine a publishing company that markets both book and audiocassette versions of its works. Create a class **publication** that stores the title (a **char** string) and price (type **float**) of a publication. From this class derive two classes: **book**, which adds a page count (type **int**); and **tape**, which adds a playing time in minutes (type **float**). Each of these three classes should have a **getdata()** function to get its data from the user at the keyboard, and a **putdata()** function to display its data. Determine whether public, private, or protected inheritance should be used. Justify your answer.

Write a **main()** program to test the **book** and **tape** classes by creating instances of them, asking the user to fill in data with **getdata()** and then displaying the data with **putdata()**.

5.3.4 Activity

Start with the **publication**, **book**, and **tape** classes of activity 3. Add a base class **sales** that holds an array of three floats so that it can record the dollar sales of a particular publication for the last three months. Include a **getdata()** function to get three sales amount from the user, and a **putdata()** function to display the sales figures. Alter the **book** and **tape** classes so they are derived from both **publication** and **sales**.

An object of class **book** or **tape** should input and output sales data along with its other data. Write a **main()** function to create a **book** object and a **tape** object and exercise their input/output capabilities. Determine whether **book** and **tape** classes should be publicly, privately, or protectedly inherited from **publication** and **sales** classes. Justify your answer.

5.3.5 Activity

Assume that the publisher in activity 3 decides to add a third way to distribute books: on computer disk, for those who like to do their reading on their laptop. Add a **disk** class that, like **book** and **tape**, is derived from **publication**. The **disk** class should incorporate the same member functions as the other classes. The data item unique to this class is the disk size: either 3-1/2 inches or 5-1/4 inches. You can use an **enum** Boolean type to store this item, but the complete size should be displayed. The user could select the appropriate size by typing 3 or 5.

5.3.6 Activity

Start with the **publication**, **book**, and **tape** classes of activity. Suppose you want to add the date of publication for both books and tapes. From the publication class, derive a new class called **publication2** that includes this member data. Determine whether inheritance or composition should be used. Then change **book** and **tape** so they are derived from **publication2** instead of **publication**. Make all the necessary changes in member functions so the user can input and output dates along with the other data. For the dates, you can use the **date** class from the Lab3 that stores a **date** as three **ints**, for month, day, and year.

5.4 References:

12 **Class notes**

13 **Object-Oriented Programming in C++ by *Robert Lafore*(Chapter 9)**

14 **How to Program C++ by *Deitel & Deitel* (Chapter 9)**

Lab No.6 Multiple and Multilevel Inheritance

6.1 Objectives of the lab

Introducing the concepts of inheritance such as

22 Function overriding
23 Multiple inheritance
24 Multilevel inheritance

This lab also provides insight into composition and multifile programming.

6.2 Pre-Lab

6.2.1 Overriding member functions of the base class

1. A derived class can override the member functions of its base class
2. To override a function of base class, the derived class provides a function with same signature as that of the base class
3. So derived class function overrides the base class function
4. Base class functions are overridden if we want to change the functionality of base class function
    - Functions can be totally changed
    - A few new additions can be made in the overriding function
5. Derived class method and base class method have same signatures but when this function is called on a derived class object, the function of derived class object will be called although the base class functions are making the interface of the derived class
6. To explicitly call the base class overridden function, use parent_class_name::member_function_name(…)

```
#include <iostream.h>
class A
{
private:
    int     num1;
public:
    A(): num1(0)
    {}
    void    display()
    {       cout<<"Number 1: "<<num1<<endl;
    }
};
class B: public A
{
private:
    int     num2;
public:
    B():num2(0)
    {}
    void    display()
    {
            cout<<"Number 2: "<<num2<<endl;
    }
};
void main()
{
    B       obj;
    obj.display();    //display of class B called
    obj.A::display(); //display of class A called
}
```

6.2.2 Multiple inheritance

1. A class can be derived from more than one classes
2. The order in which the constructors of base classes are called is not dependent on the order of mentioning constructors in base class initializer. It is dependent on the order in which inheritance is specified in the derived-class definition

6.2.3 Example

```
#include <iostream.h>
class A
{
private:
    int     num1;
public:
    A():num1(0)
    {       cout<<"Class A constructor..."<<endl;
    }
};

class B
{
private:
    int     num2;
public:
    B():num2(0)
    {
            cout<<"Class B constructor..."<<endl;
```

```
        }
};

class C: public B, public A
{
public:
        C()
        {
                cout<<"Class C constructor..."<<endl;
        }
};

void main()
{
        C       obj;
}
```

Q. if there are display functions in each of the three classes, which class' display function gets called if the statement obj.display(); gets executed? Write statements to call the display function of all the three classes on obj.

6.2.4 Multifile programming

A program can be broken down into different files as:

6.2.5 Example

Header.h

```
#ifndef HEADER_H
#define HEADER_H
class Point
{
private:
        int     x, y;
public:
        Point(): x(0), y(0)
        {}
        Point operator+(Point);
        void   display();
};
#endif
```

Header.cpp

```
#include <iostream.h>
#include "header.h"

Point Point::operator+(Point rhs)
{
	Point temp;
	temp.x= x + rhs.x;
	temp.y= y + rhs.y;
	return temp;
}

void Point::display()
{
	cout<<"("<<x<<","<<y<<")"<<endl;
}
```

Mainfile.cpp

```
#include <iostream.h>
#include "header.h"
void main()
{
	Point p1(3, 4), p2(4, 2);
	Point p3;

	cout<<"First Point is: ";
	p1.display();

	cout<<"Second Point is: ";
	p2.display();

	p3= p1 + p2;
	cout<<"The result of Addition is: ";
	p3.display();
}
```

6.3 Post-Lab

6.3.1 Activity

Create a class **Employee** that has a field for storing the complete name of employee (first and last name), a field for storing the IDentification number of employee, and another field for storing his salary.

