Experiment-1: **CONTROL FLOW**

1.1) Develop a C++ program to find the roots of a quadratic equation.

Code:

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
   float a, b, c, x1, x2, discriminant, realPart, imaginaryPart;
   cin >> a >> b >> c;
   discriminant = b*b - 4*a*c;
   if (discriminant > 0)
         {
      x1 = (-b + sqrt(discriminant)) / (2*a);
      x2 = (-b - sqrt(discriminant)) / (2*a);
      cout << "Roots are real and different." << endl;
      cout << "x1 = " << x1 << endl;
      cout << "x2 = " << x2 << endl;
   }
   else if (discriminant == 0)
         {
      cout << "Roots are real and same." << endl;
      x1 = -b/(2*a);
      cout << "x1 = x2 = " << x1 << endl;
   }
   else
         {
      realPart = -b/(2*a);
      imaginaryPart =sqrt(-discriminant)/(2*a);
      cout << "Roots are complex and different."  << endl;
      cout << "x1 = " << realPart << "+" << imaginaryPart << "i" << endl;
      cout << "x2 = " << realPart << "-" << imaginaryPart << "i" << endl;
   }
   return 0;
}
```

Output:

```
1 -4 4
Roots are real and same.
x1 = x2 = 2
```

1.2) Develop a C++ program to find factorial of a given number using recursion.

Code:

```
#include <iostream>
using namespace std;
int fact(int num)
{
	if(num==1)
	{
		return 1;
	}
	return num*fact(num-1);
}
int main()
{
	int num;
	cin>>num;
	cout<<fact(num);
	return 0;
}
```

Output:

```
6
720
```

<u>Experiment-2:</u> VARAIBLE AND SCOPE

2.1) Develop a C++ program to illustrate scope resolution and namespaces.

<u>Code:</u>

```
#include<iostream>
using namespace std;
// declare global variable
int num = 50;
int main ()
{
        // declare local variable
        int num = 100;
        // print the value of the variables
        cout << "The value of the local variable num: " << num;
        // use scope resolution operator (::) to access the global variable
        cout << "\nThe value of the global variable num: " << ::num;
        return 0;
}
```

<u>Output:</u>

```
The value of the local variable num: 100
The value of the global variable num: 50
```

2.2) Develop a C++ program illustrating Inline Functions.

<u>Code:</u>

```
#include<iostream>
using namespace std;
inline void find_area(float radius)
{
        cout<<3.14*radius*radius;
}
int main()
{
    float radius,area;
        cin>>radius;
    find_area(radius);
}
```

<u>Output:</u>

```
4
50.24
```

Experiment-3: CLASS AND OBJECT

3.1) Develop a C++ program demonstrating a Bank Account with necessary data members and member functions.

Code:

```
#include<iostream>
using namespace std;
class BankAccount
{
	public:
		string name,accno,ifscno;
		int bal,pin;
		void setdata(string n,string ano,string ifsc,int b,int p)
		{
			name=n;
			accno=ano;
			ifscno=ifsc;
			bal=b;
			pin=p;
		}
		void display()
		{
			cout<<name<<" "<<accno<<" "<<ifscno<<" "<<bal<<" "<<pin;
		}
};
int main()
{
	BankAccount b1;
	b1.setdata("Ravi","5654563","SBI556",12000,1234);
	b1.display();
}
```

Output:

```
Ravi 5654563 SBI556 12000 1234
```

3.2) Develop a C++ program for illustrating Access Specifiers :public and private.

Code:

```
#include<iostream>
using namespace std;
class BankAccount
{
	private:
		int bal,pin;
	public:
		string name,accno,ifscno;
		void setdata(string n,string ano,string ifsc,int b,int p)
		{
			name=n;
			accno=ano;
			ifscno=ifsc;
			bal=b;
```

```
                pin=p;
            }
            void display()
            {
                cout<<name<<" "<<accno<<" "<<ifscno<<" "<<bal<<" "<<pin;
            }
};
int main()
{
        BankAccount b1;
        b1.setdata("Ravi","5654563","SBI556",12000,1234);
        b1.display();
}
```

Output:

```
Ravi 5654563 SBI556 12000 1234
```

3.3) Develop a C++ program to illustrate this pointer.

Code:

```
#include<iostream>
using namespace std;
class BankAccount
{
        private:
            string name,accno,ifscno;
            int bal,pin;
        public:
            void setdata(string name,string accno,string ifscno,int bal,int pin)
            {
                this->name=name;
                this->accno=accno;
                this->ifscno=ifscno;
                this->bal=bal;
                this->pin=pin;
            }
            void display()
            {
                cout<<this->name<<" "<<this->accno<<" "<<this->ifscno<<"
"<<this->bal<<" "<<this->pin;
            }
};
int main()
{
        BankAccount b1;
        b1.setdata("Ravi","5654563","SBI556",12000,1234);
        b1.display();
}
```

Output:

