COMSATS Institute of Information Technology (CIIT)

Islamabad

Department of Computer Science

Laboratory Manual

Course: CSC241 Object Oriented Programming

COMSATS Institute of Information Technology (CIIT) Islamabad

Document Version History

Version	**Date**	**Prepared By**	**Reviewed By**	**Approved By**
1.0	17/02/2013	Saima Khan, Nusrat Shaheen, Dr. Najam-us-Saqib, Zara Hamid, Najam-un-Nisa, Mr. Salman Aslam, Sehrish Khan Sajida Kalsoom maazbin_izhar	Dr.Farhana Jabeen, Dr. Amir Hayat, Ms. Sehresh Khan, Mr. Ubaid ur Rehman	

COMSATS Institute of Information Technology (CIIT) Islamabad

Table of Contents

Lab 01 – *Review of Basic Programming Concepts*

1. Objectives

Objective of this lab is to review concepts related to structures, pointers and passing function arguments by value and by reference, so that students can easily use them in the upcoming labs.

2. Outcomes

2.1 The student will be able to declare and use structures.

2.2 The student will be able to pass arguments to functions both by value and by reference.

2.3 The student will be able to declare, initialize and use pointers.

3. Introduction

3.1 Structure:

A structure is a user defined data type. Through structures you have the ability to define a new type of data considerably more complex than the types we have been using. A structure is a combination of several different data types. It is declared by using the keyword **struct** followed by the structure name.

Syntax:

```
struct struct_name
{
  Data_type1 member_name1;
  Data_type2 member_name2;
  Data_type3 member_name3;
} object_name;
```

3.2 Passing arguments by Value:

A parameter passing mechanism in which the value of actual parameter is copied to formal parameters of called functions is known as pass by value. If the function makes any change in formal parameter, it does not affect the values of actual parameter. It is the default mechanism for passing parameters to functions.

3.3 Passing arguments by Reference:

A parameter passing mechanism in which the address of actual parameter is passed to the called function is known as pass by reference. In pass by reference, we declare the function parameters as references rather than normal variables. The formal parameter is not created separately in the memory. Formal parameter becomes a second name of actual parameter. It means that single memory is shared between actual parameter and formal parameter. If the called function makes any change in formal parameter, the change is also visible in actual parameter.

3.4 Pointers

Pointers are a type of variable that allow you to specify the address of a variable. They provide a convenient means of passing arguments to functions and for referring to more complex data types such as structures.

Syntax:

Data_type *var;

3.5 Dereference Operator

It is the operator used to access the value of the variable whose memory address is stored in pointer. It is denoted by asterisk (*). It is also used to input the value in the variable and process the data stored in the variable.

The following statement is used to store value in the first variable by the help of dereference operator.

```
*myptr=10;
```

4. Examples

4.1 Write a program that declares a structure to store date. Declare an instance of this structure to represent date of birth. The program should read the day, month and year values of birth date and display date of birth in dd/mm/yy format.

```
#include<iostream.h>
#include <conio.h>
struct Date
{
   int day;
   int month;
   int year;
};

void main( )
{
   Date birthDate;
   cout<<"Enter day of birth";        //enter day
   cin>>birthDate.day;                //display day
   cout<<"Enter month of birth";      //enter month
   cin>>birthDate.month;              //display month
   cout<<"Enter year of birth";       //enter year
   cin>>birthDate.year;               //display year
   cout<<"your date of birth is: " << birthDate.day <<"/"<< birthDate.month<<"/"
   << birthDate.year<<"\n";
   getch();
}
```

OUTPUT:

```
Enter day of birth: 12
Enter month of birth: 3
Enter year of birth: 98
Your date of birth is: 12/3/98
```

4.2 A simple example to show how arguments are passed by value to a function.

```
#include <iostream.h>
int addition (int numA, int numB)     // numA and numB are formal parameters
{
  int numC;
  numC=numA+numB;          //add numA and numB and store the result in numC
  return (numC);
}
void main ( )
{   int sum;
    sum = addition (5,3);                          //function call
    cout << "The result is " <<sum;
}
```

OUTPUT:

```
The result is 8
```

4.3 An example to demonstrate how arguments are passed by reference

```
#include <iostream.h>
void duplicate (int& var1, int& var2, int& var3) //formal parameters
{
   var1*=2;       //change value of both actual and formal parameter
   var2*=2;
   var3*=2;
}
int main ( )
{
    int x=1, y=3, z=7;       //actual parameters
    duplicate (x, y, z);      // pass actual parameters to function
    cout << "x=" << x << ", y=" << y << ", z=" << z;
    return 0;
}
```

OUTPUT:

```
x=2, y=6, z=14
```

4.4 An example which shows how pointers can be used to manipulate variables (memory locations) to which they point.

```
#include <iostream.h>
int main ( )
{
 int first, second;
 int *myptr;
 myptr = &first;
```

```
  *myptr = 10;
  myptr = &second;
  *myptr = 20;
  cout << "First value is " << first << endl;
  cout << "Second value is " << second << endl;
  return 0;
}
```

OUTPUT:

```
First value is 10
Second value is 20
```

5. Lab Tasks

5.1. Write a program that declares a structure to store book Id, price and pages of a book. The structure should include functions to assign user defined values to each book and display the record of most costly book.

5.2. Write a program to take the values of two integers and use pointers to add 10 to the value of each integer.

5.3. Write a function that swaps the values of two integer variables

- **a.** using pass by value
- **b.** and pass by reference and see their differences

6. Home Tasks

6.1. There is a structure called **employee** that holds information like employee code, name, date of joining. Write a program to create an array of the structure and enter some data into it. Then ask the user to enter current date. Display the names of those employees whose tenure is 3 or more than 3 years according to the given current date.

6.2. Write a function to sort data (in increasing order) in an array using

- a. pass by value
- b. and pass by reference.

6.3. Write a program that inputs a string value from the user and displays it in reverse using pointer.

Lab 02 – *Classes and Data Abstraction*

1. Objective:

Objective of this lab is to understand the importance of classes and construction of objects using classes.

2. Outcomes:

2.1 The student will be able to declare classes and objects.

2.2 The student will be able to declare member functions and member variables of a class.

2.3 He will understand the importance and use of constructor.

2.4 He will understand the use of destructor in class.

3. Introduction

3.1 Class & Object:

The fundamental idea behind object-oriented languages is to combine into a single unit both data and the functions that operate on that data. Such a unit is called an object. A class serves as a plan, or blueprint. It specifies what data and what functions will be included in objects of that class. An object is often called an "instance" of a class

Syntax:

Classes are generally declared using the keyword class, with the following format:

```
class class_name {
    access_specifier_1:
          member1;
    access_specifier_2:
         member2;
          …….
 } object_names;
```

3.2 Data Abstraction:

Abstraction is the process of recognizing and focusing on important characteristics of a situation or object and leaving/filtering out the un-wanted characteristics of that situation or object. For example a person will be viewed differently by a doctor and an employer.

- A doctor sees the person as patient. Thus he is interested in name, height, weight, age, blood group, previous or existing diseases etc of a person
- An employer sees a person as an employee. Therefore employer is interested in name, age, health, degree of study, work experience etc of a person.

3.3 Member Functions and Variables:

Member variables represent the characteristics of the object and member functions represent the behavior of the object. For example length & width are the member variables of class Rectangle and set_values(int,int), area() are the member functions.

3.4 Constructors:

It is a special function that is automatically executed when an object of that class is created. It has no return type and has the same name as that of the class. It is normally defined in classes to initialize data members.

Syntax:

```
class_name( )
{
    // Constructor body
 }
```

3.5 Destructors:

It is a special function that is automatically executed when an object of that class is destroyed. It has no return type and has the same name as that of the class preceded by tild (~) character. Unlike constructors, destructors cannot take arguments.

Syntax:

```
~ class_name ()
{
   // Destructor body
 }
```

4. Examples

4.1 The following example shows the declaration of class Rectangle. It also demonstrates how member functions can be defined both inside and outside the class and how objects of class rectangle can be used to access its member functions.

```
#include <iostream.h>
class Rectangle {
   int length, width;
   public:
   void set_values (int,int);       //set the values of length and width
   int area () {return (length*width);}
};
void Rectangle::set_values (int l, int w) {
  length = l;
  width = w;
}
int main () {
  Rectangle rect;
  rect.set_values (3,4); //pass values to set length and width
  cout << "area =" << rect.area( );
  return 0;
}
```

OUTPUT:

```
area= 12
```

4.2 This example demonstrates how constructors and destructors work

```
class counter
   {
      private:
            int count;
      public:
            counter()  { count=0;  cout<<"I am a constructor"<<endl; }
            ~counter()  { cout<<"I am a destructor"<<endl; }
   };
void main( )
   {
   counter c;
   getch();
   }
```

5. **Lab Tasks**

5.1. Write a class that displays a simple message "I am object no. __", on the screen whenever an object of that class is created.

5.2. Create a class that imitates part of the functionality of the basic data type int, call the class Int. The only data in this class is an integer variable. Include member functions to initialize an Int to zero, to initialize it to an integer value and to display it. Write a program that exercises this class by creating an Int variable and calling its member functions.

6. Home Assignments:

6.1 Write a program to calculate the number of objects created and destroyed for the counter class.

6.2 Create a class named time, the data members are hours, minutes and seconds. Write a function to read the data members supplied by the user, write a function to display the data members in standard (24) hour and also in (12) hour format.

6.3 Write a class marks with three data members to store three marks. Write three member functions, set_marks() to input marks, sum() to calculate and return the sum and avg() to calculate and return average marks.

Lab 03 – *Class Scope and Accessing Class Members*

1. Objectives

The objective of this lab is to teach the students, the scope of class data members and its member functions.

2. Outcome

At the end of this lab student will be familiar with the accessing rules of class data members and member functions

3. Introduction

In object oriented programming, methods and variables have various scope. Scope means that the method or variable may or may not be directly accessible to other objects or classes. Classes that do not have instances may be accessible to the system.

One of the techniques in object-oriented programming is encapsulation. It concerns the hiding of data in a class and making them available only through its methods. In this way the chance of making accidental mistakes in changing values is minimized. C++ allows you to control access to classes, methods, and fields via so-called access modifiers. The access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes. To take advantage of encapsulation, you should minimize access whenever possible.

3.1. Class Scope

Class variables and class methods are associated with a class. An instance of the class (object) is not required to use these variables or methods. Class methods cannot access instance variables or methods, only class variables and methods.

3.2. Instance Scope

Instance variables and instance methods are associated with a specific object. They can access class variables and methods.

3.3. Private Scope

Private variables and private methods are only accessible to the object they are contained in.

3.4. Protected Scope

Protected variables and protected methods are accessible by the class they are in and inheriting classes (sub classes) only.

3.5. Public Scope

Public variables and public methods are accessible outside the object they are contained in. They are accessible to any other object.

3.6. Encapsulation

The process of providing a public interface to interact with the object while hiding other information inside the object is called encapsulation.

4. Examples

The following program illustrates the usage of objects and classes in C++:

// The program uses public member functions to input two private numbers, add two private numbers and display the result

```
#include<iostream>

using namespace std;

class add //Specifies the class
{
private:
    int iNum1, iNum2, iNum3; //Member data

public:
    void input(int iVar1, int iVar2) //Member function
    {
       cout<<"Functions to assign values to the member data"<<endl;
       iNum1=iVar1;
       iNum2=iVar2;
    }
    void sum(void) //Member function
    {
       cout<<"Functions to find the sum of two numbers"<<endl;
       iNum3=iNum1+iNum2;
    }
    void disp(void) //Member function
    {
       cout<<"The sum of the two numbers is "<<iNum3<<endl;
    }
};
/////////main function of the program//////////
void main()
{
    add A1;
    int iX, iY;
    cout<<"Input two numbers"<<endl;
    cin>>iX;
    cin>>iY;
    A1.input(iX, iY);
    A1.sum();
    A1.disp();
    system("pause");
}
```

5. Lab Tasks

5.1. Code the example given above and check the errors if you try to access the private data members in main() function.

5.2. Modify the above task by making the scope of public member functions as private. Create access functions in public scope to access private member functions from main().

5.3. Code the example given above and include a private constructor in the class. Create objects of this class. Test the code and write down how the constructor will be called or unable to be called?.

6. Home Tasks

6.1. Create a class of subtraction having two private data members. Create class methods to get data from users and for subtraction of data members. Use appropriate access modifiers for class methods.

7. References

[i] http://www.tutorialspoint.com/cplusplus/cpp_class_access_modifiers.htm

[ii] http://www.learncpp.com/cpp-tutorial/83-public-vs-private-access-specifiers/

Lab 04 – *Objects and Methods*

1. Objectives

The objective of this lab is to teach the students, how the objects can be passed to and returned from the functions, how the functions can define outside the class and inline functions.

2. Outcome

Following Outcomes will be achieved from this lab work.

2.1. At the end of this lab student will be able to pass objects to the functions.

2.2. At the end of this lab student will be able to return objects from the functions.

2.3. At the end of this lab student will be able to define functions outside the class if declared inside.

2.4. At the end of this lab student will be able to understand and use inline functions.

3. Introduction

3.1. Passing to and Returning Objects from function

C++ objects are normally passed by reference to avoid expensive duplication and to let other functions use the same object as the calling function. A member function is always given access to the object for which it was called: the object connected to it with the dot operator. But it may be able to access other objects passed to them. A member function also can return an object. From function any new object created within the function or a nameless object can be returned.

