SREE VAHINI INSTITUTE OF SCIENCE & TECHNOLOGY :: TIRUVURU
KRISHNA (DT) ANDHRA PRADESH : 521235

II B.TECH I SEMESTER

OBJECT ORIENTED PROGRAMMING THROUGH C++

LAB MANUAL

R-20 REGULATION

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDEX

List of Experiments:

<table>
<tr><td rowspan="2">4</td><td>Exercise -4 (Inheritance)

1. Write C++ Programs and incorporating various forms of Inheritance
i) Single Inheritance
ii) Hierarchical Inheritance
iii) Multiple Inheritances
iv) Multi-level inheritance
v) Hybrid inheritance</td><td rowspan="2">20-34</td></tr>
<tr><td>2. Also illustrate the order of execution of constructors and destructors in inheritance</td></tr>
<tr><td rowspan="5">5</td><td>Exercise -5(Templates, Exception Handling)</td><td rowspan="5">35-41</td></tr>
<tr><td>1. a)Write a C++ Program to illustrate template class</td></tr>
<tr><td>2. b)Write a Program to illustrate member function templates</td></tr>
<tr><td>3. c) Write a Program for Exception Handling Divide by zero</td></tr>
<tr><td>4. d)Write a Program to rethrow an Exception</td></tr>
<tr><td rowspan="4">6</td><td>Exercise -6</td><td rowspan="4">42-51</td></tr>
<tr><td>1. Write a C++ program illustrating user defined string processing functions using
pointers (string length, string copy, string concatenation)</td></tr>
<tr><td>2. Write a C++ program illustrating Virtual classes & virtual functions.</td></tr>
<tr><td>3. Write C++ program that implement Bubble sort, to sort a given list of integers in
ascending order</td></tr>
</table>

EXERCISE:1

i.AIM: Write a c++ main function to create objects of distance class then input two distances and output the sum.

THEORY:

Develop a class to measure distance as feet (should be int), inches (should be float). Include member functions to set and get attributes. Include constructors. Develop functions to add two distances.

SOURCE CODE:

```
#include<iostream>
using namespace std;
class dist
{
  public:
  int feet,inch,x,y,z;
  void input()
  {
    cout<<"enter feet and inches:"<<"\n";
    cin>>feet>>inch;
  }
  void show()
  {
    cout<<"The distance is ";
    cout<<feet<<" feet "<<inch<<" inch\n";
```

```
  }
  void sum(dist x,dist y)
  {
    feet=x.feet+y.feet;
    inch=x.inch+y.inch;
    if(inch>=12)
    {
      feet=feet+1;
      inch=inch-12;
    }
  }
};
int main()
{
  dist x,y,z;
  x.input();
  y.input();
  z.sum(x,y);
  z.show();
}
```

OUTPUT:

```
Enter feet and inches:
6    3
Enter feet and inches:
3    11
The distance is: 10 feet and 2 inches
```

ii. AIM: Write a C++ program to illustrate the use of constructors and destructors (by using above programs)

THEORY:

Constructors and destructors are fundamental to the concept in C++.both constructor and destructor are more or less like normal functions (but with some differences) that are provided to enhance the capabilities of a class.

Constructor, as the name suggests is used to allocate memory (if required) and construct the objects of a class while destructor is used to do the required clean-up when a class object is destroyed. In this article, we will study the concept of constructors and destructors through working examples.

As we have already discussed that a constructor is used for creating an object. In precise terms, a constructor is a special function that gets called automatically when the object of a class is created. Similarly, a destructor is a special function that gets called automatically when a class object is deleted or goes out of scope.

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors initialize values to object members after storage is allocated to the object.

```
class A
{
```

```
  int x;
  public:
  A();  //Constructor
};
```

While defining a constructor you must remember that the name of constructor will be same as the name of the class, and constructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

SOURCE CODE:

```
#include<iostream>
using namespace std;
class dist
{
  public:
  int feet,inch;
  dist(){ }
  dist (int x,int y)
  {
    feet=x;
    inch =y;
  }
  dist (float m,float n)
  {
    feet=m;
    inch=n;
  }
  void display()
  {
    cout<<"Resulted distance is: ";
```

```
    cout<<feet<<" feet "<<inch<<" inch\n";
  }
  void sum(dist obj1,dist obj2)
  {
    feet=obj1.feet+obj2.feet;
    inch=obj1.inch+obj2.inch;
    if(inch>=12)
    {
      feet=feet+1;
      inch=inch-12;
    }
  }
  ~dist()
  {
    cout<<"object deleted:\n";
  }
};
int main()
{
  dist obj1(8,9),obj2(4.0f,7.3f),obj3;
  obj3.sum(obj1,obj2);
  obj3.display();
}
```

OUTPUT:

```
Object deleted.
Object deleted.
Resulted distance is:  13 feet 4 inch
Object deleted.
Object deleted.
Object deleted.
```

iii. AIM: Write a C++ program for function overloading in adding the distance between objects.