```
Ravi 5654563 SBI556 12000 1234
```

Experiment-4: **FUNCTIONS**

4.1) Develop a C++ program illustrate function overloading.

Code:

```
#include<iostream>
using namespace std;
class BankAccount
{
	private:
		string name,accno,ifscno;
		int bal,pin;
	public:
		void setdata(string name,string accno,string ifscno,int bal,int pin)
		{
			this->name=name;
			this->accno=accno;
			this->ifscno=ifscno;
			this->bal=bal;
			this->pin=pin;
		}
		void display()
		{
			cout<<this->name<<" "<<this->accno<<" "<<this->ifscno<<"
"<<this->bal<<" "<<this->pin<<endl;
		}
		void setpin(int pin)
		{
			this->pin=pin;
		}
		void setpin()
		{
			this->pin=4321;
		}
};
int main()
{
	BankAccount b1;
	b1.setdata("Ravi","5654563","SBI556",12000,1234);
	b1.display();
	b1.setpin();
	b1.display();
	b1.setpin(1298);
	b1.display();
}
```

Output:

```
Ravi 5654563 SBI556 12000 1234
Ravi 5654563 SBI556 12000 4321
Ravi 5654563 SBI556 12000 1298
```

4.2) Develop a C++ program to illustrate the use of default arguments.

Code:

```
#include<iostream>
using namespace std;
class BankAccount
{
	private:
		string name,accno,ifscno;
		int bal,pin;
	public:
		void setdata(string name,string accno,string ifscno,int bal,int pin)
		{
			this->name=name;
			this->accno=accno;
			this->ifscno=ifscno;
			this->bal=bal;
			this->pin=pin;
		}
		void display()
		{
			cout<<this->name<<" "<<this->accno<<" "<<this->ifscno<<" 
"<<this->bal<<" "<<this->pin<<endl;
		}
		void setpin(int pin=4321)
		{
			this->pin=pin;
		}
};
int main()
{
	BankAccount b1;
	b1.setdata("Ravi","5654563","SBI556",12000,1234);
	b1.display();
	b1.setpin();
	b1.display();
	b1.setpin(1298);
	b1.display();
}
```

Output:

```
Ravi 5654563 SBI556 12000 1234
Ravi 5654563 SBI556 12000 4321
Ravi 5654563 SBI556 12000 1298
```

4.3) Develop a C++ program illustrating friend function.

<u>Code:</u>

```
#include<iostream>
using namespace std;
class BankAccount
{
        private:
                string name,accno,ifscno;
                int bal,pin;
        public:
                void setdata(string name,string accno,string ifscno,int bal,int pin)
                {
                        this->name=name;
                        this->accno=accno;
                        this->ifscno=ifscno;
                        this->bal=bal;
                        this->pin=pin;
                }
                void display()
                {
                        cout<<this->name<<" "<<this->accno<<" "<<this->ifscno<<"
"<<this->bal<<" "<<this->pin<<endl;
                }
                friend void setpin(BankAccount&,int pin);
};
void setpin(BankAccount &b,int pin=4321)
{
        b.pin=pin;
}
int main()
{
        BankAccount b1;
        b1.setdata("Ravi","5654563","SBI556",12000,1234);
        b1.display();
        setpin(b1);
        b1.display();
        setpin(b1,1298);
        b1.display();
}
```

<u>Output:</u>

```
Ravi 5654563 SBI556 12000 1234
Ravi 5654563 SBI556 12000 4321
Ravi 5654563 SBI556 12000 1298
```

Experiment-5: CONSTRUCTOR AND DESTRUCTOR

5.1) Develop a C++ Program to illustrate the use of Constructors and Destructors.

Code:

```
#include<iostream>
using namespace std;
class BankAccount
{
	private:
		string name,accno,ifscno;
		int bal,pin;
	public:
		BankAccount(string name,string accno,string ifscno,int bal,int pin)
		{
			this->name=name;
			this->accno=accno;
			this->ifscno=ifscno;
			this->bal=bal;
			this->pin=pin;
		}
		void display()
		{
			cout<<this->name<<" "<<this->accno<<" "<<this->ifscno<<"
"<<this->bal<<" "<<this->pin<<endl;
		}
		friend void setpin(BankAccount&,int pin);
		~BankAccount()
		{
			cout<<"Object Destructed"<<endl;
		}
};
void setpin(BankAccount &b,int pin=4321)
{
	b.pin=pin;
}
int main()
{
	BankAccount b1("Ravi","5654563","SBI556",12000,1234);
	b1.display();
	setpin(b1);
	b1.display();
	setpin(b1,1298);
	b1.display();
}
```

Output:

```
Ravi 5654563 SBI556 12000 1234
Ravi 5654563 SBI556 12000 4321
Ravi 5654563 SBI556 12000 1298
Object Destructed
```

5.2) Develop a C++ program illustrating Constructor overloading.

Code:

```
#include<iostream>
using namespace std;
class BankAccount
{
	private:
		string name,accno,ifscno;
		int bal,pin;
	public:
		BankAccount(string name,string accno,string ifscno)
		{
			this->name=name;
			this->accno=accno;
			this->ifscno=ifscno;
			this->bal=0;
			this->pin=9876;
		}
		BankAccount(string name,string accno,string ifscno,int bal,int pin)
		{
			this->name=name;
			this->accno=accno;
			this->ifscno=ifscno;
			this->bal=bal;
			this->pin=pin;
		}
		void display()
		{
			cout<<this->name<<" "<<this->accno<<" "<<this->ifscno<<"
"<<this->bal<<" "<<this->pin<<endl;
		}
		~BankAccount()
		{
			cout<<"Object Destructed"<<endl;
		}
};
int main()
{
	BankAccount b1("Ravi","5654563","SBI556",12000,1234);
	b1.display();
	BankAccount b2("Suresh","234565","SBI556");
	b2.display();
}
```

Output:

```
Ravi 5654563 SBI556 12000 1234
Suresh 234565 SBI556 0 9876
Object Destructed
Object Destructed
```

5.3) Develop a C++ program illustrating Copy Constructor.

Code:

```
#include<iostream>
using namespace std;
class BankAccount
{
	private:
		string name,accno,ifscno;
		int bal,pin;
	public:
		BankAccount(string name,string accno,string ifscno,int bal,int pin)
		{
			this->name=name;
			this->accno=accno;
			this->ifscno=ifscno;
			this->bal=bal;
			this->pin=pin;
		}
		BankAccount(BankAccount &b)
		{
			this->name=b.name;
			this->accno=b.accno;
			this->ifscno=b.ifscno;
			this->bal=b.bal;
			this->pin=b.pin;
		}
		void display()
		{
			cout<<this->name<<" "<<this->accno<<" "<<this->ifscno<<" 
"<<this->bal<<" "<<this->pin<<endl;
		}
		~BankAccount()
		{
			cout<<"Object Destructed"<<endl;
		}
};
int main()
{
	BankAccount b1("Ravi","5654563","SBI556",12000,1234);
	b1.display();
	BankAccount b2(b1);// copy constructor
	b2.display();
}
```

Output:

```
Ravi 5654563 SBI556 12000 1234
Ravi 5654563 SBI556 12000 1234
Object Destructed
Object Destructed
```

<u>Experiment-6:</u> OPERATOR OVERLOADING

6.1) Develop a C++ program to Overload Unary, and Binary Operators using member function.

<u>Code:</u>

```
#include<iostream>
using namespace std;
class BankAccount
{
	private:
		string name,accno,ifscno;
		int bal,pin;
	public:
		BankAccount(string name,string accno,string ifscno,int bal,int pin)
		{
			this->name=name;
			this->accno=accno;
			this->ifscno=ifscno;
			this->bal=bal;
			this->pin=pin;
		}
		void display()
		{
			cout<<this->name<<" "<<this->accno<<" "<<this->ifscno<<" 
"<<this->bal<<" "<<this->pin<<endl;
		}
		void operator ++()//Unary operator overloading
		{
			++bal;
		}
		int operator +(BankAccount &B2)//Binary operator overloading{
			return bal+B2.bal;}
};
int main()
{
	BankAccount b1("Ravi","5654563","SBI556",12000,1234);
	b1.display();
	BankAccount b2("Suresh","234565","SBI556",1500,8765);
	b2.display();
	//amount in this bank
	int amount=b1+b2;//Binary operator overloading call
	cout<<amount<<endl;
	++b1;// Unary operator overloading call
	b1.display();
}
```

<u>Output:</u>

```
Ravi 5654563 SBI556 12000 1234
Suresh 234565 SBI556 1500 8765
13500
Ravi 5654563 SBI556 12001 1234
```

6.2) Develop a C++ program to Overload Unary, and Binary Operators using friend function.

Code:

```
#include<iostream>
using namespace std;
class BankAccount
{
	private:
		string name,accno,ifscno;
		int bal,pin;
	public:
		BankAccount(string name,string accno,string ifscno,int bal,int pin)
		{
			this->name=name;
			this->accno=accno;
			this->ifscno=ifscno;
			this->bal=bal;
			this->pin=pin;
		}
		void display()
		{
			cout<<this->name<<" "<<this->accno<<" "<<this->ifscno<<"
"<<this->bal<<" "<<this->pin<<endl;
		}
		friend void operator ++(BankAccount &B);
		friend int operator +(BankAccount &B1,BankAccount &B2);