The code below demonstrate simple object passing to and returning from a member function

```
#include<iostream>

using namespace std;

class Complex
{
    private:
                    double real, imag;
    public:

   Complex()    // Default Constructor
   {
                    real = 0.0;
                    imag = 0.0;
   }
   // Two argument Constructor
   Complex (double r, double im)
   {
      real = r;
                    imag = im;
   }
```

```
    void show()
    {
        cout<<real<<"+"<<imag<<"i"<<endl;
    }
    Complex Add (Complex& b)
    {
        return Complex(real + b.real, imag+ b.imag);
    }
};

void main()
{
   Complex C1(11, 2.3);
   Complex C2(9, 2.3);
   Complex C3;
   C3 = C1.Add(C2);
   C3.show();
   system("pause");
}
```

3.2. Defining Member Functions Outside The Class

Member functions of a class can be defined outside the class. For defining the member functions outside the class the functions should be declared within the class definition then the functions can be defined outside the class using scope resolution operator(::). The sample code is given below.

class myclass
{
  void function(); //declaration
};
	void myclass::function()
	{
	  //function definition outside the class
	}

3.3. Inline Functions

A member function that is defined inside its class member list is called an inline member function. Member functions containing a few lines of code are usually declared inline. If you define a member function outside of its class definition, it must appear in a namespace scope enclosing the class definition. You must also qualify the member function name using the scope resolution (::) operator.

An equivalent way to declare an inline member function is to either declare it in the class with the inline keyword (and define the function outside of its class) or to define it outside of the class declaration using the inline keyword.

In the following example, member function Y::f() is an inline member function:

class Y {
private:
  char* a;
	class Y {
	private:
	  char* a;

public:
  char* f() { return a; }
};
	public:
	  char* f();
	};
	
	inline char* Y::f() { return a; }

4. Examples

The example below demonstrates the aspects of constructor overloading, defining member functions outside the class, and most importantly objects as function arguments. This program starts with a distance dist2 set to an initial value and adds to it a distance dist1, whose value is supplied by the user, to obtain the sum of the distances. It then displays all three distances

```
#include<iostream>

using namespace std;

class Distance //English Distance class
{
private:
      int feet;
      float inches;
public:
      //constructor (no args)
      Distance() : feet(0), inches(0.0)
      { }
      //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
      { }

      void getdist() //get length from user
      {
           cout<<"\nEnter feet: ";
           cin>>feet;
           cout<<"Enter inches: ";
           cin>>inches;
      }
      inline void showdist(); //display distance

      void add_dist( Distance, Distance ); //declaration
};
//---------------------------------------------------------------
//definition of inline function which display distance

inline void Distance::showdist()
{
      cout<<feet<<"\""<<inches<<"\'";
}

//add lengths d2 and d3
void Distance::add_dist(Distance d2, Distance d3)
{
      inches = d2.inches + d3.inches; //add the inches
      feet = 0; //(for possible carry)
      if(inches >= 12.0) //if total exceeds 12.0,
      {
           //then decrease inches
           inches -= 12.0; //by 12.0 and
           feet++; //increase feet
      } //by 1
      feet += d2.feet + d3.feet; //add the feet
}
////////////////////////////////////////////////////////////////
```

```
void main()
{
      Distance dist1, dist3; //define two Distance Objects
      Distance dist2(11, 6.25); //define and initialize dist2
      dist1.getdist(); //get dist1 from user
      dist3.add_dist(dist1, dist2); //dist3 = dist1 + dist2
      //display all lengths
      cout<<"\ndist1 = ";
      dist1.showdist();
      cout<<"\ndist2 = ";
      dist2.showdist();
      cout<<"\ndist3 = ";
      dist3.showdist();
      cout<<endl;
      system("pause");
}
```

5. Lab Tasks

5.1. Code the example of complex class given above and include the functions for addition, subtraction and multiplication of objects of complex class and return the object containing result. Test all the functions in main.

5.2. Modify the above task such that; define the member function show() outside the class and also define it as inline.

5.3. Modify the task (5.1) by defining all the member functions outside the class definition.

5.4. Test the distance class example given above.

6. Home Tasks

6.1. Modify the Distance class example of lab task (5.4) by including functions for subtraction and multiplication of distance class objects like addition.

7. References

[iii] http://www.tutorialspoint.com/cplusplus/cpp_class_access_modifiers.htm
[iv] http://www.learncpp.com/cpp-tutorial/83-public-vs-private-access-specifiers/

Lab 05 – *Constructor and Function Overloading*

1. Objectives

This Lab will help students in understanding the concept and utility of constructors, how they are helpful in object initialization, different forms of a constructor, method overloading as well as *const* keyword is also covered in the lab.

2. Outcome

After completion of this Lab students know what is a constructor, when it is called and how it can be called and get benefit of method overloading where ever required. Student will also know what are a constant function, constant member and constant objects.

3. Introduction

Initializing the object private data member can be performed with a help of a public member function. With this approach you have to call that function after creating each object of that class. It will be a convenient approach if and object's private data members can be initialize itself when it is created, without requiring a separate call to a member function. Automatic initialization is carried out using a special member function called a **constructor**. A constructor is a member function that is executed automatically whenever an object is created. A constructor having no argument is called a default constructor.

Function overloading is a concept where you can have multiple functions with same name. When a call to one of such function take place the compiler can differentiate as follow

- Through number of parameters

  For example you can have following functions in you program

  1. void display( );
  2. void display(int x);
  3. void display(int x, int y);

  Now if you call display(3, 2); the compiler knows that 3rd function is called because 2 arguments are pass to that function
- Through types of parameters

  If No. of parameters are same than the compiler can differentiate through types of parameter. For example you can have following function in your program.

  1. int add(int y, int x);
  2. float add(float a, float b);

  now if you call add (3.6, 5.78); then compiler know that 2nd function is called because 2 float types arguments are passed to that function

A constructor can be overloaded. This means that we can have a constructor with no argument (called a default constructor), a constructor with one argument and a constructor with more than one argument if required. This is called constructor overloading.

Some objects need to be modifiable and some do not. Keyword const (constant) is used to specify that an object is not modifiable. Any attempt to modify the const object results in compilation error. For example the statement below declares a const object noon of class Time and initializes it to 12 noon

```
const  Time noon (12, 0, 0)
```

Any member function of a class can be constant if that function will not change the value of private data member of invoking object.

Constant data member of a class must be initialized as soon as the object is created. That is it must be initialize in constructor initializer list.

4. Examples

4.1 Function overloading

```
#include<iostream>
using namespace std;
/* function prototype */
void display(int);
void display(int, int);
void display(double, double);
/* main function */
main(){
   display(100);   // call to display (int);
   display(50, 60);  // call to display(int, int);
   display(200.45, 205.75); // call to display(float, float);
   system("pause");
}
void display(int x){
   cout<<"display with single int argument"<<endl;
   cout<<"value = "<<x<<endl;
}

void display(int x, int y){
   cout<<"display with double int argument"<<endl;
   cout<<"value 1 = "<<x<<", value 2 = "<<y<<endl;
}
void display(double x, double y){
   cout<<"display with double float argument"<<endl;
   cout<<"value 1 = "<<x<<", value 2 = "<<y<<endl;
}
```

<u>**Program output**</u>

```
display with single int argument
value = 100
display with double int argument
value 1 = 50, value 2 = 60
display with double float argument
value 1 = 200.45, value 2 = 205.75
Press any key to continue . . .
```

4.2 Constructor example

```
#include <iostream>
using namespace std;
class Distance //English Distance class
{
        private:
                int feet;
                float inches;
        public:
           Distance(){
                cout<<"default constructor"<<endl;
                feet = 0;  inches = 0;
             }
             Distance(float mtrs){
                                 // this constructor convert meters to feet and inches
               cout<<"one argument constructor"<<endl;
               float ft = mtrs * 3.28084; // convertinf meters to feet
               feet = (int) ft;  // extracting int part from ft
               inches = (ft - feet)*12; // converting decimal part of ft into inches
             }
            Distance (int f, float i){
              cout<<"two argument constructor"<<endl;
              feet = f;  inches = i;
            }
          void setdist(int ft, float in) {  //set Distance to args
              feet = ft;
              inches = in;
          }
          void getdist() {  //get length from user
             cout << "\nEnter feet: "; cin >> feet;
             cout << "Enter inches: "; cin >> inches;
          }
          void initialize( ) {
              feet = 0;
              inches = 0;
          }
          void showdist( ) {  //display distance
               cout << "feet = "<< feet <<"\t inches = "<<inches<<endl;
          }
};

main()
{
    /*two objects are created so default constructor is called two times*/
    Distance dist1, dist2; // objects data member are initialize by default
constructor
   /*      if default constructor is not present then
           we have to call initialize function with each
            object to initialize its private data member      */
   //dist1.initialize(); // in comments because we have
   //dist2.initialize(); // default constructor

   dist1.setdist(11, 6.25); //set dist1 values using setdist function
   Distance dist3(3, 5.75); // here 2 argument constructor will be called
                            // that initialize dist3 data member with 3 and 5.75 values
   Distance dist4(1); // here 1 argument construct will be called

   cout << "dist1 : "; dist1.showdist();
```

```
    cout << "dist2 : "; dist2.showdist();
    cout << "dist3 : "; dist3.showdist();
    cout << "dist4 : "; dist4.showdist();
    system("pause");
}
```

Program output

```
default constructor
default constructor
two argument constructor
one argument constructor
dist1 : feet = 11       inches = 6.25
dist2 : feet = 0        inches = 0
dist3 : feet = 3        inches = 5.75
dist4 : feet = 3        inches = 3.37008
Press any key to continue . . .
```

4.3 Example program with const keyword

```
#include<iostream>
using namespace std;
class Time
{
  private:
    int hour;   // 0 - 23 (24-hour clock format)
    int minute; // 0 - 59
    int second; // 0 - 59
    const int id;
  public:
    Time( int h , int m , int s, int i ): id(i){
      //id  = i;
      setTime( h, m, s);
    }
    // set functions
    void setTime( int h, int m, int s){ // set time
      setHour( h );
      setMinute( m );
      setSecond( s );
    }
    void setHour( int h){ // set hour
      hour = ( h >= 0 && h < 24 ) ? h : 0;
    }
    void setMinute( int m){ // set minute
      minute = ( m >= 0 && m < 60 ) ? m : 0;
    }
    void setSecond( int s){ // set second
      second = ( s >= 0 && s < 60 ) ? s : 0;
    }
    // get functions (normally declared const)
    int getHour() const{ // return hour
```

```
      return hour;
    }
    int getMinute() const{ // return minute
      //minute = 10;  // Error cannot assign value to data member in const function
      return minute;
    }
    int getSecond() const{ // return seco
      return second;
    }
    // print functions (normally declared const)
    void printTime() const{
      cout<<hour<<" : "<<minute<<" : "<<second<<endl;
    }
};
main(){
        Time wakeUp( 5, 30, 30, 1 ); // non-constant object
        const Time noon( 12, 0, 0, 2 ); // constant object
                                   // OBJECT     MEMBER FUNCTION
        wakeUp.setHour( 4 );       // non-const  non-const
        //noon.setHour( 14 );      // const      non-const  --> Error
        wakeUp.printTime();        // non-const  const
        noon.printTime();          // const      const
        system("pause");
}
```

Program output

```
4 : 30 : 30
12 : 0 : 0
Press any key to continue . . .
```

5. Lab Tasks

5.1.

Area of a circle is $\pi \times r^2$ where r = radius

Area of a triangle is ½ × b × h where b = base, h = height

Area of a rectangle is w × h w = width, h = height

Area of a square is a^2 where a = length of side

Write four different functions with same name that is Area to calculate the area of circle, triangle, rectangle and square.

5.2. Write a definition of a Counter class having one private data member count of integer type. This class has following functions

- void inc_count( ); // will increment the value of count by 1
- int get_count ( ); // will return the value of count

this class has two contructor

- Counter( ); // that initialize count by 0
- Counter (int i); // that initialize the count by i

Create two objects of Counter class. Write a cout statement in constructor and then check whether that statement appear when two object are created. Then increment object 1 3 times and increment object 2 4 times and display their count values.

5.3 Write a definition of class named Race. It has following private data member

- carNo (int)
- driverID (int)
- carModel (int)

The class has one constructor Race (int, int, int) that initializes the values of carNo, driverID and carModel. Race class has following member functions

- void InputValues( ) // this will be used to input values of data member of Book object from user
- void setValues(int cn, int di, int cm); // it will assign values of cn, di and cm to carNo, driverID and carModel respectively
- void display( ); // it displays the value of private

Create two object of Rave class. Assign values using InputValues and setValues function and display them using display function.

5.4 Write a definition of a distance class as shown in the example 4.2 above. Make all the appropriate function constant. Include a constant data member called id of integer type.

Create two object constant and non-constant. Assign values and display them. Also check what happens

- If you try to modify private data member of an object from the definition of const function
- If you try to modify the private data member of const object from the definition of non-constant function.

6. Home Tasks

Write a definition of class named Date that contains three elements the month, the day of the month, and the year, all of type int.

- Write two constructors, a default constructor (that initialize each data element of object with zero) and a constructor that takes three parameters (the month, the day of the month, and the year) and initialize the data member of the object with these parameters.
- Write a function void printDate() that displays the data elements of the object.
- Write a function void setDate(int, int, int) that takes three parameters (he month, the day of the month, and the year) and initialize the data member of the object with these parameters.