THEORY:

In some programming languages, **function overloading** or **method overloading** is the ability to create multiple methods of the same name with different implementations. Calls to an overloaded function will run a specific implementation of that function appropriate to the context of the call, allowing one function call to perform different tasks depending on context.

For example, *doTask()* and *doTask(object O)* are overloaded methods. To call the latter, an object must be passed as a parameter, whereas the former does not require a parameter, and is called with an empty parameter field. A common error would be to assign a default value to the object in the second method, which would result in an *ambiguous call* error, as the compiler wouldn't know which of the two methods to use.

Another appropriate example would be a *Print(object O)* method. In this case one might like the method to be different when printing, for example, text or pictures. The two different methods may be overloaded as *Print(text_object T); Print(image_object P)*. If we write the overloaded print methods for all objects our program will "print", we never have to worry about the type of the object, and the correct function call again, the call is always: *Print(something)*.

SOURCE CODE:

```
#include<iostream>
using namespace std;
class dist
{
  public:
```

```
int feet,inch;
dist()
{
  feet=0;
  inch=0;
}
dist(int x,int y)
{
  feet=x;
  inch=y;
}
dist(float m,float n)
{
  feet=m;
  inch=n;
}
void display()
{
  cout<<"The distance is ";
  cout<<feet<<" feet "<<inch<<" inch\n";
}
void sum(dist ob1,dist ob2)
{
  feet=ob1.feet+ob2.feet;
  inch=ob1.inch+ob2.inch;
  if(inch>=12)
  {
    feet=feet+1;
    inch=inch-12;
  }
}
void sum(int x,int y)
{
```

```
    feet+=x;
    inch+=y;
    if(inch>=12)
    {
      feet=feet+1;
      inch=inch-12;
    }
  }
};
int main()
{
  dist ob1(22,11),ob2(6.0f,5.3f),ob3,ob4(19,6);
  cout<<"Value of Obj1 is:\n";
  ob1.display();
  cout<<"Value of Obj2 is:\n";
  ob2.display();
  ob3.sum(ob1,ob2);
  cout<<"Addition of Obj1 and Obj2 is\n";
  ob3.display();
  ob4.sum(29,9);
  cout<<"After addition value of Obj4 is:\n";
  ob4.display();
}
```

OUTPUT:

```
Value of Obj1 is:
The distance is 22 feet 11 inch
Value of Obj2 is:
The distance is 6 feet 5 inch
Addition of Obj1 and Obj2 is
The distance is 29 feet 4 inch
```

After addition, value of Obj4 is:
The distance is 49 feet 3 inch

Exercise: 2

i. Write a program implementing Friend Function

```
#include <iostream>
using namespace std;
class Box
{
   private:
         int length;
   public:
          Box (): length (0) {}
   friend int printLength (Box); //friend function
};
int printLength (Box b)
{
    b. length +=10;
    return b. length;
}
int main ()
{
   Box b;
   cout <<" Length of box:" <<printLength (b)<<endl;
    return 0;
```

}

Output:

Length of box:10

ii. Write a program to illustrate this pointer

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id; //data member (also instance variable)
     string name; //data member(also instance variable)
     float salary;
     Employee(int id, string name, float salary)
     {
         this->id = id;
        this->name = name;
        this->salary = salary;
     }
     void display()
     {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;
     }
};
int main(void) {
   Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
   Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
   e1.display();
   e2.display();
   return 0;
```

25. }

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

iii. Write a Program to illustrate pointer to a class

```
#include <iostream>

using namespace std;

class Rectangle

{

        private:

                int length;

                int breadth;

        public:

                Rectangle(int l, int b)

                {

                        length=l;

                        breadth=b;

                }

                int getArea()

                {

                        return 2*length*breadth;

                }
```

```
};


int main()
{
  // creating an object of Rectangle
  Rectangle var1(5,2); // parameterized constrcutor


  /* creating a pointer of Rectangle type &
    assigning address of var1 to this pointer */
  Rectangle* ptr = &var1;


  /* calculating area of rectangle by using pointer
  ptr to call the objects getArea() function
  */
  int area = ptr->getArea();
  cout<<"Area of rectangle is: "<<area;
 return 0;
}
```

Output

Area of rectangle is: 20**or**

```
#include <iostream>
```

```
using namespace std;

class Box {
   public:
      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0) {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume() {
         return length * breadth * height;
      }

   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

int main(void) {
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2
   Box *ptrBox;                // Declare pointer to a class.