Provide

a) a **no-argument constructor** for initializing the fields to default values

b) a **3-argument constructor** for initializing the fields to values sent from outside. Use strcpy function to copy one string into another.

c) a setter function (mutator) that sets the values of these fields by getting input from user.

d) an accessor function to display the values of the fields.

Derive three classes from this employee class: **Manager**, **Scientist**, and **Laborer**. **Manager** class has an additional data member of # of subordinates. **Scientist** class contains additional information about # of publications. **Laborer** class is just similar to the employee class. It has no additional capabilities. Derive a class **foreman** from the **Laborer** class that has an additional data member for storing the percentage of quotas met by a **foreman**. Provide appropriate no-argument and n-argument constructors for all the classes. Provide the overridden getter and setter functions here too to input and output all the fields. Determine whether public, private, or protected inheritance should be used.

6.3.2 Activity

Create a class **Student** that contains information about a student's name, semester, roll no, and date of admission. To store the date of admission, again reuse the date class that you have already developed. Determine whether you should use inheritance or composition.

Provide

a) a **no-argument constructor** for initializing the values of data members to some defaults.

b) a **4-argument constructor** to initialize the data members sent from the calling function at the time of creation of an object(date should be sent from outside in the form of a date object).

c) An **input** function for setting the status of a student.

d) A **display** function to display all the attributes of a student.

6.3.3 Activity

Derive a class **Undergraduate** from **Student** class of activity 2 that contains some additional information. This information is about the semester gpa of a student and the credit points earned per semester. To store this data, provide a 2D array (2x8 array since at maximum there are 8 semesters for an undergraduate program). One dimension of the array should hold information about the SGPA of each semester so far and the other dimension should hold the corresponding credit points earned in that semester.

a) Create a no-argument and a 5-argument constructor for data member initialization.

b) Provide overridden functions for getting and setting the data members.

c) Provide another function to calculate the CGPA of student on the basis of the information provided by the 2D array.

Derive a class **Graduate** from **Student** class that also has the same additional information as the **Undergraduate** class but in this case, the array is 2x4 since at maximum there are four semesters in a Graduate program. There are two additional data members: one to store the title of the last degree held and another to store the area of specialization in graduate program. Provide appropriate constructors and overridden member functions.

6.3.4 Activity

Derive a class **Complex2** from the class **Complex** of lab 4 that has additional member functions:

a) **norm** for calculating the normal (a2 + b2) of a complex number,

b) **polar** for calculating the phase of a complex number (a + bi) by atan(b/a) and magnitude by sqrt(a2 + b2),

c) **conj** for determining and displaying the conjugate of a complex number,

d) a **rectangular** function to get the rectangular coordinates back from phase and magnitude of a complex number.

6.3.5 Activity

Redo all the questions of Lab5 using multi-file programming. Create a header file for your class declaration, a .cpp file for class definition, and another .cpp file containing main function.

6.4 References

15 **Class notes**

16 **Object-Oriented Programming in C++ by** ***Robert Lafore*** **(Chapter 9)**

17 **How to Program C++ by** ***Deitel & Deitel*** **(Chapter 9)**

Lab No.7 Dynamic Memory Allocation and miscellaneous topics

7.1 Objectives of the lab:

This lab covers some miscellaneous topics such as

- Dynamic memory allocation
- Copy Constructor (Deep & Shallow copy)
- Const members and objects
- Cascaded function calls
- Static data members, member functions, and objects

7.1.1 Activity

Study the following programs, execute them, and determine what possibly the cause of error. Write the additional code to make these programs run.

PROGRAM 1

```
#include <iostream.h>
#include <string.h>

class student
{
private:
        char    *name;
        int     roll;
        int     semester;
public:
        student(char *n, int r, int s): roll(r), semester(s)
        {
                name=new char[strlen(n) + 1];
                strcpy(name, n);
        }
        void set()
        {
                cout<<"Enter name: "<<endl;
                cin>>name;
                cout<<"Enter roll no: "<<endl;
                cin>>roll;
                cout<<"Enter semester: "<<endl;
```

```
		cin>>semester;
	}

	void show()
	{
		cout<<"Name: "<<name<<endl;
		cout<<"Roll NO: "<<roll<<endl;
		cout<<"Semester: "<<semester<<endl;
	}

	~student()
	{
		delete[] name;
	}

};

void main()
{
	student s1("Bjarne Stroustrup", 3, 3);
	s1.show();

	{
		student s2=s1;
		s2.show();
	}
	s1.show();
}
```

PROGRAM 2

```
#include <iostream.h>
#include <string.h>
class student
{
private:
	char	*name;
	int	roll;
	int	semester;
public:

	student(): roll(0),	semester(0)
	{
		name=new char[20];
		strcpy(name, "");
	}

	student(char *n, int r, int s): roll(r), semester(s)
	{
		name=new char[strlen(n) + 1];
		strcpy(name, n);
	}


	void set()
	{
		cout<<"Enter name: "<<endl;
		cin>>name;
		cout<<"Enter roll no: "<<endl;
		cin>>roll;
		cout<<"Enter semester: "<<endl;
		cin>>semester;
	}

	void show()
	{
		cout<<"Name: "<<name<<endl;
		cout<<"Roll NO: "<<roll<<endl;
		cout<<"Semester: "<<semester<<endl;
	}

	~student()
	{
		delete[] name;
	}

};

void main()
{
	student s1("Bjarne Stroustrup", 3, 3);
	s1.show();

	student s2(s1);
	s2.show();
```

```
        s2.set();
        cout<<"After setting s2:"<<endl;
        cout<<"Data of student S1"<<endl;
        s1.show();

        cout<<"Data of student s2"<<endl;
        s2.show();
}
```

7.1.2 Activity

Create a class called **student**. This class contains data members for name, roll no, and CGPA of a student.