Write a main function create two object of class **Date**, the data member of one object is initialized with zero through default constructor. The data member of second object is initialized with some values using a constructor that takes three parameters.

Prompt the user to input date (the month, the day of the month, and the year) in a main function, assign these values to the first object (using function setDate) and then display the value of the data members of two objects using function printDate().

Lab 06 – *Static Class Data and Static Member Function*

1. Objectives

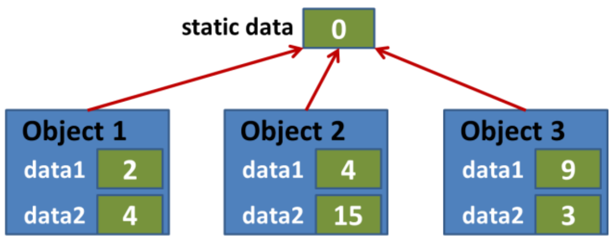
This Lab will help students in understanding the concept static data and member function of a class. Student will also know what a static object is.

2. Outcome

After completion of this Lab students know in which scenario static data members, static functions and static objects are used and helpful.

3. Introduction

There is an important exception to the rule that each object of a class has its own copy of all the data members of the class. In certain cases, only one copy of a variable should be shared by all objects of a class. A static data member is used for such propose.. Such a variable represents "class-wide" information (i.e., a property of the class shared by all instances, not a property of a specific object of the class). The declaration of a static member begins with keyword static.



In the above figure, consider Object1, Object2 and Object3 are three of a class. Each object have its own private data members. The class has a static data member that is share among all the object of that class. This data member has it own memory allocated that is separate from the memory allocated for each object of that class.

<u>Static data member – access</u>
A public static data member can be access using any object of the class or class name
- [Object Name].[name of static variable]
  - e1.count;
- [Class Name] :: [name of static variable]
  - Employee ::count;

A private static member can be access using a public member function
- [objectName].[public member function]
  - e1.getCount();
- [Class name]::[public member function]
  - Employee::getCount();

A static function is a function that can only access static data member of class. Ity cannot access non static data member of that class

Static<u> vs. non static functions</u>

- Static function can only access static data member of class
  - className::static_function();
  - Object_of_class.static function();
- Non static member function can access static and non static data member of class

– Object_of_class.Non_static_function();

4. Examples

4.1 Static data member and static function

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

class gamma {
  private:
    static int total;
    int id;
  public:
    gamma() {
      total++;
      id = total;
    }
    ~gamma() {
      total--;
      cout <<"Destroying ID number : "<< id << endl;
    }
    static void showtotal() {
       cout <<"Total is "<<total<<endl;
       //cout<<id<<endl;  // cannot access non static data member
    }
    void showid() {
       cout <<"ID number is " << id<<endl;
    }
};
//Static data initialization
int gamma::total = 0;

main() {
  gamma g1;
  gamma::showtotal();
  gamma g2, g3;
  gamma::showtotal();
  g1.showid();
  g2.showid();
  g3.showid();
  system("pause");
}
```

```
Program Output
Total is 1
Total is 3
ID number is 1
ID number is 2
ID number is 3
Press any key to continue . . .
```

5. Lab Tasks

Create a SavingsAccount class. Use a static data member annualInterestRate to store the annual interest rate for each of the savers. Each member of the class contains a private data member savingsBalance indicating the amount the saver currently has on deposit. Provide member function calculateMonthlyInterest that calculates the monthly interest by multiplying the balance by annualInterestRate divided by 12; this interest should be added to savingsBalance. Provide a static member function modifyInterestRate that sets the static annualInterestRate to a new value. Write a driver program to test class SavingsAccount. Instantiate two different objects of class SavingsAccount, saver1 and saver2, with balances of $2000.00 and $3000.00, respectively. Set the annualInterestRate to 3 percent. Then calculate the monthly interest and print the new balances for each of the savers. Then set the annualInterestRate to 4 percent, calculate the next month's interest and print the new balances for each of the savers.

6. Home Task:

Write C++ program to count the number of objects created and destroyed for a class using static data members and static member functions

7. References

Programming in C++ , By Robert Lafore

Lab 07 – *Inheritance and Overriding*

1. Objectives

To familiarize the students with various concepts and terminologies of inheritance in Programming.

2. Outcome

After this lab the students should be able to declare the derived classes along with the access of base class members. They should learn the purpose of protected Access Specifier and working with derived class constructors. They should be familiar with the use of overriding.

3. Introduction

3.1. Inheritance

Inheritance is a way of creating a new class by starting with an existing class and adding new members. The new class can replace or extend the functionality of the existing class.The existing class is called the base class and the new class is called the derived class.

3.2. Protected Access Specifier

Protected members are directly accessible by derived classes but not by other users.

A class member labeled **protected** is accessible to member functions of derived classes as well as to member functions of the same class.

3.3. Derived class constructor

Constructors are not inherited, even though they have public visibility. However, the super reference can be used within the child's constructor to call the parent's constructor. In that case, the call to parent's constructor must be the first statement.

3.4. Overriding

Method overriding, in object oriented programming, is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has same name, same parameters or signature, and same return type as the method in the parent class. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. Some languages allow a programmer to prevent a method from being overridden.

4. Examples

4.1. This example will explain the method to specify the derived class.

```
#include<iostream>
#include<conio.h>
using namespace std;

class BaseClass {                         // start of base class
            int num1;
```

```
      public:                              //public section of class
            void setInt(int n);            // function prototype
             int getInt();                 // function prototype

};                                               // end of base class

class DerivedClass : public BaseClass {    //start of derived class
             int num2;
         public:
            void setJ(int n);              // function prototype
             int mul();                    // function prototype

};                                         //end of derived class

void BaseClass::setInt(int n)              // function declaration

{
      num1 = n;
}

int BaseClass::getInt()                    // function declaration
{
      return num1;
}

void DerivedClass::setJ(int n)             //function declaration
{
      num2 = n;
}

int DerivedClass::mul()                    // function declaration
{
      cout<<"Result=";
      return (num2 * getInt());
}
int main()
{
      DerivedClass ob;                     //derived class object

      ob.setInt(10);                       // load num1 in BaseClass
      ob.setJ(4);                    // load num2 in DerivedClass

      cout << ob.mul()<<endl;              // displays 40
       system("pause");
      return 0;
}
```

4.2. This example will explain the way to access the base class members with derived class object.

#include<iostream>

using namespace std;

```
class Counter                             //base class
{
   protected:                             //NOTE: not private
      unsigned int count;                 //count
   public:
      Counter() : count(0)                //no-arg constructor
         {  }
      Counter(int c) : count(c)           //1-arg constructor
         {  }
      unsigned int get_count() const      //return count
         { return count; }
      Counter operator ++ ()              //incr count (prefix)
         { return Counter(++count); }
   };
```

```
class CountDn : public Counter            //derived class
{
public:
   Counter operator -- ()              //decr count (prefix)
      { return Counter(--count); }
};
int main()
{
 CountDn c1;                           //c1 of class CountDn

 cout << "\nc1=" << c1.get_count();    //display c1

 ++c1; ++c1; ++c1;                     //increment c1, 3 times
 cout << "\nc1=" << c1.get_count();    //display it

 --c1; --c1;                           //decrement c1, twice
 cout << "\nc1=" << c1.get_count();    //display it
 cout << endl;
 system("pause");
 return 0;
}
```

4.3. This example demonstrates that how we can use the concept of Overriding.

```
 #include<iostream>
#include<conio.h>
using namespace std;
class Stack {
protected:                                    //NOTE: can't be private
    enum { MAX = 3 };                   //size of stack array
    int st[MAX];                        //stack: array of integers
   int top;                             //index to top of stack
public:
Stack()                                 //constructor
{
  top = -1;
}
void push(int var)
{
   st[++top] = var;                     //put number on stack
}
int pop()
{
  return st[top--];                     //take number off stack
}
};
class Stack2 : public Stack {
 public:
void push(int var)
{                                             //put number on stack
  if(top >= MAX-1)                            //error if stack full
    {
      cout << "\nError: stack is full";
      exit(1);
    }
  Stack::push(var);                          //call push() in Stack class
 }
    int pop()                                 //take number off stack
    {
     if(top < 0)                              //error if stack empty
```

```
            { cout << "\nError: stack is empty\n";
              exit(1);
            }
      return Stack::pop();                           //call pop() in Stack class
          }
      };
  int main()
 {
 Stack2 s1;
 s1.push(11);                                        //push some values onto stack
 s1.push(22);
 s1.push(33);
 cout << endl << s1.pop();                           //pop some values from stack
 cout << endl << s1.pop();
 cout << endl << s1.pop();
 cout << endl << s1.pop();                           //oops, popped one too many...
 cout << endl;
 system("pause");
 getch();
 return 0;
 }
```

5. Lab Tasks

5.1. Imagine a publishing company that markets both book and audio-cassette versions of its works. Create a class publication that stores the *title* and *price* of a publication.

a. from this class derive two classes:

i. book, which adds a *page count* and
ii. tape, which adds a *playing time* in minutes.
iii. each of these three classes should have *getdata()* function to get its data from the user at the keyboard and a *putdata()* function to display its data.

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

5.2. Write a class Person that has attributes of *id, name* and *address*. It has a constructor to initialize, a member function to input and a member function to display data members. Create another class Student that inherits Person class. It has additional attributes of *rollnumber* and *marks*. It also has member function to input and display its data members.

5.3. Write a base class Computer that contains data members of *wordsize(in bits), memorysize (in megabytes), storagesize (in megabytes)* and *speed (in megahertz).* Derive a Laptop class that is a kind of computer but also specifies the object's length, width, height, and weight. Member functions for both classes should include a default constructor, a constructor to inialize all components and a function to display data members.

6. Home Tasks

6.1. Write a program having a base class Student with data members *rollno, name* and Class define a member functions *getdata()* to input values and another function *putdata()* to display all values. A class Test is derived from class Student with data members *T1marks, T2marks, T3marks, Sessional1, Sessional2, Assignment* and *Final.* Also make a function *getmarks()* to enter marks for all variables except Final and also make a function *putmarks()* to display result. Make a function *Finalresult()* to calculate value for final variable using other marks. Then display the student result along with student data.

6.2. Write a program that declares two classes. The parent class is called ***Simple*** that has two data members *num1* and *num2* to store two numbers. It also has four member functions.

- The *add()* function adds two numbers and displays the result.
- The *sub()* function subtracts two numbers and displays the result.
- The *mul()* function multiplies two numbers and displays the result.
- The *div()* function divides two numbers and displays the result.

The child class is called ***Complex*** that overrides all four functions. Each function in the child class checks the value of data members. It calls the corresponding member function in the parent class if the values are greater than 0. Otherwise it displays error message.

6.3. An electricity board charges the following rates to domestic users to discourage large consumption of energy.

- For the first 100 units − 50 P per unit
- Beyond 100 units − 60 P per unit

If the total *cost* is more than Rs.250.00 then an additional surcharge of 15% is added on the difference. Define a class Electricity in which the function Bill computes the *cost*. Define a derived class More_Electricity and override Bill to add the *surcharge*.

6.4. (Package Inheritance Hierarchy) Package-delivery services, such as FedEx®, DHL® and UPS®, offer a number of different shipping options, each with specific costs associated. Create an inheritance hierarchy to represent various types of packages. Use Package as the base class of the hierarchy, then include classes TwoDayPackage and OvernightPackage that derive from Package. Base class Package should include data members representing the *name, address, city, state* and *ZIP code* for both the sender and the recipient of the package, in addition to data members that store the *weight* (in ounces) and *cost per ounce* to ship the package. Package's constructor should initialize these data members. Ensure that the *weight* and *cost per ounce* contain positive values. Package should provide a public member function *calculateCost*() that returns a double indicating the cost associated with shipping the package. Package's *calculateCost()* function should determine the cost by multiplying the *weight* by the *cost per ounce*. Derived class TwoDayPackage should inherit the functionality of base class Package, but also include a data member that represents a *flat fee* that the shipping company charges for two-day-delivery service. TwoDayPackage's constructor should receive a value to initialize this data member. TwoDayPackage should redefine member function *calculateCost*() so that it computes the *shipping cost* by adding the *flat fe*e to the *weight-based cost* calculated by base class Package's *calculateCost*() function. Class OvernightPackage should inherit directly from class Package and contain an additional data member representing an additional fee per ounce charged for overnight-delivery service. OvernightPackage should redefine member function calculateCost() so that it adds the additional *fee per ounce* to the standard *cost per ounce* before calculating the shipping *cost*. Write a test program that creates objects of each type of Package and tests member function *calculateCost*().

7. References

[i]Object-Oriented Programming in C++, By Robert Lafore

[ii]C++ How to Program, By Deitel & Deitel

Lab 08 – *Class Hierarchies*

1. Objectives

To familiarize the students with class hierarchies, multiple inheritance and multilevel inheritance.

2. Outcome

After this lab the students should be able to differentiate between public and private Inheritance. They should learn the role of constructor and destructors in derived classes. They should be able to implement multilevel inheritance and multiple inheritance..

3. Introduction

3.1. Public and private inheritance

The public keyword in the inheritance syntax means that publicly accessible members inherited from the base class stay publicly accessible in the derived class. But for private keyword in the inheritance syntax means that accessible members inherited from the base become private members of derived class.

3.2. Constructors and Destructors in Derived Classes

Derived classes do not inherit constructors or destructors from their base classes, but they do call the constructor and destructor of base classes. When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor. When an object of a derived class is destroyed, its destructor is called first, then that of the base class.

3.3. Multiple inheritance and Multi-level inheritance

Classes can be derived from other classes that are already derived from other classes. This creates multi-level Inheritance.

Syntax:

```
class A {
public:
        int x;
};
class B : public A {
public:
         int y;
};
class C : public B { };
```

A Class can be derived from more than one base class. This is called *Multiple inheritance.*

Syntax:

```
class A{
          };
class B{
          };
class C: public A, public B
{
 };
```

4. Examples

4.1. This example demonstrates the Public and Private Inheritance between the classes.