};
void operator ++(BankAccount &B){
	B.bal++;}
int operator +(BankAccount &B1,BankAccount &B2){
	return B1.bal+B2.bal;}
int main()
{
	BankAccount b1("Ravi","5654563","SBI556",12000,1234);
	b1.display();
	BankAccount b2("Suresh","234565","SBI556",1500,8765);
	b2.display();
	//amount in this bank
	int amount=b1+b2;//Binary operator overloading call
	cout<<amount<<endl;
	++b1;// Unary operator overloading call
	b1.display();
}
```

Output:

```
Ravi 5654563 SBI556 12000 1234
Suresh 234565 SBI556 1500 8765
13500
Ravi 5654563 SBI556 12001 1234
```

6.3) Develop a case study on Overloading Operators and Overloading Functions.

OPERATOR OVERLOADING

Operator overloading is x syntactic sugar, and is used because it allows programming using notation nearer to the target domain and allows user-defined types a similar level of syntactic support as types built into a language. It is common, for example, in scientific computing, where it allows computing representations of mathematical objects to be manipulated with the same syntax as on paper.

Operator overloading does not change the expressive power of a language (with functions), as it can be emulated using function calls. For example, consider variables a, b, c of some user-defined type, such as matrices:

```
a + b * c
```

In a language that supports operator overloading, and with the usual assumption that the '*' operator has higher precedence than '+' operator, this is a concise way of writing:

```
add (a, multiply (b,c))
```

However, the former syntax reflects common mathematical usage.

FUNCTION OVERLOADING

Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading.

In Function Overloading "Function" name should be the same and the arguments should be different. Function overloading can be considered as an example of
a polymorphism feature in C++.
The parameters should follow any one or more than one of the following conditions for Function overloading:
Parameters should have a different type.

```
add(int a, int b)
add(double a, double b)
```

The example code shows how function overloading can be used. As functions do practically the same thing, it makes sense to use function overloading.

```
function int generateNumber(int MaxValue) {
return rand * MaxValue
}
```

```
function int generateNumber(int MinValue, int MaxValue) {
return MinValue + rand * (MaxValue – MinValue)
}
```

Experiment-7: INHERITANCE

7.1) Develop C++ Programs to incorporate various forms of Inheritance

C++ classes can be reused in several ways. Once a class been written and tested it can be adapted by other programmers to suit their requirements. This is basically done by creating new class reuse of the properties of existing class. The mechanism of deriving a new class from an existing one is called INHERITANCE.

The old class is refer to as “BASE Class” and the new one is called “Derived Class”.

Syntax:

```
class derived_class: (visibility mode) Base_class
{
    Coding of Derived_class;
};
```

The colon (:) indicates that the derived class is derived from base class. the visibility mode is optional. If it is present may be either private. The default visibility mode is private. The visibility mode is specifies whether the features of the base class are privately derived or publicly derived.

When a base class is privately inherited by a base class, “public members of the base class become “Private members of the derived class.

When a base class is publicly inherits by a derived class, “Public members “ of the base class become “public members” of the derived class.

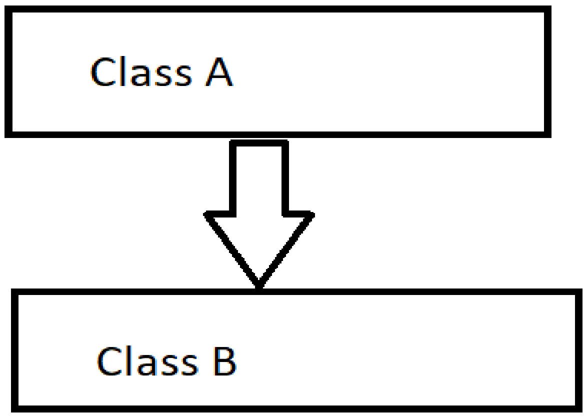
Types of inheritance:

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 Inheritance.
4. Multi-level Inheritance.
5. Hybrid Inheritance.

i. Single Inheritance

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.



Code:

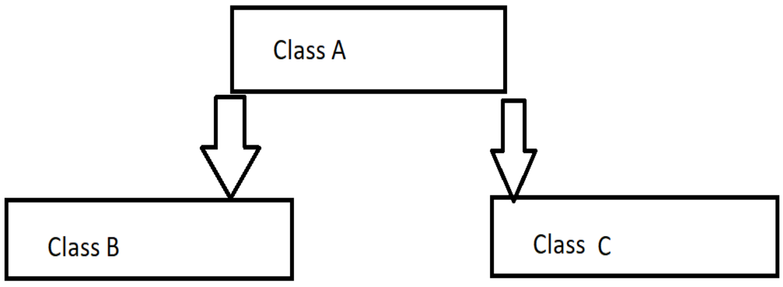
```
#include<iostream>
using namespace std;
class A
{
   protected:
   char name[10];
   int age;
};
class B:public A
{
   public:
   float h;
   int w;
   void get_data()
   {
      cout<<"Enter name and age:\n";
      cin>>name>>age;
      cout<<"\n Enter weight and height:\n";
      cin>>w>>h;
  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

In Inheritance, classes can inherit behaviour 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.