1. Provide a **no-argument constructor** for initializing the data members to some fixed value.
2. Provide a **2-argument constructor** to initialize the data members to the values sent from the calling function.
3. Provide separate **setter** functions for setting each data member. These functions should take the values from user at run-time.
4. Provide separate **getter** functions for each data member. The getter functions should return the value of the corresponding fields.
5. Create a function **display** that displays all the information to user.

Let us suppose that we want to keep information about average CGPA of students in a particular department. Make appropriate changes in the class to handle this extra information (**Hint**: provide **static** data members for average CGPA and no of students and set the values for these members in constructor). Provide a **static** function to display this additional information.

7.1.3 Activity

Create a class called **employee**. This class maintains information about name (**char***), department (**char***), salary (**double**), and period of service in years (**double**).

1. Provide a **no-argument constructor** to initialize the data members to some fixed values.
2. Provide a **4-argument constructor** to initialize the members to values sent from calling function.

   (You have to make dynamic allocation for both name and department data members in constructor.)
3. Provide a **copy-constructor** that performs the deep copy of the data members.
4. Provide an **input** function that takes all the values from user during run-time.

5. Provide a **display** function that displays all the information about a specific student to user.

6. Provide a **destructor** to free the memory allocated to name and department in constructor.

   Write a driver program to test the functionality of the above-mentioned class.

7.1.4 Activity

Create a class called **Complex** for performing arithmetic with complex numbers. Complex numbers have the form realPart + imaginaryPart * i

where i is $\sqrt{-1}$

Write a program to test your class. Use floating-point variables to represent the private data of the class.

1. Provide a **constructor** that enables an object of this class to be initialized when it is declared.
2. Provide a **no-argument cons**tructor with default values in case no initializers are provided.
3. **Print** Complex numbers in the form (a, b), where **a** is the real part and **b** is the imaginary part.
4. Provide separate **setter** functions for setting the real and imaginary portions of this class.
5. Provide separate **getter** functions for both the fields that must return the values of real and imaginary parts of a complex number.

Note: Write the setter functions as well as the print function in a manner that allows for the cascaded calls of these functions.

Write a driver program that tests the functionality of this class.

7.2 References:

18 **Class notes**

19 **Object-Oriented Programming in C++ by** ***Robert Lafore*** **(Chapter 6)**

20 **How to Program C++ by** ***Deitel & Deitel*** **(Chapter 7)**

Lab No.8 Friend Functions and Classes

8.1 Objectives of the lab:

Introducing the concepts of friendship such as

25 Friend functions
26 Friend classes

8.2 Pre-Lab

8.2.1 Friend Functions

1 A regular C-style function accessing the non-public members of the objects of a class to which it is a friend
   - ‽ Not a member function
   - ‽ Can access non-public members of the objects to which it is a friend

2 Declaration inside the class preceded by keyword **friend**
   - ‽ Keyword friend is not used with definition. //Compiler error

3 Definition outside the class but like the definition of normal C-style functions
4 **this** is NOT Available to friend Functions
5 A friend function is not called in the manner member functions are called. Friend function is called like normal C-style functions and takes the object of the class to which it is a friend
6 A friend can be declared in **public**, **private**, or **protected** portions.
   - ‽ Member access specifiers have no effect on friend function
   - ‽ Friend function is not a member

7 Friend functions are not inherited

8.2.2 Example

```
#include <iostream.h>
class test
{
private:
    int     data;
public:
    test(): data(0)
    {}

    void show()
    {       cout<<data<<endl;
    }

    friend void set_data(test &);
};
void set_data(test &obj)
{
    obj.data=3;
}

void main()
{
    test    t;
    set_data(t);    //called like normal function
    t.show();
}
```

8.2.3 Friend classes

- Not directly associated with the Class
- All the member functions of the friend class can access non-public data members of the original class
    - If class A is the friend of class B. all its member functions can access the nonpublic members of class B
- Declaration

```
class B
{
friend class A;
…
}
```

- Can be friend of more than one classes
- Violation of encapsulation
- SHOULD BE USED ONLY WHEN REQUIRED

8.2.4 Example

```
#include <iostream.h>
#include <string.h>
class A
{
private:
    int     x;
    friend class B;
public:
    A(): x(0)
    {}

    void showA()
    {
            cout<<x<<endl;
    }
};
class B
{
private:
```

```
            int     y;
public:
            B(): y(0)
            {}

            void showB(A &a)
            {
                    a.x=4;
                    cout<<a.x<<endl;
                    cout<<y<<endl;
            }
};

void main()
{
    A       obja;
    B       objb;
    objb.showB(obja);
    cout<<"after calling showB()..."<<endl;
    obja.showA();
}
```

Q. What would happen if we write the statement friend class B; in public or protected portions? Would it have any effect on behavior of the class?

8.3 Post-Lab

8.3.1 Activity

Create a class **RationalNumber** that stores a fraction in its original form (i.e. without finding the equivalent floating pointing result). This class models a fraction by using two data members: an integer for numerator and an integer for denominator. For this class, provide the following functions:

a) A **no-argument constructor** that initializes the numerator and denominator of a fraction to some fixed values.

b) A **two-argument constructor** that initializes the numerator and denominator to the values sent from calling function. This constructor should prevent a 0 denominator in a fraction, reduce or simplify fractions that are not in reduced form, and avoid negative denominators.

c) A **display** function to display a fraction in the format a/b.

d) Provide the following operator functions as **non-member friend functions.**

i. An overloaded **operator +** for addition of two rational numbers.