```
 #include <iostream>
 class base
 {
   private:
   int pridataA;                      //private data member
   protected:
   int prodataA;                      //protected data member
   public:
   int pubdataA;                      //public data member

 };
class derived1: public base            //publically - derived class
{
   public:
   void funct()
   {
    int a;
   a=pridataA;                        //error: not accessible
   a=prodataA;                        //ok
   a=pubdataA;                        //ok
   }
};
class derived2: public base                  //privately - derived class
{
   public:
   void funct()
   {
    int a;
   a=pridataA;                        //error: not accessible
   a=prodataA;                              //ok
   a=pubdataA;                              //ok
    }
};
int main()
{
 int a;
 derived1 objd1;
 a=objd1.pridataA;                    //error: not accessible
 a= objd1.prodataA;                   // error: not accessible
 a= objd1.pubdataA;                   //ok (base public to derived1)
 derived2 objd2;
 a=objd2.pridataA;                    //error: not accessible
 a=objd2.prodataA;                          // error: not accessible
 a=objd2.pubdataA;   // error: not accessible (base private to derived2)
 return 0;
}
```

4.2. This example will explain the process of multiple inheritance.

Example 1:

```
#include<iostream>
#include<conio.h>
using namespace std;

class student
{
    protected:
       int rno,m1,m2;
```

```
   public:
                void get()
              {
                            cout<<"Enter the Roll no :";
                            cin>>rno;
                            cout<<"Enter the two marks   :";
                            cin>>m1>>m2;
              }
};
class sports
{
    protected:
       int sm;                   // sm = Sports mark
    public:
                void getsm()
              {
                 cout<<"\nEnter the sports mark :";
                 cin>>sm;

              }
};
class statement:public student,public sports
{
    int tot,avg;
    public:
    void display()
              {
                 tot=(m1+m2+sm);
                 avg=tot/3;
                 cout<<"\n\n\tRoll No    : "<<rno<<"\n\tTotal \t:"<<tot;
               cout<<"\n\tAverage    : "<<avg;
              }
};
int main()
{
   statement obj;
   obj.get();
   obj.getsm();
   obj.display();
   getch();
}
```

4.3. This example demonstrates the concept of multilevel inheritance between the classes.

```
#include<iostream>
using namespace std;
class base1
{
    protected:
      int number1;
    public:
     void showA()
      {
         cout<<"enter a value for first number:"<<endl;
         cin>>number1;
      }
};
class base2:public base1
{
    protected:
        int number2;
    public:
    void showB()
    {
```

```
                cout<<"enter value for second number:";
                cin>>number2;
          }
     };

     class derive:public base2
    {
       public:
          void showC()
          {
                showA();                        //accessing base1's function
                showB();                        //accessing base2's function
                cout<<"number1*number2 ="<<number1*number2;
          }
     };
     int main()
     {
        derive ob1;
        ob1.showC();
        cout<<endl;
        system("pause");
     }
```

5. Lab Tasks

5.1. Create a Class named base which has two data members. Then derive a class derived1 from base class which has one data members. Derive a class derived2 from derived1.

i. Write functions for each class to get and display values.

ii. Write main() function to create object of derived2 and through that object access the data member of base class and derived1 class

5.2. Create a class Person having *name*, *age* and *gender* as its data members. Create another class Employee which has *employername* and *dailywages* as it data member. From these two classes derive another class teacher which contains teacher grade as data member.

i. Write set and get functions to enter and display the data members.

ii. Write main function to implement these classes. Enter the teacher data to show multiple inheritance.

5.3. Consider the following classes

```
#include <iostream.h>
class Date
   {
         int day;
         int month;
         int year;
   public:
         Date();
         ~Date();
         void display();       // displays the date
         Date get();           //  accesses the date members
         void set();           // sets the date members
   };
   class Time
   {
         int hour;
         int minute;
         int second;
   public:
         Time();
         ~Time();
         void display();      // displays the time
         Time get();    // accesses the time members
         void set();          // sets the time members
   };

class DateAndTime : public Date, public Time
   {
         int digital;
   public:
         void display();            // prints date and time
   };
```

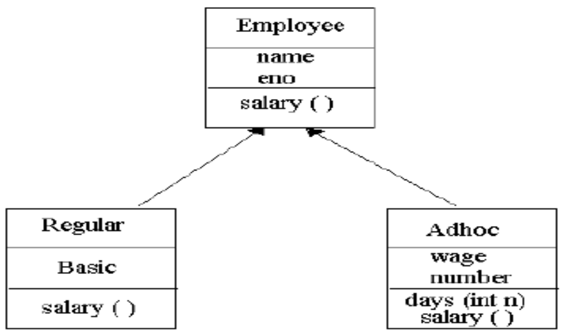
- **i.** Define an instance object of class DateTime called Watch.
- **ii.** Write a main () function that would initialize the values through the constructor functions, and then allows them to be reset through the set () functions. Be sure and display the results following the constructor before you use the set functions.
- **iii.** Through the use of the display () function, the time and date are to be displayed. Note that the display () functions in all three classes need to be defined, as well as the constructor and all the access functions.
- **iv.** Turn in the .CPP file(s) and screen output.

5.4. Write a class Teacher that contains the attribute *teacher name, age* and *address*. It also contains member function to input and display its attributes. Write another class Author that contains the attributes *author name, address* and *number of books* written by him. It also contains member functions to input and display its attributes. Write a third class Scholar that inherits both Teacher and Author classes. Test these classes from main() by creating objects of derived classes and testing functions in a way that clear concept of multiple Inheritance.

6. Home Tasks

6.1. An organization has two types of employees: regular and adhoc. Regular employees get a salary which is basic + DA + HRA where DA is 10% of basic and HRA is 30% of basic. Adhoc employees are daily wagers who get a salary which is equal to Number * Wage.

(i) Define the classes shown in the following class hierarchy diagram:



(ii) Define the constructors. When a regular employee is created, basic must be a parameter.

When adhoc employee is created wage must be a parameter.

(iii) Define the destructors.

(iv) Define the member functions for each class. The member function days ( ) updates number of the Adhoc employee.

(v) Write a test program to test the classes.

6.2. Write a program having a base class Student with data member *rollno* and member functions *getnum*() to input *rollno* and *putnum*() to display *rollno*. A class Test is derived from class Student with data member *marks* and member functions *getmarks*() to input *marks* and *putmarks*() to display *marks*. Class Sports is also derived from class Student with data member *score* and member functions *getscore*() to input *score* and *putscore*() to display *score*. The class Result is inherited from two base classes, class Test and class Sports with data member *total* and a member function *display*() to display *rollno*, *marks*, *score* and the *total*(*marks* + *score*).

6.3. Write a class LocalPhone that contains an attribute *phone* to store a local telephone number. The class contains member functions to input and display phone number. Write a child class NatPhone for national phone numbers that inherits LocPhone class. It additionally contains an attribute to store *city code*. It also contains member functions to input and show the city code. Write another class IntPhone for international phone numbers that inherit NatPhone class. It additionally contains an attribute to store *country code*. It also contains member functions to input and show the country code. Test these classes from main() by creating objects of derived classes and testing functions in a way that clear concept of multi-level Inheritance.

6.4. Start with the publication, book and tape classes. Add 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 sale amount from the user and a putdata() function to display the sales figure.

Alter the book and tape classes, so they are derived from both publication and sales.

An object of book or tape should should input and output ans sales data along with other data.

Write a main function to create a book and tape object and exercise their input/output capabilities.

7. **References**:

[i] Object-Oriented Programming in C++, By Robert Lafore

[ii] C++ How to Program, By Deitel & Deitel

Lab 09 – *Composition / Containership*

1. Objectives

The objective of this lab is to teach the students, the concept of Composition, which is also known as Containership.

2. Outcome

2.1. At the end of this lab student will know the purpose of Composition.

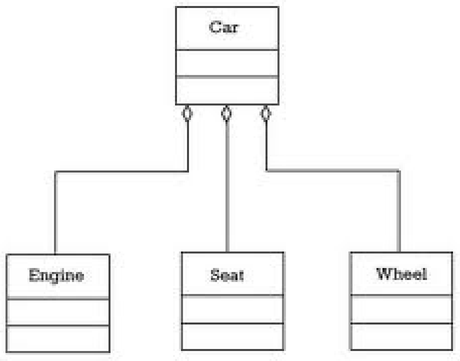
2.2. Student will be able to use Composition in different real time scenarios.

3. Introduction

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc. Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on. This process of building complex objects from simpler ones is called composition (also known as object containership).

More specifically, composition is used for objects that have a "has-a" relationship to each other. A car has-a metal frame, has-an engine, and has-a transmission. A personal computer has-a CPU, a motherboard, and other components. You have-a head, a body.

```
class wheel
{
};
class car
{
        wheel w1;
}
```



4. Examples

Composition is about expressing relationships between objects. Think about the example of a manager. A manager has the properties e.g Title and club dues. A manager has an employment record. And a manager has an educational record. The phrase "has a" implies a relationship where the manager owns, or at minimum, uses, another object. It is this "has a" relationship which is the basis for composition. This example can be programmed as follows:

```
#include<iostream>
#include<string>

using namespace std;

class studentRecord
{
private:
        string degree;
public:
        studentRecord()
        {}
        void getdata()
        {
                cout<<"Enter Degree: ";
                cin>>degree;
        }
};

class employeeRecord
{
private:
        int emp_id;
        double salary;
public:
        employeeRecord ()
        {}
        void getdata()
        {
                cout<<"Enter Employee ID: ";
                cin>>emp_id;
                cout<<"Enter Salary: ";
                cin>>salary;
        }
};

class manager
{
private:
        string title;
        double dues;
        employeeRecord emp;
        studentRecord stu;
public:
        void getdata()
        {
                emp.getdata();
                cout<<"Enter Title: ";
                cin>>title;
                cout<<"Enter Dues: ";
                cin>>dues;
                stu.getdata();
        }
};

void main()
{
        manager m1;
        cout<<"Enter data for manager 1: ";
        m1.getdata();
        system("pause");
}
```

5. Lab Tasks

5.1. Code the example given above with mentioning some message indicating the class construction and destruction in the constructors and destructors of each class and check the calling of constructors and destructors.

5.2. Create an Address class, which contains street#, house#, city and code (all of type char*). Create another class Person that contains an address of type Address. Give appropriate get and set functions for both classes. Test class person in main.

5.3. Write the program, which has two classes one, is Date having members (day, month, year) and the other class is called Employee. The employee has Date class as member as each employee has Date of joining, Date of Birth etc.

a. Determine if an employee joined the organization within last five years if the current year is 2012.
b. Determine if an Employee has age less than 40 years?

6. Home Tasks

6.1. Read digits as Characters input and convert to equivalent numeral values.

7. References

[i] http://www.learncpp.com/cpp-tutorial/102-composition/

Lab 10 – *Polymorphism*

1 Objectives

To familiarize the students with the concepts of virtual functions and polymorphism, late/dynamic binding, abstract class and pure virtual functions.

2 Outcome

This lab should teach the students to use the concept of polymorphism which is the ability of objects of different types to respond to functions of same name.

- Program in general rather than program in specific.
- Cast derived class objects as objects of the base class.
- Invoke parent class functions on derived class objects through polymorphism
- Use virtual functions
- Create abstract classes with the help of pure virtual functions

3 Introduction

3.1 Polymorphism

The word polymorphism is a combination of two words poly and morphism. ‘Poly’ means many and ‘morphism’ means form. In object-oriented programming, polymorphism is the ability of objects of different types to respond to functions of the same name. The user does not have to know the exact type of the object in advance. Polymorphism enables to “program in general” rather than “program in specific”.

- It enables writing programs that process objects of classes that are in the same class hierarchy as if they are all objects of the hierarchy’s base class.
- There are derived classes that inherit from the generic base class
  - ➢ The derived classes inherit the generic interface from base class
  - ➢ Each derived class implements/overrides the generic interface according to the functionality it is supposed to offer but interface or function names remain the same
  - ➢ Derived classes make objects but since they carry base class’s interface, they are processed/treated as objects of base class
  - ➢ Each same function call through an object results in different actions/responses depending on which object has it been called through

3.2 Virtual Functions

A function that appears to exist in some part of a program but does not exist really is called virtual function. Virtual functions are used to implement polymorphism. They enable user to execute completely different functions by same function call. Virtual function allows implementing a different functionality in the derived class.

- With virtual functions, the type of object being pointed to determines which virtual function to execute
  - o Doesn’t matter what type of pointer is being used
- Technically, polymorphism is to declare a pointer of base class and point it towards an object of any derived class, and then using the pointer, calling the functions of the derived class
  - o To be able to do this, you need to have virtual functions in the base class and override them in derived classes

3.3 Early Binding and Late Binding

- The early binding occurs when everything required to call a function is known at compile time.
- The late binding occurs when some information to call a function is decided at execution time.

3.4 Abstract Class

An abstract class is a class that can only be a base class for other classes. You cannot create an instance of an abstract class. Instead, it contains operations that are common to all of its derived classes.

- A class becomes abstract, if it contains **even one** “pure virtual function”
- Pure virtual function:
  - Doesn’t and can’t have a function body i.e. its just a function prototype/function declaration
  - Is appended by “= 0”
  - MUST and must always be overridden in derived classes
  - Note: simple virtual function doesn’t necessarily need overriding in derived classes
- Abstract classes are almost always only used as base classes

4 Examples

Run the following examples.

4.1 Example 1

This example demonstrates a polygon-rectangle hierarchy with the use of virtual functions in the base class and how Rectangle objects are being casted into the Base class type through pointers.