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
```

Code:

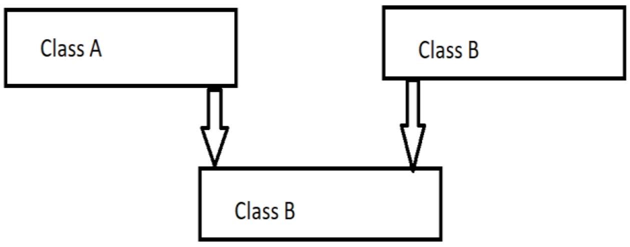
```
#include<iostream>
using namespace std;
class A
{
   protected:
   char name[20];
   int age;
};
class B:public A
{
   public:
   float h;
   int w;
   void get_data1()
   {
      cout<<"Enter name:";
      cin>>name;
      cout<<"Enter weight and height:";
      cin>>w>>h;
   }
   void show()
   {
      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 Inheritance



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.

Code:

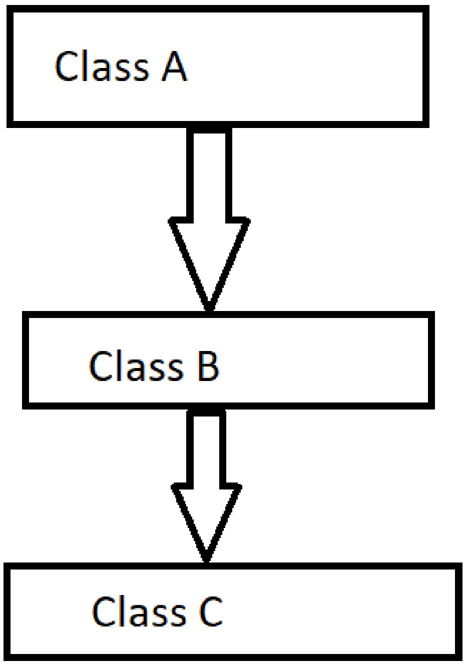
```
#include<iostream>
using namespace std;
class A
{
   protected:
   char name[20];
   int age;
};
class B
{
   protected:
   int w;
   float h;
};
class C:public A,B
{
   public:
   char g;
   void get_data()
   {
      cout<<"Enter name and age:\n";
      cin>>name>>age;
      cout<<"Enter weight 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

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.



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();
}
```

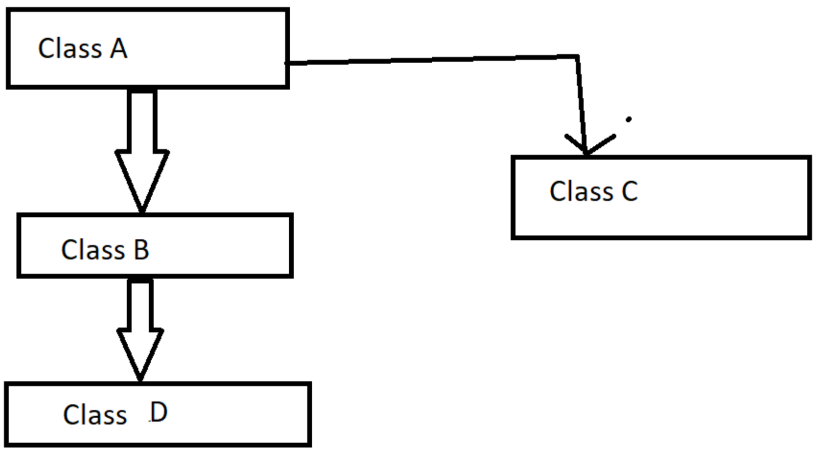
Output:

```
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

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.



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
```

7.2) Develop a C++ program in C++ to illustrate the order of execution of constructors and destructors in inheritance.

Code:

```
#include <iostream>
using namespace std;
class A
{
        public:
                A()
                {
                        cout<<"A's Constructor"<<endl;
                }
                ~A()
                {
                        cout<<"A's Destructor"<<endl;
                }
};
class B : A
{
        public:
                B(){
                        cout<<"B's Constructor"<<endl;
                }
                ~B(){
                        cout<<"B's Destructor"<<endl;
                }
};
class C : B
{
        public:
                C(){
                        cout<<"C's Constructor"<<endl;
                }
                ~C(){
                        cout<<"C's Destructor"<<endl;
                }
};
int main()
{
        C c;
}
```

Output:

```
A's Constructor
B's Constructor
C's Constructor
C's Destructor
B's Destructor
A's Destructor
```

<u>Experiment-8:</u> POINTERS

8.1) Develop a C++ program to illustrate object as a class member.