Two fractions a/b and c/d are added together as:

$$\frac{a}{b}+\frac{c}{d}=\frac{(a*d)+(b*c)}{b*d}$$

ii. An overloaded **operator -** for subtraction of two rational numbers.

Two fractions a/b and c/d are subtracted from each other as:

$$\frac{a}{b}-\frac{c}{d}=\frac{(a*d)-(b*c)}{b*d}$$

iii. An overloaded **operator *** for subtraction of two rational numbers.

Two fractions a/b and c/d are multiplied together as:

$$\frac{a}{b}*\frac{c}{d}=\frac{a*c}{b*d}$$

iv. An overloaded **operator /** for division of two rational numbers.

If fraction a/b is divided by the fraction c/d, the result is

$$\frac{a}{b} \Big/ \frac{c}{d}=\frac{a*d}{b*c}$$

v. Overloaded relational operators

a. **operator >**: should return a variable of type **bool** to indicate whether 1st fraction is greater than 2nd or not.

b. **operator <**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than 2nd or not.

c. **operator >=**: should return a variable of type **bool** to indicate whether 1st fraction is greater than or equal to 2nd or not.

d. **operator <=**: should return a variable of type **bool** to indicate whether 1st fraction is smaller than or equal to 2nd or not.

vi. Overloaded equality operators for **RationalNumber** class

a. **operator==**: should return a variable of type **bool** to indicate whether 1st fraction is equal to the 2nd fraction or not.

b. **Operator!=:** should a **true** value if both the fractions are not equal and return a **false** if both are equal.

8.3.2 Activity

Create a class called **Time** that has separate int member data for hours, minutes, and seconds. Provide the following member functions for this class:

a) A **no-argument constructor** to initialize hour, minutes, and seconds to 0.

b) A **3-argument constructor** to initialize the members to values sent from the calling function at the time of creation of an object. Make sure that valid values are provided for all the data members. In case of an invalid value, set the variable to 0.

c) A member function **show** to display time in 11:59:59 format.

d) Provide the following functions as **friends**

   a. An overloaded **operator+** for addition of two Time objects. Each time unit of one object must add into the corresponding time unit of the other object. Keep in view the fact that minutes and seconds of resultant should not exceed the maximum limit (60). If any of them do exceed, subtract 60 from the corresponding unit and add a 1 to the next higher unit.

   b. Overloaded operators for **pre- and post- increment**. These increment operators should add a 1 to the **seconds** unit of time. Keep track that **seconds** should not exceed 60.

   c. Overload operators for **pre- and post- decrement**. These decrement operators should subtract a 1 from **seconds** unit of time. If number of seconds goes below 0, take appropriate actions to make this value valid.

A **main()** programs should create two initialized **Time** objects and one that isn't initialized. Then it should add the two initialized values together, leaving the result in the third **Time** variable. Finally it should display the value of this third variable. Check the functionalities of ++ and -- operators of this program for both pre- and post-incrementation.

8.3.3 Activity

Create a class called **Distance** containing two members feet and inches. This class represents distance measured in feets and inches. For this class, provide the following functions:

a) A **no-argument constructor** that initializes the data members to some fixed values.

b) A **2-argument constructor** to initialize the values of feet and inches to the values sent from the calling function at the time of creation of an object of type Distance.

c) Provide the following operators as friends:

    a. **operator+** to add two distances: Feet and inches of both objects should add in their corresponding members. 12 inches constitute one feet. Make sure that the result of addition doesn't violate this rule.

    b. **operator+=** for addition of two distances.

    c. **operator >**: should return a variable of type **bool** to indicate whether 1st distance is greater than 2nd or not.

    d. **operator <**: should return a variable of type **bool** to indicate whether 1st distance is smaller than 2nd or not.

    e. **operator >**=: should return a variable of type **bool** to indicate whether 1st distance is greater than or equal to 2nd or not.

    f. **operator <=**: should return a variable of type **bool** to indicate whether 1st distance is smaller than or equal to 2nd or not.

    g. **operator==:** should return a variable of type **bool** to indicate whether 1st Distance is equal to the 2nd distance or not.

    h. **operator**!=: should a **true** value if both the distances are not equal and return a **false** if both are equal.

8.3.4 Activity

Let us model digital storage. Digital data is stored in the form of bits. 8 bits constitute one byte. Create a class **Storage** that specifies the size of a file. This class has two integer data members: bits and bytes.

a) Provide a **no-argument constructor** to initialize the size of some file to 0 bits and 0 bytes.

b) Provide a **two-argument constructor** to initialize the size of a file to values specified at the time of creation of an object.

c) Provide the following functions as friends:

    a. Provide an overloaded **operator +** that is used to indicate the size of the resultant file obtained as a result of merging two files.

b. Provide an overloaded **operator +=** that is used to indicate the size of a file if another file is concatenated at the end of it.

c. Provide overloaded **post-increment and pre-increment** operators to increment the size of a file by one bit. (You must write the functions to accommodate statements like f2=++f1; and f2=f1++; where f1 and f2 are instances of the class Storage)

d. Provide an overloaded **operator >** to determine whether one file is larger in size than the other or not. This function should return a **bool** type variable.

Write a driver program to test the functionality of this class.

8.3.5 Activity

Create a class called **IntegerSet**. Each object of class **IntegerSet** can hold integers in the range 0 through 49. A set is represented internally as an array of ones and zeros. Array element a[i] is 1 if integer i is in the set. Array element a[j] is 0 if integer j is not in the set. The default constructor initializes a set to the so-called "empty set," i.e., a set whose array representation contains all zeros.