```
#include<iostream>
#include<stdlib.h>
using namespace std;

class Polygon
{
    private:
        double width, height;
    public:
       Polygon(double w, double h): width(w), height(h)
       {}
         double getWidth()
        { return width; }
         double getHeight()
        { return height; }
         virtual double getArea()   //virtual function
        { return 0; }
         virtual string getType()    //virtual function
        { return "Generic Polygon"; }
};

class Rectangle: public Polygon
{
    public:
        Rectangle(double w, double h): Polygon(w, h)
    {}
        double getArea()
        {
        return (getWidth() * getHeight());
        }
        string getType()
    { return "Rectangle"; }
};
int main()
{
        Polygon* p = new Polygon(20, 30);
        Polygon* r = new Rectangle(20, 30);
        cout<<p->getType()<<", Area: "<<p->getArea()<<endl;
        cout<<r->getType()<<", Area: "<<r->getArea()<<endl;
        system("pause");
        return 0;
}
```

4.2 Example 2

This example demonstrates the polygon-rectangle hierarchy with pure virtual functions. The base class now is an abstract base class and lots of other classes added to the hierarchy.

```
#include<iostream>
#include<stdlib.h>
using namespace std;

class Polygon
{
      private:
        double width, height;
      public:
        Polygon(double w, double h): width(w), height(h)
        {}
        double getWidth()
        { return width; }
        double getHeight()
        { return height; }
        virtual double getArea() = 0;      //pure virtual function
        virtual string getType() = 0;      //pure virtual function
};

class Rectangle: public Polygon
{
      public:
            Rectangle(double w, double h): Polygon(w, h)
        {}
        double getArea()
        { return (getWidth() * getHeight()); }
        string getType()
        { return "Rectangle"; }
};
```

```
class Square: public Rectangle
{
      public:
        Square(double l): Rectangle(l, l)
        {}
        string getType()
        { return "Square"; }
};

class Parallelogram: public Rectangle
{
      public:
        Parallelogram(double w, double h): Rectangle(w, h)
        {}
        string getType()
        { return "Parallelogram"; }
};

class Triangle: public Polygon
{
      public:
        Triangle(double w=0, double h=0): Polygon(w, h)
        {}
       double getArea()
       { return (getWidth() * getHeight())/2; }
       string getType()
     { return "Triangle"; }
};
int main()
{
    Polygon* ppoly[100]={new Rectangle(1.2, 2.1),
    new Square(2.5),
    new Triangle(6, 4),
    new Parallelogram(20,30),
    new Rectangle(100,200),
    new Rectangle(1.2, 2.1),
    new Square(2.5),
    new Triangle(6, 4),
    new Parallelogram(20,30),
    new Rectangle(100,200),
    NULL};     //array of pointers
            for(int i=0; ppoly[i]!=NULL; i++)
                        cout<<i+1<<". "
                        <<ppoly[i]->getType()<<endl
                                    <<"\n\t- Width  = "<<ppoly[i]->getWidth()
                                    <<"\n\t- Height = "<<ppoly[i]->getHeight()
                                    <<"\n\t- Area   = "<<ppoly[i]->getArea()
                                    <<endl<<endl<<"---------------"<<endl<<endl;
   system("pause");
   return 0;
}
```

4.3 Example 3

The following example uses pure virtual function in the person class. The person class is an abstract class and no objects can be created for it. It also adds two derived classes, student and professor. The derive classes each contain a function called getData() and isOutstanding(). Student and professor objects are casted into the person class type through array of pointers.

```
#include<iostream>
#include<stdlib.h>
#include<string.h>
using namespace std;

class person //person class
{
   protected:
            string name;
   public:
            void getName()
             { cout << " Enter name: "; cin >> name; }
            void putName()
            { cout << "Name is: " << name << endl; }

             virtual void getData() = 0; //pure virtual func
             virtual bool isOutstanding() = 0; //pure virtual func
};
class student : public person //student class
{
   private:
         float gpa; //grade point average
   public:
         void getData() //get student data from user
         {
         person::getName();
         cout << " Enter student's GPA: "; cin >> gpa;
         }
         bool isOutstanding()
         { return (gpa > 3.5) ? true : false; }
};

class professor : public person //professor class
{
   private:
         int numPubs; //number of papers published
   public:
         void getData() //get professor data from user
         {
         person::getName();
         cout << " Enter number of professor's publications: ";
         cin >> numPubs;
         }
         bool isOutstanding()
         { return (numPubs > 100) ? true : false; }
};
```

```
int main()
{
    person* persPtr[100]; //array of pointers to persons
    int n = 0;                    //number of persons on list
    char choice;
    do {
          cout << "Enter student or professor (s/p): ";
          cin >> choice;
          if(choice=='s') //put new student
             persPtr[n] = new student;
          else                  //put new professor
             persPtr[n] = new professor;
           persPtr[n++]->getData(); //get data for person
           cout << " Enter another (y/n)? ";
           cin >> choice;
        } while( choice=='y' ); //cycle until not 'y'
    for(int j=0; j<n; j++) //print names of all persons
        {
           persPtr[j]->putName();
           if( persPtr[j]->isOutstanding() )
           cout << " This person is outstanding\n";
        }
     system("pause");
     return 0;
} //end main()
```

5 Lab Tasks

5.1 Consider the class

```
class base
{
          public:
          virtual void iam()
          {
          cout << "base\n";
           }
};
```

a. Derive two classes from class base, and for each define iam() to write out the name of the class.
b. Declare objects of each class, and call iam() from them.
c. Assign the address of objects of the derived classes to base pointers and call iam() through the pointers.
d. Remove the virtual keyword from the base class member function, run your code again, and compare the results.

5.2

Develop a simple payroll application. There are three kinds of employees in the system: salaried employee, hourly employee, and commissioned employee. The system takes as input an array containing employee objects, calculates salary polymorphically, and generates report.

Make Employee an abstract class. Declare salary() and display() as pure virtual functions in it. Derive salaried employee (monthly), hourly employee (per hour basis), and commissioned employee (bonus on completing each target) from base class Employee. The display() function should show employee no, employee name, and salary of all employees.

5.3

Create a base class called shape. Use this class to store two double type values that could be used to compute the area of figures. Derive two specific classes called triangle and rectangle from the base shape. Add to base class, a member function get_data() to initialize base class data members and another member functions display_area() to compute and display the area of figures. Mark the display_area() as a virtual function and redefine this function in the derived class to suit their requirements.(Use pure virtual function)

6 Home Tasks

Create a class hierarchy that performs conversions from one system of units to another. Your program should perform the following conversions,

i. Liters to Gallons,
ii. Fahrenheit to Celsius and
iii. Feet to Meters

The base class **convert** declares two variables, val1 and val2, which hold the initial and converted values, respectively. It also defines the functions getinit() and getconv(), which return the initial value and the converted value. These elements of convert are fixed and applicable to all derived classes that will inherit convert. However, the function that will actually perform the conversion, compute(), is a pure virtual function that must be defined by the classes derived from convert. The specific nature of compute() will be determined by what type of conversion is taking place.
Three classes will be derived from convert to perform conversions of Liters to Gallons (l_to_g), Fahrenheit to Celsius (f_to_c) and Feet to Meters (f_to_m), respectively. Each derived class overrides compute() in its own way to perform the desired conversion.
Test these classes from main() to demonstrate that even though the actual conversion differs between l_to_g, f_to_c, and f_to_m, the interface remains constant.

7 References

[1] “Lab Manual”, available on “http://www.scs.dauniv.ac.in/ Labmanual/PraList %20C++ .pdf”, 25 July. 2012
[2] “C/C++Programming Lab Manual” on http://www.csupomona.edu/~hlin/ Cplus Manual /CCplusmanual.pdf, 25 July. 2012
[3] “C++ How to Program”, By H. M. Deitel - Deitel & Associates, Inc., P. J. Deitel - Deitel & Associates, Inc.
[4] Object Oriented Programming in C++, By Robert Lafore.

Lab 11 – *File processing/Handling and Advanced File Operations*

1. Objectives

1.1. To review the basic concept of files

1.2. To understand the use of random/sequential access files

1.3. To understand and use various types of files (binary and text)

2. Outcome

2.1. Students should be able to create, read, write and update files.

2.2. Students should be able to include C++ stream input/output and file processing.

2.3. Students should be able to select appropriate data structures and file methods for programs of varying complexity

3. Introduction

A file is a collection of information stored (usually) on a disk. Files, just like variables, have to be defined in the program. The data type of a file depends on whether it is used as an input file, output file, or both. Input files have a data type called ifstream, Onput files have a data type of ofstream, and files used as both have the data type fstream. We must add the #include <fstream> directive when using files.

Examples:

```
ofstream outfile;          // defining outfile as an output file
ifstream infile;           // defining infile as an input file
fstream datafile;          // defining datafile to be both an input and output file
```

After their definition, files must still be opened, used (information stored to or data read from the file), and then closed.

3.1. Opening Files:

A file is opened with the open function. This ties the logical name of the file that is used in the definition to the physical name of the file used in the secondary storage device (disk). The statement infile.open("payroll.dat"); opens the file payroll.dat and lets the program know that infile is the name by which this file will be referenced within the program. If the file is not located in the same directory as the C++ program, the full path (drive, etc.) MUST be indicated:

infile.open("a:\\payroll.dat"); This tying of the logical name infile with the physical name payroll.dat means that wherever infile is used in the program, data will be read from the physical file payroll.dat. A program should check to make sure that the physical file exists. This can be done by a conditional statement as is shown in th example section.

3.2. Reading from a File:

Files have an “invisible” end of line marker at the end of each line of the file. Files also have an invisible end of file marker at the end of the file. When reading from an input file within a loop, the program must be able to detect that marker as the sentinel data (data that meets the condition to end the loop).

3.3. Output Files

Output files are opened the same way: outfile.open("payment.out").

Whenever the program writes to outfile, the information will be placed in the physical file payment.out. Notice that the program generates a file stored in the same location as the source file. The user can indicate a different location for the file to be stored by indicating the full path (drive, etc.).

3.4. Files Streams.

A file can be used for both input and output. The fstream data type, which is used for files that can handle both input and output, must have a file access flag as an argument to the open function so that the mode, input or output, can be determined. There are several access flags that are used to indicate the use of the file. The following chart lists frequently used access flags.

3.4.1. Flag mode Meaning

3.4.1.1. ios::in Input mode.

The file is used for "reading" information. If the file does not exist, it will not be created.

3.4.1.2. ios::out Output mode.

Information is written to the file. If the file already exists, its contents will be deleted.

3.4.1.3. ios::app Append mode.

If the file exists, its contents are preserved and all output is written to the end of the file. If it does not exist then the file will be created. Notice how this differs from ios::out.

3.4.1.4. ios::binary Binary mode.

Information is written to or read from in pure binary format.

3.5. Closing a File

Files should be closed before the program ends to avoid corrupting the file or losing valuable data.

```
infile.close();

outfile.close();

dataFile.close();
```

3.6. Passing Files as Parameters to Functions.

Files can be passed as parameters just like variables. Files are always passed by reference. The & symbol must be included after the data type in the function heading and prototype.

void GetData(ifstream& infile, ofstream& outfile);

// prototype of function with files as parameters

Recall that each file has an end of line marker for each line as well as the end of file marker at the end of the file. Whenever whitespace (blanks, newlines, controls, etc.) is part of a file, a problem exists with the traditional >> operator in inputting character data..

A program reading first names into some string variable (array of characters) has a problem reading a name like Mary Lou since it has a blank space in it. The blank space between Mary and Lou causes the input to stop when using the >> operator. The get function can be used to input such strings.

infile.get(firstname, 20);

The get function does NOT skip leading whitespace characters but rather continues to read characters until it either has read, in the example above, 19 characters or it reaches the newline character \n (which it does NOT consume).

3.7. Binary Files

Although ASCII text is the default method for storing information in files, we can specify that we want to store data in pure binary format by "opening" a file in binary mode with the ios::binary flag. The write member function is then used to write binary data to the file. This method is particularly useful for transferring an entire array of data to a file. Binary files are efficient since they store everything as 1s or 0s rather than as text.

3.8. Files and Records

Files are often used to store records. A "field" is one piece of information and a "record" is a group of fields that logically belong together in a unit.

Example:

Name	Test1	Test2	Final
Brown	89	97	88
Smith	99	89	97

Each record has four fields: Name, Test1, Test2, and Final. When records are stored in memory, rather than in files, C++ structures provide a good way to organize and store them.

```
struct Grades
{
        char name[10];
        int test1;
        int final;
};
```

An identifier defined with name 'Grades' is a structure that can hold one record.

The write function, mentioned in the previous section, can be used to write records to a file.