   // Save the address of first object
   ptrBox = &Box1;

   // Now try to access a member using member access operator
   cout << "Volume of Box1: " << ptrBox->Volume() << endl;

   // Save the address of second object
   ptrBox = &Box2;

   // Now try to access a member using member access operator
   cout << "Volume of Box2: " << ptrBox->Volume() << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Constructor called.
Constructor called.
Volume of Box1: 5.94
Volume of Box2: 102
```

Exercise: 3

AIM: Write a program to Overload Unary and Binary Operators as Member Function, and Non Member Function.

Unary operator as member function.

AIM: Write a C++ program to overload unary operator as member function.

THEORY:

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 post-fix as well like obj++ or obj–.

SOURCE CODE:

```
#include<iostream>
using namespace std;
class num
{
   private:
   int a,b,c;
   public:
   num(int j,int k,int m)
   {
      a=j;b=k;c=m;
   }
   void show(void);
   void operator ++( );
};
void num::show()
{
   cout<<"\n a= "<<a<<"\n b= "<<b<<"\n c= "<<c;
}
void num::operator ++( )
{
   ++a;
   ++b;
   ++c;
}
int main()
{
   num n(13,63,241);
   n.show();
   ++n;
   n.show();
}
```

OUTPUT:

a=14
b=64
c=242

ii. Binary operator as nonmember function.

AIM: Write a C++ program to illustrate binary operator as a non-member function.

THEORY:

A binary operator is an operator that operates on two operands and manipulates them to return a result. Operators are represented by special characters or by keywords and provide an easy way to compare numerical values or character strings.

Binary operators are presented in the form:

Operand1 Operator Operand2

SOURCE CODE:

```
#include<iostream>
using namespace std;
class FBOP
{
   int x,y;
   public:
   FBOP(){  }
   FBOP(int a,int b)
   {
```

```
      x=a;
      y=b;
   }
   void show()
   {
      cout<<"x= "<<x<<"\t y= "<<y<<endl;
   }
   friend FBOP operator+(FBOP ob1,FBOP ob2);
};
FBOP operator+(FBOP ob1,FBOP ob2)
{
   FBOP temp;
   temp.x=ob1.x+ob2.x;
   temp.y=ob1.y+ob2.y;
   return temp;
}
int main()
{
   int a,b;
   cout<<"Enter first object values:\n";
   cin>>a>>b;
   FBOP ob1(a,b);
   cout<<"Enter second object values:\n";
   cin>>a>>b;
   FBOP ob2(a,b),ob3;
   ob3=ob1+ob2;
   cout<<"Values of first object are:\n";
   ob1.show();
   cout<<"Values of second object are:\n";
   ob2.show();
   cout<<"Addition of two objects is:\n";
   ob3.show();
}
```

OUTPUT:

```
Enter first object values:
12      36
Enter second object values:
42      15
Values of first object are:
x=12    y=36
Values of second object are:
x=42    y=15
Addition of two objects is:
x=54    y=51
```

ii. Write a c ++ program to implement the overloading assignment = operator.

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

class Length
{
   private:
      int kmeter;
      int meter;

   public:

      Length() //default constructor
      {
         kmeter = 0;
         meter = 0;
      }
      Length(int km, int m)  //overloaded constructor.
      {
         kmeter= km;
         meter = m;
      }
      void operator = (const Length &l )
      {
         kmeter = l.kmeter;
         meter = l.meter;
      }

      //method to display length.
```

```
      void disLength()
      {
         cout <<kmeter<<"Km "<<meter<<"M"<< endl;
      }
};

int main()
{
   Length l1(1, 112), l2(2, 27);

   cout <<"First length: ";
   l1.disLength();
   cout <<"Second length:";
   l2.disLength();

   //overloading assignment operator.
   l1 = l2;
   cout << "First Length :";
   l1.disLength();

   return 0;
}
```