Provide **friend** functions for the common set operations. For example,

1. Provide an **operator +** non-member function that creates a third set which is the set-theoretic union of two existing sets (i.e., an element of the third set's array is set to 1 if that element is 1 in either or both of the existing sets, and an element of the third set's array is set to 0 if that element is 0 in each of the existing sets).

2. Provide an **operator*** function that creates a third set which is the set-theoretic intersection of two existing sets (i.e., an element of the third set's array is set to 0 if that element is 0 in either or both of the existing sets, and an element of the third set's array is set to 1 if that element is 1 in each of the existing sets).

3. Provide an **insertElement** member function that inserts a new integer k into a set (by setting a[k] to 1).

4. Provide a **deleteElement** member function that deletes integer m (by setting a[m] to 0).

5. Provide a **setPrint** member function that prints a set as a list of numbers separated by spaces. Print only those elements that are present in the set (i.e., their position in the array has a value of 1).

6. Provide an **operator==** non-member function that determines if two sets are equal.

7. Provide an **operator~** non-member function that determines the complement of a set. Complement is computed by placing a 1 at the index where there was a 0 before and vice-versa in the complement set.

Now write a driver program to test your **IntegerSet** class. Instantiate several **IntegerSet** objects. Test that all your member functions work properly.

8.4 References

21 **Class notes**

22 **Object-Oriented Programming in C++ by** ***Robert Lafore***

23 **How to Program C++ by** ***Deitel & Deitel***

Lab No.9 Polymorphism and Virtual Functions

9.1 Objectives of the lab

Introducing the concepts of polymorphism such as

27 Basics of polymorphism
28 Virtual functions
29 Concrete classes
30 Abstract classes

9.2 Pre-Lab

9.2.1 Polymorphism

- A generic term that means 'many shapes'. In C++ the simplest form of Polymorphism is overloading of functions.
- Ability for objects of different classes to respond differently to the same function call
- Base-class pointer (or reference) calls a **virtual** function
  - C++ chooses the correct overridden function in object
- Attained by making a function virtual in base class.
- Keyword **virtual** is used in function declarator.
- Keyword **virtual** is not necessary to be used in derived classes. The overridden functions in derived classes are **virtual** automatically.

9.2.2 Example

```
#include<iostream.h>
class shape
{
public:
        virtual void draw()
        {
                cout<<"Shape class"<<endl;
        }
};
class triangle: public shape
{
public:
        void draw()
        {
                cout<<"Triangle class"<<endl;
        }
};
class rectangle: public shape
{
public:
        void draw()
        {
                cout<<"Rectangle class"<<endl;
        }
};

class circle: public shape
{
public:
        void draw()
        {
                cout<<"Circle class"<<endl;
        }
};

void main()
{
        shape   *sh;
        triangle        t;
        rectangle       r;
        circle          c;
        sh=&t; sh->draw();
        sh=&r; sh->draw();
        sh=&c;sh->draw();
}
```

Q. What is the effect of the following statement?

```
shape       s;
```

9.2.3 Abstract and Concrete Classes

- Abstract classes
  - Sole purpose is to provide a base class for other classes
  - No objects of an abstract base class can be instantiated
    - Too generic to define real objects, i.e. **TwoDimensionalShape**
  - Can have pointers and references
- Concrete classes

        - classes that can instantiate objects
- Provide specifics to make real objects , i.e. **Square**, **Circle**
- An instance of abstract class can not be created
- A derived class of an abstract base class remains abstract unless the implementation of all the pure virtual functions is not provided.
    - Derived class inherits the pure virtual function
    - Any class having a pure virtual function is virtual
    - Override the pure virtual function in derived class. (Do not provide a 0 in declarator)

9.2.4 How to make a class abstract?

- Making abstract classes
- Declare one or more virtual functions as "**pure**" by initializing the function to zero

  virtual double earnings() **const** = 0;
- A class with no pure virtual function is a concrete class.

9.3 Post-Lab

9.3.1 Activity

Define an abstract base class **shape** that includes protected data members for area and volume of a shape, public methods for computing area and volume of a shape (make the functions **virtual**), and a display function to display the information about an object. Make this class abstract.

Derive a concrete class **point** from the **shape** class. This **point** class contains two protected data members that hold the position of **point**. Provide no-argument and 2-argument constructors. Override the appropriate functions of base class.

Derive a class **Circle** publicly from the **point** class. This class has a protected data member of **radius**. Provide a no-argument constructor to initialize the fields to some fixed values. Provide a 3-argument constructor to initialize the data members of **Circle** class to the values sent from outside. Override the methods of base class as required.

Derive another class **Cylinder** from the **Circle** class. Provide a protected data member for height of cylinder. Provide a no-argument constructor for initializing the data members to default values. Provide a 4-argument constructor to initialize x- and y-coordinates, radius, and height of cylinder. Override the methods of base class.

Write a driver program to check the polymorphic behavior of this class.

9.3.2 Activity

Let us create the shape class hierarchy. Create a **Shape** class which must be abstract since we are not going to create any instance of this class. Provide data members for storing area and volume in this class. Provide a virtual function for displaying the data members of this class, a virtual function for computing volume of a shape, and a virtual function for computing area of a shape.

Derive two abstract classes **TwoDimensional** and **ThreeDimensional** from this class. Provide a virtual member function to display whether an object is 2-dimensional or 3-dimensional. Also include virtual area and volume functions so these calculations can be performed for objects of each concrete class in hierarchy.

Implement the following hierarchy and write a driver program to test the polymorphic behavior of this class.