```
fstream test("score.dat", ios::out|ios::binary);
Grades student;                    // defines a structure variable
// code that gets information into the student record
test.write((char *) &student, sizeof(student));
```

The test.write function used to write a record stored as a struct is similar to the write function used for an array with one big difference. Notice the inclusion of (&).Arrays are passed by pointer.

3.9. Random Access Files

There are two file stream member functions that are used to move the read/write position to any byte in the file. The seekp function is used for output files and seekg is used for input files.

```
dataOut.seekp(30L, ios::beg);
```

This instruction moves the marker position of the file referred by dataOut to 30 positions from the beginning of the file. The first argument, 30L (L indicates a long integer), represents an offset (distance) from some point in the file that will be used to move the read/write position. That point in the file is indicated by the second argument (ios::beg). This access flag indicates that the offset is calculated from the beginning of the file. The offset can be calculated from the end (ios::end) or from the current (ios::cur) position in the file.

If the eof marker has been set (which means that the position has reached the end of the file), then the member function clear must be used before seekp or seekg is used.

Two other member functions may be used for random file access: tellp and tellg. They return a long integer that indicates the current byte of the file's read/write position. As expected, tellp is used to return the write position of an output file and tellg is used to return the read position of an input file.

Assume that a data file letterGrades.txt has the following single line of information:

ABCDEF

Marker positions always begin with 0. The mapping of the characters to their position is as follows:

A B C D E F

0 1 2 3 4 5

4. Examples

4.1. Opening Files

```
ifstream infile;

infile.open("payroll.dat");

if (!infile)

{

        cout << Error opening file.  It may not exist were indicated.\n;

        return 1;

}
```

Note: return 1 is used as an indicator of an abnormal occurrence. In this case the file in question can not be found.

4.2. Reading from a File:

```
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
        ifstream infile;                                // infile will refer to an input file
        ofstream outfile;                               // output will refer to output file
        infile.open("payroll.dat");       // This statement opens 'payroll.data' as an
        input file.
        // Whenever infile is used, data from the file payroll.dat will be read.
        outfile.open("payment.out"); // This statement opens 'payment.out'  as an
        output file.
        // Whenever outfile is used, information will be sent to the file payment.out.

        int hours;              // The number of hours worked
        float payRate;          // The rate per hour paid
        float net;              // The net pay
        if (!infile)
          {
                cout << "Error opening file.\n";
                cout << "Perhaps the file is not where indicated.\n";
                return 1;
          }
        outfile << fixed << setprecision(2);
        outfile << "Hours   Pay Rate   Net Pay"  << endl;
        infile >> hours;
        while (infile)
          {
                infile >> payRate;
                net = hours * payRate;
                outfile << hours << setw(10) << "$ " << setw(6)<< payRate <<
                setw(5) << "$ " << setw(7)<< net << endl;
                infile >> hours;
         }
        infile.close();
        outfile.close();
        return 0;
}
```

Notice the statement outfile << fixed << setprecision(2); in the above program. This shows that the format procedures learned for cout can be used for output files as well. Remember that setw(x) can be used as long as the iomanip header file is included.

This program assumes that a data file exists and contains an undetermined number of records with each record consisting of two data values, hours and payRate. Suppose the input data file (payroll.dat) contains the following:

```
        40 10.00

        30 6.70

        50 20.00
```

The program will produce the following output file (payment.out).

```
Hours     Pay Rate        Net Pay

          40 $         10.00            $ 400.00

          30 $         6.70             $ 201.00

          50 $         20.00            $1000.00
```

The input file contains data for one employee on each line. Each time through the while loop, information is processed for one employee.

There are other ways of determining when the end of the file is reached. The eof( ) function reports when the end of a file is encountered. The loop in Sample Program 11.1 can be replaced with the following:

```
infile >> hours;

while (!infile.eof())
```

```
{
        infile >> payRate;
        net = hours * payRate;
        outfile << hours << setw(10) << "$ " << setw(6) << payRate << setw(5)
        << "$ " << setw(7) << net << endl;
        infile >> hours;
}
```

In addition to checking to see if a file exists, we can also check to see if it has any data in it. The following code checks first if the file exists and then if it is empty.

```
inData.open("sample2.dat");
if(!inData)
        cout << "file does not exist" << endl;
        cout << "File is empty" << endl;
else
        //rest of program
```

The peek function actually looks ahead in the file for the next data item, in this case to determine if it is the end of file marker. ch must be defined as char data type. Since the peek function looks "ahead" for the next data item, it can be used to test for end of file in reading values from a file within a loop without priming the read.

4.3. File Streams

```
#include <fstream>
using namespace std;
int main()
{
        fstream test ("grade.dat", ios::out)
        /* test will refer to 'grade.dat' which will be opened as output file, ios::out is
        the file access flag*/
        // code of the program goes here. The code will put values in the file
        test.close();                    // close the file
        test.open("grade.dat", ios::in)
         // the same file is reopened now as an input file ios::in is the
        //file access flag
        test.close();
}
```

In the following example, we check for a file's existence before opening it. We first attempt to open the file for input. If the file does not exist, then the open operation fails and we open it as an output file.

```
fstream dataFile;
dataFile.open("grades.txt", ios::in);
if (dataFile.fail())
// The fail will return true if file does not exist
{
        // The file does not exist, so create it.
        dataFile.open("grades.txt", ios::out);
        // file is processed here:  data sent to the file
}
else                                    // the file already exists.
```

```
{
        cout << "The file grades.txt already exists. \n";
        // process file here
        dataFile.close( ); //close the file
}
```

Just as cin >> is used to read from the keyboard and cout << is used to write to the screen, filename >> is used to read from the file filename and filename << is used to write to the file filename.

4.4. Passing Files as Parameters to Functions.

Given the following data file

```
Mary Lou <eol>
Becky <eol>
Debbie <eol>
<eof>
```

Note: Both the <eol> and <eof> are NOT visible to the programmer or user.
There are several options for reading and printing this data.

```
char dummy;                          // created to read the end of line character
char firstname[80];                  // array of characters for the first name
outfile << "Name " << endl;
infile.get(firstname,80);            // priming the read inputs the first name
while(infile)
{
        infile.get(dummy);                // reads the end of line character into dummy
        outfile << firstname << endl;     // outputs the name
        infile.get(firstname,80);         // reads the next name
}
```

In the above example, dummy is used to consume the end of line character.
input.get(firstname,80); reads the string Mary Lou and stops just before reading the <eol> end of line character. The infile.get(dummy) gets the end of line character into dummy.
Another way to do this is with the ignore function, which reads characters until it encounters the specific character it has been instructed to look for or until it has skipped the allotted number of characters, whichever comes first. The statement infile.ignore(81,'\n') skips up to 81 characters stopping if the new line '\n' character is encountered. This newline character IS consumed by the function, and thus there is no need for a dummy character variable.

```
char firstname[80];
outfile << "Name " << endl;
infile.get(firstname,80);
{
        infile.ignore(81,'\n');           // read and consume the end of line character
        outfile << firstname << endl;
        infile.get(firstname,80);
}
```

The following sample program shows how names with embedded whitespace along with numeric data can be processed. Parts in bold indicate the changes from Sample Program 11.2. Assume that the payroll.dat file has the following information:

```
John Brown  40  10.00
```

The program will produce the following information in payment.out:

```
Name            Hours    Pay Rate    Net Pay
John Brown       40       $10.00      $ 400.00
```

```
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;
const int MAX_NAME = 11;
int main()
{
    ifstream infile;        // object to manipulate input file
    ofstream outfile;    // object to manipulate outfile
    infile.open("payroll.dat");  // This statement opens ‘payroll.dat’  as an input file.
    while(!infile.fail())
        outfile.open("payment.out"); // This statement opens ‘payment.out’ as an
    output file.
    int hours;             // The number of hours worked
    float payRate;     // The rate per hour paid
    float net;           // The net pay
    char ch;            // ch is used to hold the next value  (read as character) from the
    file
    char name[MAX_NAME];   // array of characters for the name of a student, with
    at most 10 //characters
    if (!infile)
    {
        cout << "Error opening file.\n";
        cout << "Perhaps the file is not where indicated.\n";
        return 1;
    }
    outfile << fixed << setprecision(2);
    outfile << "Name  Hours     Pay Rate   Net Pay"  << endl;
    while ((ch = infile.peek()) != EOF)     // reading until not finished
    {
        infile.get(name,MAX_NAME);       // gets names with blanks
        infile >> hours;
        infile >> payRate;
        infile.ignore(81,'\n');     // ignores the rest of the line and consumes end of line
        marker
        net = hours * payRate;
        outfile << name << setw(10) << hours << setw(10) << "$" << setw(6) <<
        payRate << setw(5) << "$ " << setw(7)<< net << endl;
    }
    infile.close();
    outfile.close();
    return 0;
}
```

Another way to read in lines containing whitespace characters is with the getline member function.

```
char firstName[80];
outfile << "Name " << endl;
infile.getline(firstName,81);
while (infile)
{
        outfile << firstName << endl;
        intfile.getline(firstName,81);
}
```

4.5. Binary Files

fstream test("grade.dat", ios::out | ios::binary); // This defines and opens the file 'grade.dat' as an output binary file

// creates and initializes an integer array

int grade[arraysize] = {98, 88, 78, 77, 67, 66, 56, 78, 98, 56};

test.write((char*)grade, sizeof(grade)); // write all values of array to file

test.close();   // close the file

test.write((char*)grade, sizeof(grade));

In the above example calls the write function. The name of the file to be written to is 'grade.dat'. The first argument is a character pointer pointing to the starting address of memory, in this case to the beginning of the grade array. The second argument is the size in bytes of the item written to the file. Sizeof() is a function that determines the size in bytes of its argument.

The following sample program initializes an array and then places those values into a file as binary numbers. The program then adds 10 to each element of the array and prints those values to the screen. Finally the program reads the values from the same file and prints them. These values are the original numbers. Study the program and its comments very carefully.

```
#include <fstream>
#include <iostream>
using namespace std;
const int ARRAYSIZE = 5;
int main()
{
     fstream test("grade.dat", ios::out | ios::binary);
                 // note the use of | to separate file access flags
     int grade[ARRAYSIZE] = {98,88,78,77,67};
     int count;                                 // loop counter
     test.write((char*)grade, sizeof(grade)); // write values of grade array to file
     test.close();                              // close file
     // now add 10 to each grade
     cout << "The values of grades with 10 points added\n";
     for (count =0; count < ARRAYSIZE; count++)
     {
         grade[count] = grade[count] + 10; // this adds 10 to each elemnt of the array
         cout << grade[count] << endl; // write the new values to the screen
     }
     test.open("grade.dat", ios::in);  // reopen the file but now as an input file
     test.read((char*) grade, sizeof(grade));
     /*  The above statement reads from the file test and places the values found into
     the grade array.  As with the write function, the first argument is a character
     pointer even though the array itself is an integer.  It points to the starting address
     in memory where the file information is to be transferred */
     cout << "The grades as they were read into the file"  << endl;
     for (count =0; count < ARRAYSIZE; count++)
```

```
    {
        cout << grade[count] << endl; // write the original values to the screen
    }
    test.close();
    return 0;
}
```

The output to the screen from this program is as follows:

```
The values of grades with 5 points added
98
88
78
77
67
The grades as they were read into the file
108
98
88
87
77
```

4.6. Files and Records

The following example takes records from the user and stores them into a binary file.

```
#include <fstream>
#include <iostream>
#include <cctype>                                   // for toupper function
using namespace std;
const int NAMESIZE = 31;
struct Grades         // declaring a structure
{
        char name[NAMESIZE];
        int test1;
        int test2;
        int final;
};
int main()
{
        fstream tests("score.dat", ios::out | ios::binary); // defines tests object referring
        to 'score'dat' file which will be opened as an output binary file
        Grades student;
        char more;                                  // used to determine if there is
        more input
        do
        {
                cout << "Enter the following information"  << endl;
                cout << "Student's name: ";
                cin.getline(student.name,NAMESIZE);
                cout << "First test score :";
                cin >> student.test1;
                cin.ignore();    // ignore rest of line
                cout << "Second test score: ";
                cin >> student.test2;
                cin.ignore();
                cout << "Final test score: ";
                cin >> student.final;
                cin.ignore();
                tests.write((char *) &student, sizeof(student)); // write this record to
                the file referred by tests object
                cout << "Enter a y if you would like to input more data\n ";
                cin >> more;
                cin.ignore();
        }
        while (toupper(more) == 'Y');
        tests.close();
        return 0;
}
```

4.7. Random Access Files

The following example demonstrates the use of **seekg** and **tellg.**

```
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;
int main()
{
      fstream inFile("letterGrades.txt", ios::in);
      long offset;   // used to hold the offset of the read position from some point
      char ch;       // holds character read at some position in the file
      char more;     // used to indicate if more information is to be given
      do
      {
            cout << "The read position is currently at byte "<< inFile.tellg() << endl;
            // This prints the current read position (found by the tellg function)
            cout << "Enter an offset from the beginning of the file: ";
            cin >> offset;
            inFile.seekg(offset, ios::beg);
            // This moves the pointer upto offset length from the beginning of the file.

            inFile.get(ch);  // This gets one byte of information from the file
            cout << "The character read is " << ch << endl;
            cout << "If you would like to input another offset enter a Y" << endl;
            cin >> more;
            inFile.clear(); // This clears the file in case the EOF flag was set
      }
      while (toupper(more) == 'Y');
      inFile.close();
      return 0;
}
```

CAUTION: If you enter an offset that goes beyond the data stored, it prints the previous

Character’s offset.

5. LAB Tasks

5.1. Assume that the data file has hours, payRate, stateTax, and fedTax on one line for each employee. stateTax and fedTax are given as decimals (5% would be .05). Complete this program by filling in the code (places in bold).