<u>Code:</u>

```
#include <iostream>
using namespace std;
class Personal{
	public:
		string name,city;
		Personal(string n,string c){
			name=n;
			city=c;
		}
};
class Customer{
	public:
		string accno,ifsc;
		int bal;
		Personal *d;
		Customer(string ano,string icode,int b,Personal D){
			accno=ano;
			ifsc=icode;
			bal=b;
			d=&D;
		}
		void show(){
			cout<<accno<<" "<<ifsc<<" "<<bal<<" "<<d->name<<" "<<d->city;
		}
};
int main()
{
	Personal p("Sudhir","KKD");
	Customer c1("s34565123","sbi123",1200,p);
	c1.show();
}
```

<u>Output:</u>

```
s34565123 sbi123 1200 Sudhir KKD
```

8.2) Develop a C++ program to illustrate pointer to a class.

<u>Code:</u>

```
#include<iostream>
using namespace std;
class BankAccount{
	private:
		string name,accno,ifscno;
		int bal,pin;
	public:
		BankAccount(string name,string accno,string ifscno,int bal,int pin){
			this->name=name;
			this->accno=accno;
```

```
                        this->ifscno=ifscno;
                        this->bal=bal;
                        this->pin=pin;
                }
                void display()
                {
                        cout<<this->name<<" "<<this->accno<<" "<<this->ifscno<<"
"<<this->bal<<" "<<this->pin<<endl;
                }
};
int main()
{
        BankAccount *b2,b1("Ravi","5654563","SBI556",12000,1234);
        b1.display();
        b2=&b1;
        b2->display();
}
```

<u>Output:</u>

```
Ravi 5654563 SBI556 12000 1234
Ravi 5654563 SBI556 12000 1234
```

8.3) Develop a C++ program to illustrate Virtual Base Class.

<u>Code:</u>

```
#include <iostream>
using namespace std;
class A{
        public:
                int a;
                A()
                {
                        a = 10;
                }
};
class B : public virtual A {
};
class C : public virtual A {
};
class D : public B, public C {
};
int main()
{
        D object; // object creation of class d
        cout << object.a << endl;
        return 0;
}
```

<u>Output:</u>

```
10
```

<u>Experiment-9:</u> POLYMORPHISM

9.1) Develop a C++ program to illustrate virtual functions.

<u>Code:</u>

```
#include<iostream>
using namespace std;
class Personal
{
	public:
		string name;
		Personal(string name)
		{
			this->name=name;
		}
		virtual void display()
		{
			cout<<name<<endl;
		}
};
class Student:public Personal
{
	public:
		string rollno;
		Student(string name,string rollno):Personal(name)
		{

			this->rollno=rollno;
		}
		void display()
		{
			cout<<rollno<<" ";
			Personal::display();
		}
};
int main()
{
	Personal *obj;
	Student s1("sudhir","123");
	obj=&s1;
	obj->display();
}
```

<u>Output:</u>

```
123 sudhir
```

9.2) Develop a C++ program to illustrate runtime polymorphism.

Code:

```
#include<iostream>
using namespace std;
class base
{
        public:
                virtual void print()
                {
                        cout << "print base class" <<endl;
                }

                void show()
                {
                cout << "show base class" <<endl;
                }
};
class derived:public base
{
        public:
                // print () is already virtual function in
                // derived class, we could also declared as
                // virtual void print () explicitly
                void print()
                {
                        cout << "print derived class" <<endl;
                }
                void show()
                {
                cout << "show derived class" <<endl;
                }
};
int main()
{
        base* bptr;
        derived d;
        bptr = &d;
        // Virtual function, binded at
        // runtime (Runtime polymorphism)
        bptr->print();
        // Non-virtual function, binded
        // at compile time
        bptr->show();
        return 0;
}
```

Output:

```
print derived class
show base class
```

9.3) Develop a C++ program to illustrate pure virtual function and calculate the area of different shapes by using abstract class.

Code:

```
#include<iostream>
using namespace std;
class Shape{
	public:
		virtual void calarea()=0;
};
class Circle:public Shape{
	public:
		int r;
		float area;
		Circle(int r)
		{
			this->r=r;
		}
		void calarea()
		{
			area=3.141*r*r;
		}
};
class Triangle:public Shape{
	public:
		int b,h;
		float area;
		Triangle(int b,int h)
		{
			this->b=b;
			this->h=h;
		}
		void calarea()
		{
			area=(b*h)/2;
		}
};
int main()
{
	Circle c(10);
	c.calarea();
	cout<<c.area<<endl;
	Triangle T(2,3);
	T.calarea();
	cout<<T.area;
}
```

Output:

```
314.1
3
```

Experiment-10: **TEMPLATES**

10.1) Develop a C++ Program illustrating function template.