9.3.3 Activity

Create a class called **publication** that stores the **title** (**char** array) and **price** (float) of a publication. From this class derive tow classes: **book**, which adds a page count (type **int**) and tape, which adds a playing time in minutes (type **float**). Each of the three classes should have a getdata() function to get its data form the user at the keyboard and a putdata() function to display the data.

Write a main program that creates an array of pointers to publication. In a loop, ask the user for data about a particular type of book or tape to hold the data. Put the pointer to the object in the array. When the user has finished entering the data for all books and tapes, display the resulting data for all the books and tapes entered, using a for loop and a single statement such as

```
pubarr[j]->putdata();
```

To display the data form each object in the array.

9.4 References:

24 **Class notes**

25 **Object-Oriented Programming in C++ by** ***Robert Lafore***

26 **How to Program C++ by** ***Deitel & Deitel***

Lab No.10 Lists, Stacks and Queues

10.1 Objectives of the lab:

Introducing the concepts of some linear data structures such as

31 Linear lists
32 Stack
33 Linear Queue
34 Circular Queue

10.2 Pre-Lab

10.2.1 Stack

1 Stores arbitrary objects
2 Insertions and deletions follow the Last-in-First-out scheme
3 Also known as LIFO or FILO structure
4 Main Stack Operations
   - Push (Object o): Inserts element o
   - Pop (Object o): Removes element o
5 Auxiliary stack operations
   - Top (): Returns the last inserted element without removing it.
   - Size(): Returns the number of stored elements
   - IsEmpty(): A Boolean value indicating whether no elements are stored

10.2.2 Stack Algorithms

We have two algorithms for stack:

1 Insertion called as **PUSH** operation: add an item to the top of the Stack
2 Deletion called as **POP** operation: delete an item from the stack

10.2.3 Algorithm for PUSH

```
1. [Stack is full already?]
   If Top= MAXSTK, then print: Overflow, and Return.

2. Set TOP= TOP +1.
3. SET stack (TOP)=item[Inserts ITEM in new position.]
4. End
```

10.2.4 Algorithm for POP operation

```
1. Check for underflow i.e.
   If (Top= = 0)
```

Print underflow and return

2. assign item to variable i.e.
   var =stack(TOP)
3. Decrement top by one   top=top-1
4. End

10.3 Queue

1 Stores arbitrary objects
2 Insertions and deletions follow the First In First Out scheme
3 Insertions are at the rear of the queue and deletions at the front
4 Main Queue Operations
   - Enqueue(Object o): Inserts element o, at the rear of the queue
   - Dequeue(): Removes & returns the element from the front of the queue
5 Auxiliary Queue Operations
   - Front(): Returns the element at the front without removing it
   - Size(): Returns the number of elements stored
   - IsEmpty(): Returns a boolean value indicating whether the queue is empty
6 Exceptions
   - Attempting the dequeue and Front operations on an empty queue

10.3.1 Queue Algorithms

We have two algorithms for queue:

3 Insertion called as **Enqueue** operation: add an   item to the rear of the queue
4 Deletion called as **Dequeue** operation: remove an item from front of queue

10.3.2 Algorithm for Enqueue

```
1.  [Queue is full already?]
    If   FRONT= 1 and REAR=N, or if FRONT=REAR+1, then print: Overflow, and Return.
2.  Find new value of REAR i.e.
    If FRONT= NULL, then    set FRONT +REAR+1
    Else if REAR= N, then
                Set REAR=1
    ELSE
            Set REAR=REAR+1.
3.  SET QUEUE [REAR] =item [Inserts ITEM in new position.]
4.  End
```

10.3.3 Algorithm for Dequeue

```
1.  Check for    underflow   i.e.
    If (FRONT = = 0)
      Print underflow and return
2.  assign item to variable i.e.
    var =QUEUE[FRONT]
```

```
3. [Find new value for FRONT]
   If FRONT =REAR, then
           SET FRONT = REAR= NULL
   ELSE if FRONT =N, then
               SET FRONT =1
   ELSE
               SET FRONT= FRONT +1
4. End
```

10.4 Post-Lab

10.4.1 Activity

Implement a complete array based Stack.

10.4.2 Activity

Implement a complete array based Queue.

10.4.3 Activity

Implement a circular-queue whose front and rear are connected together. Justify what is the problem in linear queue due to which we need a circular queue.

10.4.4 Activity

Implement a list using array with following operations:

1 Traversal
2 Insertion at beginning
3 Insertion at end
4 Insertion at nth location
5 Deletion from beginning
6 Deletion from end
7 Deletion from nth location
8 Searching an element

10.4.5 Activity

Create a structure for **book** that contains information about title, price, edition, and no of pages of the book. Use your program of activity 1 to push and pop books on a stack.

10.4.6 Activity

Let us model the flow of customers in a queue. Create a structure called **Person** that contains information about first name, last name, age, sex, and address of a person. Use your program of activity

2 to enqueue and dequeue Persons from a queue.