Output:

```
First length: 1Km 112M
Second length: 2Km 27M
First Length: 2Km 27M
```

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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
```

```
      void operator = (const Distance &D ) {
         feet = D.feet;
         inches = D.inches;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet <<  " I:" <<  inches << endl;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11);

   cout << "First Distance : ";
   D1.displayDistance();
   cout << "Second Distance :";
   D2.displayDistance();

   // use assignment operator
   D1 = D2;
   cout << "First Distance :";
   D1.displayDistance();

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

EXERCISE: 4

1. Types of inheritance:

The c++ classes can be derived in several ways. Based on that the inheritance can be divided in to Five types.

1. Single Inheritance.
2. Hierarchical Inheritance.

3. Multiple Inheritances.
4. Multi Level Inheritance.
5. Hybrid or Multi-path Inheritance.

i) Single Inheritance

AIM: Write a C++ program to illustrate Single Inheritance.

THEORY:

Single inheritance enables a derived class to inherit properties and behavior from a single parent class. It allows a derived class to inherit the properties and behavior of a base class, thus enabling code reusability as well as adding new features to the existing code. This makes the code much more elegant and less repetitive. Inheritance is one of the key features of object-oriented programming (OOP).

Single inheritance is safer than multiple inheritance if it is approached in the right way. It also enables a derived class to call the parent class implementation for a specific method if this method is overridden in the derived class or the parent class constructor.

The inheritance concept is used in many programming languages, including C++, Java, PHP, C#, and Visual Basic. To implement inheritance, C++ uses the ":" operator, while Java and PHP use the "extend" keyword, and Visual Basic uses the keyword "inherits." Java and C# enable single inheritance only, while other languages like C++ support multiple inheritance.

Single or Simple Inheritance

SOURCE CODE:

```
#include<iostream>
using namespace std;
class A
{
   protected:
   char name[10];
   int age;
};
class B:public A
{
   public:
   float h;
   int w;
   void get_data()
   {
      cout<<"Enter name and age:\n";
      cin>>name>>age;
      cout<<"\n Enter weight and height:\n";
      cin>>w>>h;
   }
   void show()
   {
      cout<<"Name: "<<name<<endl;
      cout<<"Age: "<<age<<endl;
      cout<<"Weight: "<<w<<endl;
      cout<<"Height: "<<h<<endl;
   }
};
int main()
{
   B C;
   C.get_data();
```

```
   C.show();
}
```

OUTPUT:

```
Enter name and age:
Radhika      52
Enter weight and height:
72          5.5
Name: Radhika
Age: 52
Weight: 72
Height: 5.5
```

ii) Hierarchical Inheritance.

AIM: Write a C++ program to incorporate Hierarchical Inheritance.

THEORY:

In Object Oriented Programming, the root meaning of inheritance is to establish a relationship between objects. In Inheritance, classes can inherit behavior and attributes from pre-existing classes, called **Base Classes or Parent Classes.** The resulting classes are known as **derived classes or child classes**.

So in Hierarchical Inheritance, we have 1 Parent Class and Multiple Child Classes, as shown in the pictorial representation given on this page, Inheritance. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level. One example could be

classification of accounts in a commercial bank or classification of students in a university.

In C++, such problems can be easily converted into hierarchies. The base class will include all the features that are common to the subclasses. A *subclass* can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.

A derived class with hierarchical inheritance is declared as follows:

```
class A {.....};          // Base class
class B: public A {.....};    // B derived from A
class C: public A {.....};    // C derived from A
```

Hierarchical Inheritance

This process can be extended to any number of levels. Let us understand this concept by a simple C++ program:

SOURCE CODE:

```
#include<iostream>
using namespace std;
class A
{
   protected:
   char name[20];
   int age;
};
class B:public A
{
   public:
```

```
    float h;
    int w;
    void get_data1()
    {
        cout<<"Enter name:";
        cin>>name;
        cout<<"Enter weight and height:";
        cin>>w>>h;
    }
    void show()
    {
        cout<<"This is class B and it is inherited from Class A\n";
        cout<<"Name: "<<name<<endl;
        cout<<"Weight: "<<w<<endl;
        cout<<"Height: "<<h<<endl;
    }
};
class C:public A
{
    public:
    char gender;
    void get_data2()
    {
        cout<<"Enter age:";
        cin>>age;
        cout<<"Enter gender:";
        cin>>gender;
    }
    void show()
    {
        cout<<"This is class C and it is inherited from class A\n";
        cout<<"Age: "<<age<<endl;
        cout<<"Gender: "<<gender<<endl;
```

```
    }
};
int main()
{
    B ob;
    C ob1;
    ob.get_data1();
    ob1.get_data2();
    ob.show();
    ob1.show();
}
```