Code:

```
#include<iostream>
using namespace std;
template <typename T>
T sum(T a,T b)
{
	return a+b;
}
int main()
{
	int a,b;
	float c,d;
	cin>>a>>b;
	cin>>c>>d;
	cout<<sum(a,b)<<endl;
	cout<<sum(c,d);
}
```

Sample Input:

```
1 2
3.3 4.4
```

Sample Output:

```
3
7.7
```

10.2) Develop a C++ Program illustrating template class.

Code:

```
#include <iostream>
using namespace std;
template <typename T>
class Array {
	private:
		T* ptr;
		int size;
	public:
		Array(T arr[], int s)
		{
			ptr = new T[s];
			size = s;
			for (int i = 0; i < size; i++)
				ptr[i] = arr[i];
		}
		void print()
		{
			for (int i = 0; i < size; i++)
				cout <<*(ptr + i)<<" ";
		}
};
```

```
int main()
{
	int n;
	cin>>n;
	int arr[n],i;
	for(i=0;i<n;i++)
	{
		cin>>arr[i];
	}
	Array<int> a(arr, n);
	a.print();
	return 0;
}
```

Output:

```
5
1 2 3 4 5
1 2 3 4 5
```

10.3) Develop a C++ program to illustrate class templates with multiple parameters.

Code:

```
#include<iostream>
using namespace std;
// Class template with two parameters
template<class T1, class T2>
class Test{
		T1 a;
		T2 b;
	public:
		Test(T1 x, T2 y){
			a = x;
			b = y;
		}
		void show(){
			cout << a << " and " << b << endl;
		}
};
int main(){
	// instantiation with float and int type
	Test <float, int> test1 (1.23, 123);
	// instantiation with float and char type
	Test <int, char> test2 (100, 'W');
	test1.show();
	test2.show();
}
```

Output:

```
1.23 and 123
100 and W
```

<u>Experiment-11:</u> EXCEPTIONS

11.1) Develop a C++ program for handling Exceptions.

<u>Code:</u>

```
#include <iostream>
using namespace std;
class Student {
public:
        Student() { cout << "Constructor of Student" << endl; }
        ~Student() { cout << "Destructor of Student " << endl; }
};
int main()
{
        try
        {
                Student t1;
                throw 100;
        }
        catch(int i)
        {
                cout << "Caught " << i << endl;
        }
}
```

<u>Output:</u>

```
Constructor of Student
Destructor of Student
Caught 100
```

11.2) Develop a C++ program to illustrate the use of multiple catch statements.

<u>Code:</u>

```
#include <iostream>
using namespace std;
class Student {
public:
        Student() { cout << "Constructor of Student" << endl; }
        ~Student() { cout << "Destructor of Student " << endl; }
};
int main()
{
        try
        {
                Student t1;
                throw 100;
        }
        catch (float f)
```

```
	{
		cout << "Caught " << f << endl;
	}
	catch (int i)
	{
		cout << "Caught " << i << endl;
	}
	catch (char ch)
	{
		cout << "Caught " << ch << endl;
	}
}
```

Output:

```
Constructor of Student
Destructor of Student
Caught 100
```

<u>Experiment-12:</u> STL

12.1) Develop a C++ program to implement List, Vector and its Operations.

<u>Code:</u>

```
#include<bits/stdc++.h>
using namespace std;
void print(list<vector<int> >& listOfVectors) {
        for (auto vect : listOfVectors) {
                vector<int> currentVector = vect;
                cout << "[ ";
                for (auto element : currentVector)
                        cout << element << " ";
                cout << "]";
                cout << "\n";
        }
}
int main()
{
        // Declaring a list of vectors
        list<vector<int> > listOfVectors;
        // Declaring a vector
        vector<int> vector1;
        vector1.push_back(10);
        vector1.push_back(14);
        vector1.push_back(17);
        // Push the vector at the back in the list
        listOfVectors.push_back(vector1);
        vector<int> vector2;
        vector2.push_back(2);
        vector2.push_back(6);
        vector2.push_back(11);
        // Push the vector at the back in the list
        listOfVectors.push_back(vector2);
        // Declaring another vector
        vector<int> vector3;
        // Adding elements in the vector
        vector3.push_back(11);
        vector3.push_back(16);
        vector3.push_back(29);
        // Push the vector at the front in the list
        listOfVectors.push_front(vector3);
        // Calling print function
        print(listOfVectors);
}
```

<u>Output:</u>

```
[ 11 16 29 ]
[ 10 14 17 ]
[ 2 6 11 ]
```

12.2) Develop a C++ program to implement Deque and Deque Operations.