10.5 References

27 **Class notes**

28 **Data Structures, Schaum's outline series**

29 **Data structures and algorithms in C++ by** ***Micheal T. Goodrich and Roberto Tamassia***

Lab No.11 Linked Lists

11.1 Objectives of the lab:

Introducing the concepts of some linear data structures such as

35 Single Linked lists
36 Operation on single linked list
37 Stack and Queue using linked list

11.2 Pre-Lab

11.2.1 Singly Linked List

38 A data structure consisting of a sequence of nodes
39 Each node contains
   a. Data or element
   b. Pointer to next node
40 Advantage of Linked Lists
   c. Order of the linked items may be different to the order that the data items are stored in memory or on disk
   d. Allow insertion and removal of nodes at any point in linear time
41 Disadvantages
   e. Do not allow random access

11.2.2 Queue Algorithms

11.2.3 Algorithm for Searching

```
1. PTR = Start
2. Repeat step 3 while PTR is not equal NULL
3.   If ITEM = INFO [PTR]
         Loc = PTR and exit
     Else
         PTR = LINK [PTR].[PTR]
3. Loc = NULL
4. Exit
```

11.2.4 Algorithm for Insertion at start

```
1. If AVAIL = NULL then overflow
2. NEW = AVAIL and AVAIL = LINK[AVAIL]
3. INFO[NEW] = ITEM
4. LINK[NEW] = START
5. START = NEW
6. EXIT
```

11.2.5 Algorithm for Insertion in a sorted list

```
1. If START = NULL then LOC = NULL and return
2. If ITEM < INFO[START] then LOC = NULL and return
3. SAVE = START and PRT = LINK[START]
4. Repeat steps 5 and 6 while PTR is not equal to NULL
5. if ITEM < INFO[PTR]
       LOC = SAVE and return

6. SAVE = PTR, PTR = LINK[PTR]
7. LOC = SAVE
8. return
```

11.2.6 Algorithm for Deletion after a particular node

```
1. if LOCP = NULL
       START = LINK [START]
       Else
       LINK [LOCP] = LINK [LOC]
2. LINK[LOC] = AVAIL and AVAIL = LOC
3. exit
```

11.3 Post-Lab

11.3.1 Activity

Implement complete linked list containing the following operations:

9 Traversal
10 Insertion at beginning
11 Insertion at end
12 Insertion at nth location
13 Deletion from beginning
14 Deletion from end
15 Deletion from nth location
16 Searching an element

11.3.2 Activity

Implement stack using single linked list.

11.3.3 Activity

Implement queue using single linked list.

11.3.4 Activity

Write a program to remove duplicates from a singly linked list.

11.3.5 Activity

Write a program that creates a linked list object of 10 characters and then creates a second list object containing a copy of the first list, but in reverse order.

11.3.6 Activity

Write a program that concatenates two linked list objects of characters. The program should include function concatenate, which takes references to both list objects as arguments and concatenates the second list to the first list.

11.3.7 Activity

Write a program that uses a stack object to determine if a string is a palindrome (i.e., the string is spelled identically backwards and forwards). The program should ignore spaces and punctuation.

11.4 References:

30 **Class notes**

31 **Data Structures, Schaum's outline series**

32 **Data structures and algorithms in C++ by *Micheal T. Goodrich and Roberto Tamassia***

Lab No.12 Applications of Stacks

12.1 Objectives of the lab

Introducing the applications of stack such as:

42 Symbol balancing
43 Infix to postfix conversion
44 Expression evaluation

12.2 Pre-Lab

12.2.1 Algorithm for Symbol balancing

1 Ignore other characters, just check for balancing of parentheses, brackets, & braces.

2 A simple Algorithm

   - Make an empty Stack
   - Read characters until end of file
   - If the character is an opening symbol, push it onto the stack.
   - If it is a closing symbol, then if the stack is empty report an error.
   - Otherwise pop the stack. If the popped symbol is not the corresponding opening symbol, then report an error.
   - At the end of file if the stack is not empty, report an error

12.2.2 Algorithm for Infix to postfix conversion

1 Start with an initially empty stack

2 When an operand is read, it is immediately placed onto the output

3 Operators are not immediately output but pushed on a stack

4 If we see a right parentheses, we pop the stack and output the symbols until we encounter a left parentheses which is popped but not output

5 If we see any other symbol (‘+’, ‘*’, ‘(‘), then we pop entries from the stack until we find an entry of lower priority.

12.2.3 Algorithm for expression evaluation

For each character C in a given string

{

```
        if C is an operand
            push C onto stack;
        else // C is an operator
        {
            pop item from the stack, and store in Opr2;
            pop item from stack, and store in Opr1;
            result = Opr1 C Opr2, using C as an operator;
            push result onto stack;
        }
    }
```

12.3 Post-Lab

12.3.1 Activity

Implement the algorithm of symbol balancing using array-based as well as single linked list.

12.3.2 Activity

Implement the algorithm for infix to postfix conversion array-based as well as single linked list.

12.3.3 Activity

Implement the algorithm for expression evaluation array-based as well as single linked list.

12.4 References

33 **Class notes**
34 **Data Structures, Schaum's outline series**
35 **Data structures and algorithms in C++ by *Micheal T. Goodrich and Roberto Tamassia***

Lab No.13 Doubly Linked Lists

13.1 Objectives of the lab

Introducing the concepts of some double linked list such as

45 Basics of double linked list
46 Operations on double linked list

13.2 Pre-Lab

13.2.1 Double Linked list

1 A doubly linked list provides a natural implementation of the List ADT.
2 Nodes implement position and store
   - Element
   - Link to previous node
   - Link to next node
3 Special Trailer and Header nodes
   - Header node to traverse in one direction
   - Trailer node to help traverse in the other direction
4 How to keep track of the position index?
   - Starting from the header, keep a counter

13.3 Post-Lab

13.3.1 Activity

Implement complete double linked list containing the following operations:

17 Traversal from front to back
18 Traversal from back to front
19 Insertion at beginning
20 Insertion at end
21 Insertion at nth location
22 Deletion from beginning
23 Deletion from end
24 Deletion from nth location
25 Searching an element

13.4 References:

36 Class notes
37 Data Structures, Schaum's outline series
38 **Data structures and algorithms in C++ by *Micheal T. Goodrich and Roberto***