OUTPUT:

```
Enter name: Ramu
Enter weight and height: 63    5.8
Enter age: 26
Enter gender: M
This is class B and it is inherited form class A
Name: Ramu
Weight: 63
Height: 5.8
This is class C and it is inherited form class A
Age: 26
Gender: M
```

iii) Multiple Inheritances.

AIM: Write a C+ Inheritance+ program to incorporate Multiple Inheritance

THEORY:

Multiple inheritance is a feature of some computer programming languages in which an object or class can inherit characteristics and features from more than one parent object or parent class. It is distinct from single inheritance, where an object or class may only inherit from one particular object or class.

Multiple Inheritance has been a sensitive issue for many years, with opponents pointing to its increased complexity and ambiguity in situations such as the "diamond problem", where it may be ambiguous as to which parent class a particular feature is inherited from if more than one parent class implements said feature. This can be addressed in various ways, including using virtual inheritance. Alternate methods of object composition not based on inheritance such as mixins and traits have also been proposed to address the ambiguity.

Multiple Inheritance

SOURCE CODE:

```
#include<iostream>
using namespace std;
class A
{
   protected:
   char name[20];
   int age;
};
class B
{
   protected:
   int w;
   float h;
};
class C:public A,B
```

```
{
   public:
   char g;
   void get_data()
   {
      cout<<"Enter name and age:\n";
      cin>>name>>age;
      cout<<"Enter weight and height:\n";
      cin>>w>>h;
      cout<<"Enter gender: ";
      cin>>g;
   }
   void show()
   {
      cout<<"\nName: "<<name<<endl<<"Age: "<<age<<endl;
      cout<<"Weight: "<<w<<endl<<"Height: "<<h<<endl;
      cout<<"Gender: "<<g<<endl;
   }
};
int main()
{
   C ob;
   ob.get_data();
   ob.show();
}
```

OUTPUT:

```
Enter name and age:
Mukesh        31
Enter weight and height:
64          5.9
Enter gender: M
```

Name: Mukesh
Age: 31
Weight: 64
Height: 5.9
Gender: M

iv) Multi-level inheritance.

AIM: Write a C++ program for incorporating multi-level inheritance.

THEORY:

Inheritance is the process of inheriting properties of objects of one class by objects of another class. The class which inherits the properties of another class is called Derived or Child or Sub class and the class whose properties are inherited is called Base or Parent or Super class. When a class is derived from a class which is also derived from another class, i.e. a class having more than one parent classes, such inheritance is called **Multilevel Inheritance**. The level of inheritance can be extended to any number of level depending upon the relation. Multilevel inheritance is similar to relation between grandfather, father and child.

Multi-Level Inheritance

SOURCE CODE:

```
#include<iostream>
using namespace std;
class A
{
   protected:
   char name[20];
```

```
   int age;
};
class B:public A
{
   protected:
   int w;
   float h;
};
class C:public B
{
   public:
   char g;
   void get_data()
   {
      cout<<"Enter name and age:\n";
      cin>>name>>age;
      cout<<"Enter weight and height:\n";
      cin>>w>>h;
      cout<<"Enter gender:";
      cin>>g;
   }
   void show()
   {
      cout<<"\nName: "<<name<<endl<<"Age: "<<age<<endl;
      cout<<"Weight: "<<w<<endl<<"Height: "<<h<<endl;
      cout<<"Gender: "<<g<<endl;
   }
};
int main()
{
   C ob;
   ob.get_data();
```

```
   ob.show();
}
```

OUTPUT:

```
Enter name and age:
Madhu     35
Enter weight and height:
58          5.6
Enter gender: F
Name: Madhu
Age: 35
Weight: 58
Height: 5.6
Gender: F
```

v) Hybrid inheritance.

AIM: Write a C++ program for incorporating Hybrid inheritance.

THEORY:

Hybrid inheritance is combination of two or more types of inheritance. It can also be called multi path inheritance.

For example, below diagram shows both Hierarchical Inheritance and multi level inheritance.