Code:

```
#include<iostream>
#include<deque>
using namespace std;
void showdq(deque<int> g){
	deque<int>::iterator it;
	for (it = g.begin(); it != g.end(); ++it)
		cout << "\t" << *it;
}
int main()
{
	deque<int> gquiz;
	gquiz.push_back(10);
	gquiz.push_front(20);
	gquiz.push_back(30);
	gquiz.push_front(15);
	cout << "The deque gquiz is : ";
	showdq(gquiz);
	cout << "\ngquiz.size() : " << gquiz.size();
	cout << "\ngquiz.max_size() : " << gquiz.max_size();
	cout << "\ngquiz.at(2) : " << gquiz.at(2);
	cout << "\ngquiz.front() : " << gquiz.front();
	cout << "\ngquiz.back() : " << gquiz.back();
	cout << "\ngquiz.pop_front() : ";
	gquiz.pop_front();
	showdq(gquiz);
	cout << "\ngquiz.pop_back() : ";
	gquiz.pop_back();
	showdq(gquiz);
	return 0;
}
```

Output:

```
The deque gquiz is :    15      20      10      30
gquiz.size() : 4
gquiz.max_size() : 2305843009213693951
gquiz.at(2) : 10
gquiz.front() : 15
gquiz.back() : 30
gquiz.pop_front() :     20      10      30
gquiz.pop_back() :      20      10
```

12.3) Develop a C++ program to implement Map and Map Operations.

Code:

```
#include <iostream>
#include <iterator>
#include <map>
using namespace std;
int main()
```

```
{
        map<int, int> gquiz1;
        // insert elements in random order
        gquiz1.insert(pair<int, int>(1, 40));
        gquiz1.insert(pair<int, int>(2, 30));
        gquiz1.insert(pair<int, int>(3, 60));
        gquiz1.insert(pair<int, int>(4, 20));
        gquiz1.insert(pair<int, int>(5, 50));
        gquiz1.insert(pair<int, int>(6, 50));
        gquiz1[7]=10;  // another way of inserting a value in a map
        // printing map gquiz1
        map<int, int>::iterator itr;
        cout << "\nThe map gquiz1 is : \n";
        cout << "\tKEY\tELEMENT\n";
        for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr) {
                cout << '\t' << itr->first << '\t' << itr->second<< '\n';
        }
        cout << endl;
        // assigning the elements from gquiz1 to gquiz2
        map<int, int> gquiz2(gquiz1.begin(), gquiz1.end());
        // print all elements of the map gquiz2
        cout << "\nThe map gquiz2 after"
                << " assign from gquiz1 is : \n";
        cout << "\tKEY\tELEMENT\n";
        for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {
                cout << '\t' << itr->first << '\t' << itr->second<< '\n';
        }
        cout << endl;
        // remove all elements up to
        // element with key=3 in gquiz2
        cout << "\ngquiz2 after removal of"
                        " elements less than key=3 : \n";
        cout << "\tKEY\tELEMENT\n";
        gquiz2.erase(gquiz2.begin(), gquiz2.find(3));
        for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {
                cout << '\t' << itr->first << '\t' << itr->second<< '\n';
        }
        // remove all elements with key = 4
        int num;
        num = gquiz2.erase(4);
        cout << "\ngquiz2.erase(4) : ";
        cout << num << " removed \n";
        cout << "\tKEY\tELEMENT\n";
        for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {
                cout << '\t' << itr->first << '\t' << itr->second<< '\n';
        }
        cout << endl;
        // lower bound and upper bound for map gquiz1 key = 5
        cout << "gquiz1.lower_bound(5) : "<< "\tKEY = ";
        cout << gquiz1.lower_bound(5)->first << '\t';
        cout << "\tELEMENT = " << gquiz1.lower_bound(5)->second<< endl;
        cout << "gquiz1.upper_bound(5) : "<< "\tKEY = ";
        cout << gquiz1.upper_bound(5)->first << '\t';
        cout << "\tELEMENT = " << gquiz1.upper_bound(5)->second<< endl;
        return 0;
}
```

<u>Output:</u>

```
The map gquiz1 is :
        KEY     ELEMENT
        1       40
        2       30
        3       60
        4       20
        5       50
        6       50
        7       10


The map gquiz2 after assign from gquiz1 is :
        KEY     ELEMENT
        1       40
        2       30
        3       60
        4       20
        5       50
        6       50
        7       10


gquiz2 after removal of elements less than key=3 :
        KEY     ELEMENT
        3       60
        4       20
        5       50
        6       50
        7       10

gquiz2.erase(4) : 1 removed
        KEY     ELEMENT
        3       60
        5       50
        6       50
        7       10

gquiz1.lower_bound(5) :         KEY = 5         ELEMENT = 50
gquiz1.upper_bound(5) :         KEY = 6         ELEMENT = 50
```