```
// This program uses hours, pay rate, state tax and fed tax to determine gross and net
pay.
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;
int  main()
{
        // Fill in the code to define payfile as an input file
        float gross;
        float net;
        float hours;
        float payRate;
        float stateTax;
        float fedTax;
        cout << fixed << setprecision(2) << showpoint;
        // Fill in the code to open payfile and attach it to the physical file named
        payroll.dat
        // Fill in code to write a conditional statement to check if payfile does not
        exist.
        {
                cout << "Error opening file. \n";
                cout << "It may not exist where indicated" << endl;
                return 1;
        }
        cout << "Payrate     Hours    Gross Pay      Net Pay"<< endl   << endl;
        // Fill in code to prime the read for the payfile file.
        // Fill in code to write a loop condition to run while payfile has more data
        to process.
        {
                payfile >> payRate >> stateTax >> fedTax;
                gross = payRate * hours;
                net = gross - (gross * stateTax) - (gross * fedTax);
                cout << payRate << setw(15) << hours << setw(12) << gross<<
                setw(12)  << net << endl;
                // Fill in the code to finish this with the appropriate variable to be
                input
        }
        payfile.close();
        return 0;
}
```

5.2. Run the program. Note: the data file does not exist so you should get the error message:

Error opening file. It may not exist where indicated.

Create a data file with the following information:

```
40 15.00 .05 .12
50 10 .05 .11
60 12.50 .05 .13
```

Save it in the same folder as the .cpp file. What should the data file name be?

Run the program. Record the output here:
Change the program so that the output goes to an output file called pay.out and run the program. You can use any logical internal name you wish for the output file.

5.3. Files as Parameters and Character Data
Retrieve program Grades.cpp and the data file graderoll.dat from the Lab 11 folder. The code is as follows:

```
// FILL IN DIRECTIVE FOR FILES
#include <iostream>
#include <iomanip>
using namespace std;
// This program reads records from a file.  The file contains the following: student's
name,
//two test grades and final exam grade. It then prints this information to the screen.
const int NAMESIZE = 15;
const int MAXRECORDS = 50;
struct Grades                    // declares a structure to store information from file
{
        char name[NAMESIZE + 1];
        int test1;
        int test2;
        int final;
};
typedef Grades gradeType[MAXRECORDS];
// This makes gradeType a data type  that holds MAXRECORDS

// Fill in the code for the prototype of the function readit
// where the first argument is an input file, the second is the
// array of records, and the third will hold the number of
// records currently in the array.
int main()
{
        ifstream indata;
        indata.open("graderoll.dat");
        int numRecord;              // number of records to read in
        gradeType studentRecord;
        if(!indata)
        {
                cout << "Error opening file. \n";
                cout << "It may not exist where indicated" << endl;
                return 1;
        }
        // fill in the code to call the function ReadIt.
        // output the information
        for (int count = 0; count < numRecord; count++)
        {
                cout << studentRecord[count].name << setw(10)
                << studentRecord[count].test1<< setw(10) << studentRecord[count].test2;
                cout << setw(10) << studentRecord[count].final << endl;
        }
        return 0;
}
//************************************************************
 //This procedure reads records into an array of records from an input file and
 //keeps track of the total number of records
 //Declare inData:  data file containing information to be placed in the array
```

```
//Declare dataout: an array of records and the number of records
//***********************************************************
void   readIt
(// FILL IN THE CODE FOR THE FORMAL PARAMETERS AND THEIR
// DATA  TYPES. (inData, gradeRec and total are the formal parameters) --total is passed by
// reference)
{
        total = 0;
        inData.get(gradeRec[total].name, NAMESIZE);
        while (inData)
        {
                // FILL IN THE CODE TO READ test1
                // FILL IN THE CODE TO READ test2
                // FILL IN THE CODE TO READ final
                total++;     // add one to total
                // FILL IN THE CODE TO CONSUME THE END OF LINE
                // FILL IN THE CODE TO READ name
        }
}
```

6. Home Tasks

6.1. Complete the program by filling in the code (areas in bold). This problem requires that you study very carefully the code and the data file already written to prepare you to complete the program.

Add another field called letter to the record which is a character that holds the letter grade of the student. This is based on the average of the grades as follows: test1 and test2 are each worth 30% of the grade while final is worth 40% of the grade. The letter grade is based on a 10 point spread. The code will have to be expanded to find the average.

90–100	A
80–89	B
70–79	C
60–69	D
0–59	F

6.2. Binary Files and the write Function

Retrieve program budget.cpp from the Lab 11 folder. The code is as follows:

```
// This program reads in from the keyboard a record of financial information consisting of a //person’s name, income, rent, food costs, utilities and miscellaneous expenses.  It then  //determines the net money (income minus all expenses)and places that information in a //record which is then written to an output file.
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;
const int NAMESIZE = 15;
struct  budget  //declare a structure to hold name and financial information
{
        char name[NAMESIZE+1];
        float income;       // person's monthly income
```

```
            float rent;         // person's monthly rent
            float food;         // person's monthly food bill
            float utilities;    // person's monthly utility bill
            float miscell;      // person's other bills
            float net;          // person's net money after bills are paid
    };
    int main()
    {
            fstream indata;
            ofstream outdata
            indata.open("income.dat", ios::out | ios::binary);  // open file as binary output.
            outdata.open("student.out");    // output file that we will write student information to.
            outdata << left << fixed << setprecision(2);    // left indicates left justified for fields
            budget person;   //defines person to be a record
            cout << "Enter the following information" << endl;
            cout << "Person's name: ";
            cin.getline(person.name, NAMESIZE);
            cout << "Income :";
            cin >> person.income;
            // FILL IN CODE TO READ IN THE REST OF THE FIELDS:
            // rent, food, utilities AND miscell TO THE person RECORD
            // find the net field
            person.net = // FILL IN CODE TO DETERMINE NET INCOME (income -
            expenses)
            // write this record to the file
            // Fill IN CODE TO WRITE THE RECORD TO THE FILE indata (one
            instruction)
            indata.close();
            // FILL IN THE CODE TO REOPEN THE 'income.dat' FILE, NOW AS AN
            INPUT FILE.
            // FILL IN THE CODE TO READ THE RECORD FROM FILE AND PLACE IT
            IN THE
            // person RECORD (one instruction)
            // write information to output file
            outdata << setw(20) << "Name" << setw(10) << "Income" << setw(10) << "Rent"
            << setw(10) << "Food" << setw(15) << "Utilities" << setw(15)
            << "Miscellaneous" << setw(10) << "Net Money" << endl << endl;
            // FILL IN CODE TO WRITE INDIVIDUAL FIELD INFORMATION OF THE
            RECORD //to the STUDENT.OUT  FILE.(several instructions)
            return 0;
    }
```

6.3. This program reads in a record with fields name, income, rent, food, utilities, and miscell from the keyboard. The program computes the net (income minus the other fields) and stores this in the net field. The entire record is then written to a binary file (income.dat). This file is then closed and reopened as an input file. Fill in the code as indicated by the comments in bold.

Sample Run:

```
Enter the following information

Person's Name: Billy Berry

Income: 2500

Rent: 700

Food: 600

Utilities: 400

Miscellaneous: 500
```

The program should write the following text lines to the output file student.out.

```
Name Income Rent Food Utilities Miscellaneous Net Money
Billy Berry 2500.00 700.00 600.00 400.00 500.00 300.00
```

6.4. Alter the program to include more than one record as input. Use an array of records.

Sample Run:

```
Enter the following information
Person's Name: Billy Berry
Income: 2500
Rent: 700
Food: 600
Utilities: 400
Miscellaneous: 500
Enter a Y if you would like to input more data
n
That's all the information
```

The output file student.out should then have the following lines of text written to it.

```
Name          Income    Rent    Food    Utilities   Miscellaneous   Net Money
Billy Berry   2500.00   700.00  600.00   400.00       500.00          300.00
```

6.5. Random Access Files Retrieve program randomAccess.cpp and the data file proverb.txt from the Lab 11 folder. The code is as follows:

```
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;
int main( )
{
        fstream inFile("proverb.txt", ios::in);
        long offset;
        char ch;
        char more;
        do
        {
//  Fill in the code to write to the screen the current read position (with label)
                cout << "Enter an offset from the current read position: ";
                cin >> offset;
// Fill in the code to move the read position "offset" bytes from the CURRENT read position.
// Fill in the code to get one byte of information from the file and place it in the variable "ch".
                cout << "The character read is "  << ch << endl;
                cout << "If you would like to input another offset enter a Y"<< endl;
                cin >> more;
                //  Fill in the code to clear the eof flag.
        } while (toupper(more) == 'Y');
        inFile.close();
        return 0;
}
```

6.6. Fill in the code as indicated above by the comments above. Change the program so that the read position is calculated from the end of the file. What type of offsets would you need to enter to get characters from the proverb? Do several sample runs with different numbers to test your program.

Write a program that will:

1) read an array of records from the keyboard,

2) store this information to a binary file,

3) read from the binary file back to the array of records,

4) store this information to a textfile. Left justify the information for each field.

Each record will consist of the following fields:

first name 15 characters

last name 15 characters

street address 30 characters

city 20 characters

state 5 characters

zip long integer

You may assume a maximum of 20 records.

Sample Run:

```
Enter the following information
Person's First Name: Billy
Person's Last Name: Berry
Street: 205 Main Street
City: Cleveland
State: TX
Zip: 45679
Enter a Y if you would like to input more data
```

The output file contains the following:

```
First Name    Last Name    Street            City State      Zip Code
Billy         Berry      205   Main Street  Cleveland Tx       45679
```

7. References:

[1] C++ HOW TO PROGRAM BY Deitel and Deitel

[2] OBJECT ORIENTED PROGRAMMING BY Robert Lafore.

[3] http://digital.cs.usu.edu/~lindad/cs1405/labManual.pdf

Lab 12 – *Operator Overloading*

1. Objectives

The objective of this lab is to teach the students, the use of unary, binary and stream operators for user defined classes i.e. operator overloading.

2. Outcome

2.1. At the end of this lab student will know the purpose of Operator Overloading.

2.2. Student will be able to use unary, binary and stream operators for user defined classes.

3. Introduction

One of the nice features of C++ is that you can give special meanings to operators, when they are used with user-defined classes. This is called operator overloading. You can implement C++ operator overloads by providing special member-functions on your classes that follow a particular naming convention. For example, to overload the + operator for your class, you would provide a member-function named operator+ on your class.

The following set of operators is commonly overloaded for user-defined classes:

4. Examples

++ --	(Increment and Decrement Operators)	Unary Operators
=	(Assignment Operator)	Binary Operators
+ - *	(Binary Arithmetic Operators)	
+= -= *=	(Compound Assignment Operators)	
== != < >	(Comparison Operators)	
<< >>	(Insertion and Extraction Operators)	Stream Operators

4.1. Example of Unary Operators Overloading

The unary operators operate on a single operand and following are the examples of Unary operators:

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how increment (++) operator can be overloaded for prefix as well as postfix usage. Similar way you can overload operator (--).

```
#include<iostream>
```

```
using  namespace std;

class Time
{
   private:
      int hours;             // 0 to 23
      int minutes;           // 0 to 59
   public:
      // required constructors
      Time(){
         hours = 0;
         minutes = 0;
      }
      Time(int h, int m){
         hours = h;
         minutes = m;
      }
      // method to display time
      void displayTime()
      {
         cout<<"H: "<<hours;
         cout<<" M: "<<minutes<<endl;
      }
      // overloaded prefix ++ operator
      Time operator++ ()
      {
         ++minutes;// increment this object
         if(minutes >= 60)
         {
            ++hours;
            minutes -= 60;
         }
         return Time(hours, minutes);
      }
      // overloaded postfix ++ operator
      Time operator++( int )
      {
         // save the orignal value
         Time T(hours, minutes);
         // increment this object
         ++minutes;
         if(minutes >= 60)
         {
            ++hours;
            minutes -= 60;
         }
         // return old original value
         return T;
      }
};
void main()
{
   Time T1(11, 59), T2(10,40);

   ++T1;                  // increment T1
   T1.displayTime();      // display T1
   ++T1;                  // increment T1 again
   T1.displayTime();      // display T1

   T2++;                  // increment T2
   T2.displayTime();      // display T2
   T2++;                  // increment T2 again
   T2.displayTime();      // display T2
   system("pause");
}
```

4.2. Example of Binary Operators Overloading

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explain how addition (+) operator can be overloaded. Similar way you can overload subtraction (-) division (/) or any other binary operators.

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

class Box
{
   double length;      // Length of a box
   double breadth;     // Breadth of a box
   double height;      // Height of a box
public:

   double getVolume(void)
   {
      return length * breadth * height;
   }
   void setLength( double len )
   {
       length = len;
   }

   void setBreadth( double bre )
   {
       breadth = bre;
   }

   void setHeight( double hei )
   {
       height = hei;
   }
   // Overload + operator to add two Box objects.
   Box operator+(const Box& b)
   {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
   }
};

// Main function for the program
void main( )
{
   Box Box1;       // Declare Box1 of type Box
   Box Box2;       // Declare Box2 of type Box
   Box Box3;       // Declare Box3 of type Box
   double volume = 0.0;
   // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);
   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : ";
   cout << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : ";
   cout << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : ";
   cout << volume <<endl;

   system("pause");
}
```

4.3. Example of Binary Operators Overloading

A date is an ideal candidate for a C++ class in which the data members (month, day, and year) are hidden from view. An output stream is the logical destination for displaying such a structure. This code displays a date using the **cout** object:

```
Date dt( 1, 2, 92 );
cout << dt;
```

To get **cout** to accept a Date object after the insertion operator, overload the insertion operator to recognize an **ostream** object on the left and a Date on the right. The overloaded << operator function must then be declared as a friend of class Date so it can access the private data within a Date object.