Hybrid or Multi-Path Inheritance

SOURCE CODE:

```
#include<iostream>
using namespace std;
class A   //Base class
{
   public:
   int l;
   void len()
   {
      cout<<"Lenght: ";
      cin>>l;         //Lenght is enter by user
   }
};
class B :public A //Inherits property of class A
{
   public:
   int b,c;
   void l_into_b()
   {
      len();
      cout<<"Breadth: ";
      cin>>b;       //Breadth is enter by user
      c=b*l;        //c stores value of lenght * Breadth i.e. (l*b).
   }
};
class C
{
   public:
   int h;
   void height()
   {
      cout<<"Height: ";
```

```
      cin>>h;       //Height is enter by user
   }
};
class D:public B,public C  //Hybrid Inheritance Level
{
   public:
   int res;
   void volume()
   {
      l_into_b();
      height();
      res=h*c;
      cout<<"Volume is (l*b*h): "<<res<<endl;
   }
   void area()
   {
      l_into_b();
      cout<<"Area is (l*b): "<<c<<endl;
   }
};
int main()
{
   D d1;
   cout<<"Enter dimensions of object to get Area:\n";
   d1.area();
   cout<<"Enter values of object to get Volume:\n";
   d1.volume();
   return 0;
}
```

OUTPUT:

Enter dimensions of object to get Area:
Length: 63
Breadth: 23
Area is (l * b): 1449
Enter dimensions of object to get Volume:
Length: 12
Breadth: 27
Height: 14
Volume is (l * b * h): 4536

2. Also illustrate the order of execution of constructors and destructors in inheritance

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

class base1
{
public:
  base1 (void)
  {
    cout << " constructor of class base1\n";


  }
   ~base1 ()
  {
     cout << " destructor of class base1\n";



  }

};

class base2
{

   public:
   base2(void)
   {
      cout << " constructor of class base2\n";
```

```
  }
  ~base2()
  {

     cout << " destructor of class base2\n";

  }

};
class derive1 : public base1, public base2
{

  public:
  derive1(void)
  {
   cout << " constructor of class derive1\n";

  }
  ~derive1()
  {
     cout << " destructor of class derive1\n";


  }
};

main ()
{
  derive1 x;
  cout << " Destructors are: "<<endl;


}
```

EXERCISE : 5

1 a) AIM: Write a C++ program to illustrate template class.

THEORY:

Template is simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write same code for different data types. For example a software company may need sort() for different data types.

Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by keyword 'class'.

Templates are expended at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

SOURCE CODE:

```
#include<iostream>
using namespace std;
template <class T>
class data
{
   public:
   data(T c)
   {
      cout<<"C= size in bites "<<sizeof (c)<<endl;
   }
};
int main()
{
   data<char>h('A');
   data<int>i(100);
   data<float>f(3.12);
}
```

OUTPUT:

C= size in bites 1
C= size in bites 4
C= size in bites 4

1 B) **AIM:** Write a c++ program to illustrate member function template.

THEORY:

In Function Templates, a function template was defined outside of any template class. However, functions in C++ are often member functions of a class

You can define template member functions three ways:

1. Explicitly at file scope for each type used to instantiate the template class.
2. At file scope with the template arguments.
3. Inlined in the class template itself.

SOURCE CODE:

```
#include<iostream>
using namespace std;
template<class E>
void exchange (E&a,E&b)
{
   E temp=a;
   a=b;
   b=temp;
};
```

```
int main()
{
   int x=5,y=8;
   float a,b;
   cout<<"Enter two values:\n";
   cin>>a>>b;
   cout<<"Before exchange\n x="<<x<<"\ty="<<y<<endl;
   exchange(x,y);
   cout<<"After exchange\n x="<<x<<"\ty="<<y<<endl;
   cout<<"Before exchange\n a="<<a<<"\tb="<<b<<endl;
   exchange(a,b);
   cout<<"After exchange\n a="<<a<<"\tb="<<b<<endl;
   return 0;
}
```

OUTPUT:

```
Enter two values:
63.25    52.36
Before exchange
x= 5      y= 8
After exchange
x= 8      y=5
Before exchange
a= 63.25          b= 52.36
After exchange
a= 52.36          b= 63.25
```

Definition

perform exception handling for Divide by zero Exception.

Exception Handling Divide by zero Algorithm/Steps:

- Step 1: Start the program.
- Step 2: Declare the variables a,b,c.
- Step 3: Read the values a,b,c,.
- Step 4: Inside the try block check the condition. (a. if(a-b!=0) then calculate the value of d and display.b. otherwise throw the exception.)
- Step 5: Catch the exception and display the appropriate message.
- Step 6: Stop the program.

1 c) Write a Program for Exception Handling Divide by zero

Exception Handling Divide by zero Example Program

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

void main() {
    int a, b, c;
    float d;
    clrscr();
    cout << "Enter the value of a:";
    cin>>a;
    cout << "Enter the value of b:";
    cin>>b;
    cout << "Enter the value of c:";
    cin>>c;

    try {
```

```
        if ((a - b) != 0) {
            d = c / (a - b);
            cout << "Result is:" << d;
        } else {
            throw (a - b);
        }
    } catch (int i) {
        cout << "Answer is infinite because a-b is:" << i;
    }

    getch();
}
```

Sample Output

```
Enter the value for a: 20
Enter the value for b: 20
Enter the value for c: 40
```

1 D) AIM: Write a Program to rethrow an Exception

Theory:

If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression (throwwithout *assignment_expression*) causes the originally thrown object to be rethrown.

Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next dynamically enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred. Any catch blocks for the dynamically enclosing try block have an opportunity to catch the exception.

Program:

```
#include<iostream>
using namespace std;
void sub(int i,int j)
{
   try
   {
      if(i==0)
      {
         throw i;
      }
      else
         cout<<"Subtraction result is: "<<i-j<<endl;
   }
   catch(int i)
   {
      cout<<"Exception caught inside sub()\n";
      throw;
   }
};
int main()
{
   try
   {
      sub(8,4);
      sub(0,8);
   }
   catch(int k)
   {
      cout<<"Exception caught inside main()\n";
   }
```

```
    return 0;
}
```

OUTPUT:

```
Subtraction result is: 4
Exception caught inside sub()
Exception caught inside main()
```

EXERCISE : 6

1. Write a C++ program illustrating user defined string processing functions using pointers (string length, string copy, string concatenation)

```
1. #include <iostream>
2. using namespace std;
3. int main( ) {
4.     string s1 = "Hello";
5.         char ch[] = { 'C', '+', '+'};
6.         string s2 = string(ch);
7.         cout<<s1<<endl;
8.         cout<<s2<<endl;
9. }
```

Output:

```
Hello
C++
```

```
1. #include <iostream>
2. #include <cstring>
```

```
3.  using namespace std;
4.  int main ()
5.  {
6.    char key[] = "mango";
7.    char buffer[50];
8.    do {
9.       cout<<"What is my favourite fruit? ";
10.     cin>>buffer;
11.   } while (strcmp (key,buffer) != 0);
12.  cout<<"Answer is correct!!"<<endl;
13.   return 0;
14. }
15.
```

Output:

```
16.  What is my favourite fruit? apple
17.  What is my favourite fruit? banana
18.  What is my favourite fruit? mango
19.  Answer is correct!!
```

```
20. #include <iostream>
21. #include <cstring>
22. using namespace std;
23. int main()
24. {
25.     char key[25], buffer[25];
26.     cout << "Enter the key string: ";
27.     cin.getline(key, 25);
28.     cout << "Enter the buffer string: ";
29.      cin.getline(buffer, 25);
30.     strcat(key, buffer);
31.     cout << "Key = " << key << endl;
32.     cout << "Buffer = " << buffer<<endl;
33.     return 0;
34. }
```

Output:

```
Enter the key string: Welcome to
Enter the buffer string:  C++ Programming.
Key = Welcome to C++ Programming.
Buffer =  C++ Programming.
```

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char key[25], buffer[25];
    cout << "Enter the key string: ";
    cin.getline(key, 25);
    strcpy(buffer, key);
    cout << "Key = "<< key << endl;
    cout << "Buffer = "<< buffer<<endl;
    return 0;
}
```

Output:

```
Enter the key string: C++ Tutorial
Key = C++ Tutorial
Buffer = C++ Tutorial
```

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char ary[] = "Welcome to C++ Programming";
    cout << "Length of String = " << strlen(ary)<<endl;
    return 0;
}
```

Output:

```
Length of String = 26
```

2. **Write a C++ program illustrating Virtual classes & virtual functions.**

C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.

- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

```
#include <iostream>
using namespace std;
class A
{
   int x=5;
    public:
    void display()
    {
        std::cout << "Value of x is : " << x<<std::endl;
    }
};
class B: public A
{
    int y = 10;
    public:
    void display()
    {
        std::cout << "Value of y is : " <<y<< std::endl;
    }
};
int main()
```

```
22. {
23.     A *a;
24.     B b;
25.     a = &b;
26.    a->display();
27.     return 0;
28. }
```

Output:

```
Value of x is : 5
```

3. Write C++ program that implement Bubble sort, to sort a given list of integers in ascending order

Bubble Sort Technique

Using the bubble sort technique, sorting is done in passes or iteration. Thus at the end of each iteration, the heaviest element is placed at its proper place in the list. In other words, the largest element in the list bubbles up.

We have given a general algorithm of bubble sort technique below.

General Algorithm

Step 1: For i = 0 to N-1 repeat Step 2
Step 2: For J = i + 1 to N – I repeat
Step 3: if A[J] > A[i]
Swap A[J] and A[i]
[End of Inner for loop]
[End if Outer for loop]
Step 4: Exit
Here is a pseudo-code for bubble sort algorithm, where we traverse the list using two iterative loops.

In the first loop, we start from the 0th element and in the next loop, we start from an adjacent element. In the inner loop body, we compare each of the adjacent elements and swap them if they are not in order. At the end of each iteration of the outer loop, the heaviest element bubbles up at the end.

Pseudocode

Procedure bubble_sort (array , N)

array – list of items to be sorted

```
        N – size of array
begin
        swapped = false
        repeat
                for I = 1 to N-1
                        if array[i-1] > array[i] then
                                swap array[i-1] and array[i]
                                swapped = true
                        end if
                end for
        until not swapped
end procedure
```

The above given is the pseudo-code for bubble sort technique. Let us now illustrate this technique by using a detailed illustration.

Illustration

We take an array of size 5 and illustrate the bubble sort algorithm.

Array entirely sorted.

The above illustration can be summarized in a tabular form as shown below:

Pass	Unsorted list	comparison	Sorted list

Pass	Unsorted list	comparison	Sorted list
1	{10,5,15,0,12}	{10,5}	{5,10,15,0,12}
	{5,10,15,0,12}	{10,15}	{5,10,15,0,12}
	{5,10,15,0,12}	{15,0}	{5,10,0,15,12}
	{5,10,0,15,12}	{15,12}	{5,10,0,12,15}
2	{5,10,0,12,15}	{5,10}	{5,10,0,12,15}
	{5,10,0,12,15}	{10,0}	{5,0,10,12,15}
	{5,0,10,12,15}	{10,12}	{5,0,10,12,15}
3	{5,0,10,12,15}	{5,0}	{0,5,10,12,15}
	{5,0,10,12,15}	{5,10}	{5,0,10,12,15}
	{5,0,10,12,15}	SORTED	

As shown in the illustration, with every pass, the largest element bubbles up to the last thereby sorting the list with every pass. As mentioned in the introduction, each element is compared to its adjacent element and swapped with one another if they are not in order.

Thus as shown in the illustration above, at the end of the first pass, if the array is to be sorted in ascending order, the largest element is placed at the end of the list. For the second pass, the second largest element is placed at the second last position in the list and so on.

When we reach N-1 (where N is a total number of elements in the list) passes, we will have the entire list sorted.

Bubble sort technique can be implemented in any programming language. We have implemented the bubble sort algorithm using C++ and Java language below.

C++ Example

Let us see a programming Example to demonstrate the bubble sort.

```
#include<iostream>

using namespace std;

int main ()

{

   int i, j,temp,pass=0;

   int a[10] = {10,2,0,14,43,25,18,1,5,45};

   cout <<"Input list ...\n";

   for(i = 0; i<10; i++) {

      cout <<a[i]<<"\t";

   }

cout<<endl;

for(i = 0; i<10; i++) {

   for(j = i+1; j<10; j++)

   {

      if(a[j] < a[i]) {

         temp = a[i];

         a[i] = a[j];

         a[j] = temp;

      }

   }

pass++;

}
```

```
cout <<"Sorted Element List ...\n";

for(i = 0; i<10; i++) {

   cout <<a[i]<<"\t";

}

cout<<"\nNumber of passes taken to sort the list:"<<pass<<endl;

return 0;

}
```

Output:

Input list …

10 2 0 14 43 25 18 1 5 45

Sorted Element List …

0 1 2 5 10 14 18 25 43 45

Number of passes taken to sort the list:10