Tamassia

Lab No.14 Circularly Linked Lists and their Applications

14.1 Objectives of the lab:

Introducing the concepts of circular linked list such as

47 Basics of circular linked lists
48 Operations on circular linked lists
49 Applications of circular linked lists

14.2 Pre-Lab

14.2.1 Circular linked list

1 The first and final nodes are linked together.

2 For traversal begin at any node

  - Maintain pointer to at least one node

3 Singly-circularly-linked lists

  - Each node has one link
  - Similar to an ordinary singly-linked list except that the next link of the last node points to the first node

4 Doubly-circularly-linked lists

  - Each node has two links
  - Similar to an ordinary doubly-linked list except that the next link of the last node points to the first node

14.3 Post-Lab

14.3.1 Activity

Implement complete circular single linked list containing the following operations:

26 Traversal
27 Insertion at beginning
28 Insertion at end
29 Insertion at nth location
30 Deletion from beginning
31 Deletion from end
32 Deletion from nth location
33 Searching an element

14.3.2 Activity 14.2

Implement complete circular double linked list containing the following operations:

34 Traversal
35 Insertion at beginning
36 Insertion at end
37 Insertion at nth location
38 Deletion from beginning
39 Deletion from end
40 Deletion from nth location
41 Searching an element

14.3.3 Activity, Josephus Problem (Election Process)

A group of people are standing in a circle. Start at a predetermined position and count around the circle n-times. Once you reach the nth person, take that person out and close the circle. Then count around the circle the same n-number of times and repeat the process, until one person is left. That person wins the election.

Write a program for implementation of this problem statement.

14.4 References:

39 Class notes
40 Data Structures, Schaum's outline series
41 **Data structures and algorithms in C++ by *Micheal T. Goodrich and Roberto Tamassia***

Lab No.15 Trees and their Operations

15.1 Objectives of the lab

Introducing the concepts of trees such as

50 Basics of trees
51 Operations on trees

15.2 Pre-Lab

15.2.1 Trees

1 Represent hierarchical relationship between elements
- File structure
- Family tree
- Organization tree

15.2.2 Binary Trees

1 Finite (possibly empty) collection of elements
2 A nonempty binary tree has a root element
3 The remaining elements (if any) are partitioned into two binary trees
4 These are called the left and right subtrees of the binary tree

15.2.3 Binary Search Trees

1 A binary tree is called Binary Search Tree if at each node N of tree
1. The value at N is greater than every value in the left subtree of N
2. The value at N is less than every value in the right subtree of N

15.2.4 Tree Traversals

1 **Preorder Traversal**
- Process the root
- Traverse the left subtree of root
- Traverse the right subtree of root

2 **Inorder Traversal**
- Process the left subtree of root
- Process the root
- Traverse the right subtree of root

3 **Post-order Traversal**
- Process the left subtree of root
- Traverse the right subtree of root
- Process the root

15.2.5 Tree Algorithms

Algorithm for Insertion

1. Input value to insert in N
2. Search value N in the tree
3. If value N exists then print "Number already exists" and return. Otherwise insert the value at the point the search stops.

Algorithm for Searching and sorting

1 Sorting
  - In-order traversal of BST results in sorted retrieval of elements

2 Searching
  - a) compare value with the root node N of the tree
    1. If value<N, proceed to the left child of N
    2. If value >N, proceed to the right child of N
  - b) Repeat (a) until one the following occurs
    3. Value=N. In this case, search is successful.
    4. An empty subtree is encountered. No element matched an search is unsuccessful.

15.3 Post-Lab

15.3.1 Activity

Implement a binary search tree with following operations

2 Insertion

3 Deletion

4 Traversal (using recursion)
  - 3. Pre-order
  - 4. Post-order
  - 5. In-order

5 Searching

6 Sorting

15.4 References:

42 Class notes

43 Data Structures, Schaum's outline series

44 **Data structures and algorithms in C++ by *Micheal T. Goodrich and Roberto Tamassia***

Lab No.16 Tree Traversal using Stacks

16.1 Objectives of the lab:

Introducing some more concepts of trees such as

52 Traversal of tree using stack-based approach

16.2 Pre-Lab

16.2.1 Algorithm for pre-order traversal

1 Stack is used to hold the tree nodes from time-to-time

2 Initially:

- push NULL on stack
- CUR=root

(a) Follow the left path of CUR, process each node on the path, and push each right child (if any) onto the stack. Traversing ends when a node with no left child is encountered.

(b) Pop and assign to CUR the top element of stack. If CUR!=NULL, then repeat step (a), otherwise Exit.

16.2.2 Algorithm for in-order traversal

1 Stack is used to hold the tree nodes from time-to-time

2 Initially:

- push NULL on stack
- CUR=root

(a) Follow the left path of CUR and push each node onto the stack. Stop when a node with no left child is pushed onto the stack.

(b) Pop and process the nodes on stack. If a NULL is popped, Exit. If a node with a right child is processed, set CUR=CUR->right and repeat step (a).

16.2.3 Algorithm for post-order traversal

1 Stack is used to hold the tree nodes from time-to-time

2 Initially:

- push NULL on stack
- CUR=root
  - (a) Follow the left path of CUR and push each node onto the stack. If the node has a right child R, push –R onto the stack.
  - (b) Pop and process positive nodes on stack. If a NULL is popped, Exit. If a node with a right child is processed, set CUR= -R and repeat step (a).

16.3 Post-Lab

16.3.1 Activity

Implement the stack-based algorithms for

1. In-order traversal
2. Pre-order traversal
3. Post-order traversal

16.4 References:

45 Class notes
46 Data Structures, Schaum's outline series
47 **Data structures and algorithms in C++ by *Micheal T. Goodrich and Roberto Tamassia***