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

class Date
{
    int mo, da, yr;         //Variable Declaration
public:
       //Constructor
    Date(int m, int d, int y)
    {
        mo = m; da = d; yr = y;
    }
    friend ostream& operator<<(ostream& os, const Date& dt);
};

ostream& operator<<(ostream& os, const Date& dt)
{
    // os as cout
    os << dt.mo << '/' << dt.da << '/' << dt.yr;
    return os;
}

////////////main function/////////
void main()
{
    Date dt(5, 6, 92);
    cout<<dt;
    system("pause");
}
```

5. Lab Tasks

5.1. Create a counter class, overload ++ operator for counter post and pre increment, use the object of counter class as a loop counter for printing a table in main function.

5.2. A complex number is a number which can be put in the form a + b*i.* Create a class for complex numbers, which handle real and imaginary part separately. Class should consist minimum required constructors, get and show methods also Overload the + operator for this class which work like this formula.

$$(a + bi) + (c + di) = (a + c) + (b + d)i.$$

5.3. Create a class of Distance including feet and inches. Class should consist minimum required constructors, get and show methods also overload the % operator for this class.

6. Home Tasks

6.1. Create a calculator for the complex number by creating a class of complex number with overloading all operators in it.(Operators: ++,--,+,-,/,*, >>, <<).

7. References

[i] http://www.tutorialspoint.com/cplusplus/increment_decrement_operators_overloading.htm

Lab 13 – *Templates*

1. Objectives

1.1. To teach the students about the importance of templates.

1.2. To get the students familiar with the use of templates.

1.3. To exercise the different ways the templates can be used.

2. Outcomes

2.1 At the end of the lab, student will understand the importance and use of templates

2.2 Students will be able to declare and define function templates

2.3 Students will be able to declare and define class templates

3. Introduction

Templates make it possible to use one function or class to handle many data types. The template concept can be used in two different ways: with functions and with classes. Using a template we can create a single function that can process any type of data i.e. the formal arguments of functions are of generic type. They can accept data of any type such as int, float, long etc. Thus a single function can be used to accept values of different data types. The portability provided by a template mechanism can be extended to classes.

3.1. Normal Template Function Declaration

```
template <class T_Identifier>


T_Identifier function_name ( *argument list (optional))
{
        // code
}
```

----

* Argument may be or may not be of "T_Identifier" type. Where T_Identifier is user defined identifier.

3.2. More than one Template type parameter

```
template  <class T1_Identifier, class T2_Identifier >
T2_Identifier function_name (argument list (optional))
{
        // code
}
```

3.3. Definition of Class Template

```
template <class T_Identifier >

class name_of_class

{

        // class data member and function

}
```

4. Examples

4.1. This example will demonstrate you to use a template function.

```
#include "stdafx.h"
#include<iostream>
#include<conio.h>
using namespace std;

template <class T> // declaration of template function
void show(T value)
{
     cout<<"\n Value="<< value;
}

void main() // starting of main function
{
   char charVar='A'; // declaration and initialization of character type
variable

   int intVar=10; // declaration and initialization of int type variable

   double doubleVar= 30.5555; // declaration and initialization of double
type variable

   show(charVar); // calling of template function with character variable
   show(intVar); // calling of template function with int variable
   show(doubleVar); // calling of template function with double variable
   cout<<endl;
   system("pause");

}// End of main function
```

4.2. This program demonstrates the use of template class and displays values of different data types using constructor and templates.

```
#include "stdafx.h"
#include<iostream>
#include<conio.h>
using namespace std;

template <class T> // declaration of template class
class Test
{
      public:
      Test ( T value) // constructor
      {
```

```
                cout<< "\n"<< "Value=" << value << "\t size in bytes:"
 <<sizeof(value);
          } // end of constructor body
 }; // end of class
 void main( ) // starting of main function
  {
          Test  <char> obj1('A'); // creating object of char type
          Test  <int> obj2(100); // creating object of int type
          Test <float> obj3(3.12); // creating object of float type
          cout<<endl;
          system("pause");
 } // end of main
```

4.3. This program guides you to define a constructor with multiple template variables.

```
#include "stdafx.h"
#include<iostream>
#include<conio.h>
using namespace std;

template <class T1, class T2> // declaration of template class
class Test
{
      public:
      Test(T1 FirstTypeValue, T2 SecondTypeValue) // constructor
      {
            cout<< "\n FirstTypeValue=" <<FirstTypeValue<< "\t
SecondTypeValue="<< SecondTypeValue;
      } // end of constructor body
};  // end of class
void  main( ) // starting of main function
{
      Test <int, float> obj1(2,2.5); // initializing object with two
members of different data types (int, float)
      Test <int, char> obj2(15, 'C'); // initializing object with two
members of different data types (int, char)
      Test <float, int> obj3(3.12, 50); // initializing object with two
members of different data types (float, int)

      cout<<endl<<endl;
      system("pause");
} // end of main
```

5. Lab Tasks

5.1. Write a program to swap the contents of two variables. Use template variables as function arguments**.**

5.2. Write a template function that returns the average of all the elements of an array. The arguments to the function should be the array name and the size of the array (type *int*). In main ( ), exercise the function with arrays of type *int*, *long*, *double*, and *char*.

6. Home Task

6.1. Conceptually, the stack class or abstract data type mimics the information kept in a pile on a desk. Informally, we first consider a material on a desk, where we may keep separate piles for bills that need paying, magazines that we plan to read, and notes we have taken. These piles of materials have several properties. Each pile contains some papers (information). In addition, for each pile, we can do several tasks:

- Start a new pile,
- Place new information on the top of a pile,
- Take the top item off of the pile,
- Read the item on the top, and
- Determine whether a pile is empty. (There may be nothing at the spot where the pile should be.)

These operations allow us to do all the normal processing of data at a desk. For example, when we receive bills in the mail, we add them to the pile of bills until payday comes. We then take the bills, one at a time, off the top of the pile and pay them until the money runs out.

When discussing these operations it is customary to call the addition of an item to the top of the pile a *Push* operation and the deletion of an item from the top a *Pop* operation.

More formally, a *stack* is defined as a class that can store data and that has the following operations:

- *Make-Stack*
  Create a new, empty stack object.
- *Empty*
  Empty returns true or false (1 or 0), depending upon whether the stack contains any items or not.
- *Push*
  Push adds the specified item to the top of the stack.
- *Pop*
  If the stack is not empty, *Pop* removes the top item from the stack, and this item is returned.
  If the stack is empty, nothing is returned, and an error is reported.
- *Top*
  If the stack is not empty, the top item is returned, but the contents of the stack are not changed.
  If the stack is empty, nothing is returned, and an error is reported.

Write a program to create template-based stack. Store *int* and *float* in it.

Lab 14 – *Exception Handling*

1. Objectives

To gain experience with

1.1. Exceptional conditions that may arise within a program.
1.2. Making your programs more reliable and robust by dealing with exceptional conditions.
1.3. Using a variety of mechanisms to handle exceptional conditions.
1.4. Using try blocks and catch handlers in your own programs.

2. Outcome

2.1 At the end of this lab, student will be expected to know the exception-throwing methods.
2.2 Students will be able to use try-catch to handle exceptions.
2.3 Students will be able to develop more robust applications.

3. Introduction

An exception is an indication of a problem that occurs during a program's execution. Exception handling enables programmers to resolve (or handle) such problems. In many cases, handling an exception allows a program to continue executing as if no problem had been encountered. A more severe problem could prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem before terminating in a controlled manner.

Exception handling enables the programmer to remove error-handling code from the "main line" of the program's execution, which improves program clarity and enhances modifiability. Programmers can decide to handle any exceptions they choose all exceptions, all exceptions of a certain type or all exceptions of a group of related types (e.g., exception types that belong to an inheritance hierarchy). Such flexibility reduces the likelihood that errors will be overlooked and thereby makes a program more robust.

a. <u>Exception Handling Mechanism</u>

```
Statement;
Statement;
try
    {
        Statement;
        Statement;
        throw (object);
    }

    catch( object)
    {
        Statement;
        Statement;
    }

   class AClass
    {
        public:
        class AnError
```

```
                  {  };
                 void Func( )
                {
                  if ( /* error condition*/)
                  throw AnError( );
               }
        };

        void main( )
    {
                  try
    {
                          AClass obj1;
                          obj1.Func( );
    }
                  catch( AClass::AnError)
                  {
                                  //tell user about error
                  }
    }
```

4. Examples

4.1 A program that guides you to throw exception when first variable is not greater than second, subtract the two values otherwise.

```
#include "stdafx.h"
#include<iostream>
#include<conio.h>
using namespace std;

void main( ) // starting of main function
{
       int FirstValue, SecondValue; // declaring two variables of int type
       cout<< " \n Enter values of FirstValue and SecondValue\n";
       cin>>FirstValue>>SecondValue; // taking 2 int values from the user
       int ConditionVariable;
       ConditionVariable=FirstValue >SecondValue? 0 :1; // checking
whether 1st entered value is greater than 2nd or not
       Try // starting of try block
       {
              if (ConditionVariable == 0)
                     {
                            cout<< "Subtraction (FirstValue-
SecondValue)="<< FirstValue-SecondValue << "\n";
                     }
                     else
                     {
                            throw( ConditionVariable); // throwing exception
if ConditionVariable is not equal to zero
                     }
       } // end of try block
       catch (int thrownValue) // start of catch block
       {
              cout<< "Exception caught subtraction=" << thrownValue <<
endl; // displaying thrown value
```

```
        } // end of catch block

        cout<<endl<<endl;
        system("pause");
}
```

4.2 A program that will help you to throw multiple exceptions and define multiple catch statements.

```
#include "stdafx.h"
#include<iostream>
#include<conio.h>
using namespace std;

void Function (int value) // start of the function
{
        Try // starting of try block
                {
                        if (value==0)
                             throw value; // throwing int value
                        else if(value>0)  throw 'p'; // throwing character
value
                        else if (value<0) throw 0.0; // throwing double value
                        cout << " *** try block *** \n";
                } // end of try block
        catch (char ThrownValue) // catch exception thrown by positive value
                {
                        cout<< "Caught a positive value \n";
                } // end of catch block
        catch (int ThrownValue) // catch exception thrown by null value
                {
                        cout<< "Caught a null value \n";
                } // end of catch block
        catch (double ThrownValue) // catch exception thrown by negative
value
                {
                        cout<< "Caught a negative value \n";
                } // end of catch block
} // end of function
void main( ) // starting of main
{

        cout<< "Demo of multiple catches \n";

        Function( 0); // calling function with 0
        Function( 5); // calling function with positive value i.e. 5
        Function(-1); // calling function with negative value i.e. -1

        cout<<endl<<endl;
        system("pause");
} // end of main
```

4.3. The following example will guides you to get familiar with the re-throwing exception mechanism.

```
#include "stdafx.h"
#include<iostream>
#include<conio.h>
using namespace std;
```

```
void Subtract (int FirstValue, int SecondValue) // start of the function
{
        cout<< "inside function subtraction \n";
        try // start of the try block
                {
                        if(FirstValue==0) // throw exception if FirstValue is = 0
                               throw FirstValue;
                        else
                                cout<< "Subtraction =" <<FirstValue-SecondValue
<<endl;
                } // end of the try block

    catch( int ) // start of the catch block
                {
                        cout<< "Caught null value \n";
                        throw;
                } // end of the catch block
        cout<< "*** End of Subtract function ***\n \n";
} // end of the function Subtract

void main()// start of the main
{
        cout<<"\ninside fuction main\n";

        try // start of the try block
                {
                        Subtract(8, 5); // calling function with (8,5)
                        Subtract(0, 8); // calling function with (0,8)
                }// end of the try block

        catch (int ) // start of the catch block
                {
                        cout<< "caught null inside main \n";
                } // end of the try block
        cout<< "end of function main\n";

        cout<<endl<<endl;
        system("pause");
} // end of the main
```

5. Lab Tasks

5.1. Practice the above mentioned examples

5.2. Create a class which only works for absolute numbers, if it encounters any negative occurrence, then it throw an exception to its handler and display errors.

5.3. Modify the above task, by creating an exception class with an error code and corresponding error message. Code and message should be thrown and displayed in catch block.

6. Home Task

6.1. The *queue* is another data structure. A physical analogy for a queue is a line at a bank. When you go to the bank, customers go to the *rear* (end) of the line and customers who are serviced come out of the line from the *front* of the line.

The main property of a queue is that objects go on the *rear* and come off of the *front* of the queue.

- *Make-Queue*
  Create a new, empty queue object.
- *Empty*
  Reports whether queue is empty or not.
- *Enter(or Insert)*
  Places an object at the rear of the queue
- *Delete (or Remove)*
  Removes an object from the *front* of the queue and produces that object.

Write a program to create a queue class and do queue operations with exception handling.