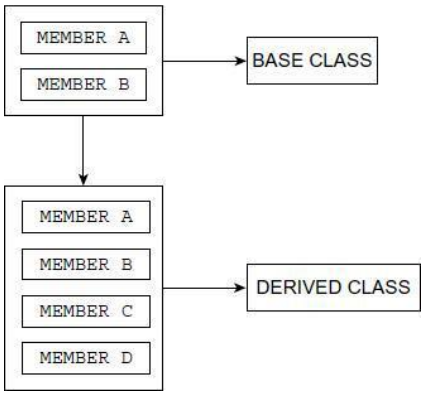
Inheritance

Inheritance is one of the most useful and essential characteristics of object-oriented programming. The existing classes are the main components of inheritance. The new classes are created from existing ones. The properties of the existing classes are simply extended to the new classes. The new classes created by using such a method are known as derived classes, and the existing classes are known as base classes, as shown in the figure. The programmer can define new member variables and functions in the derived class. The base class remains unchanged. The object of the derived class can access members of the base as well as derived classes. On the other hand, the object of the base class cannot access members of the derived classes. The base class does not know about their subclasses.



The base class is also called *super class, parent, or ancestor*, and the derived class is called *subclass, child, or descendent*. It is also possible to derive a class from a previously derived class. A class can be derived from more than one class.

Reusability: Reusability means the reuse of properties of the base class in the derived classes. Reusability is achieved using inheritance. Inheritance and reusability are not different from each other. The outcome of inheritance is reusability.

Inheritance Definition: The procedure of creating a new class from one or more existing classes is termed *inheritance*.

Syntax: A new class can be defined as per the syntax given below. The derived class is indicated by associating with the base class. A new class also has its own set of member variables and functions. The syntax given below creates the derived class.

```
class name_of_the_derived_class: access specifiers name_of_the_base_class
{
        // member variables of new class (derived class)
}
```

The names of the derived and base classes are separated by a colon (:). The access specifiers may be private, public or protected. The keyword private or public is specified followed by a colon In the absence of an access specifier, the default is private. The access specifiers decide whether the characteristics of the base class are derived privately or publicly. The derived class also has its own set of member variables and functions. The following are the possible syntaxes of declaration:

Ex 1:

```
class B: public A
{
        // Members of class B
};
```

In the above syntax, class A is a base class, and class B is a derived class. Here, the class B is derived publicly.

Ex 2:

```
class B: private A // private derivation
{
        // members of class B
};
```

Ex 3:

```
class B: A // by default private derivation
{
        // members of class B
};
```

Ex 4:

```
class B: protected A // same as private
{
        // members of class B
};
```

It is important to note the following points:

- When a public access specifier is used (Ex. 1), the public members of the base class are public members of the derived class. Similarly, the protected members of the base class are protected members of the derived class.
- When a private access specifier is used, the public and protected members of the base class are the private members of the derived class.

Public Inheritance: When a class is derived publicly, all the public members of the base class can be accessed directly in the derived class.

```
// PUBLIC DERIVATION //
class A // BASE CLASS
{
        public:
                int x;
};
class B: public A      // DERIVED CLASS
{
        public:
                int y;
};
int main()
{
        B b;            // DECLARATION OF OBJECT
        b.x=20;
        b.y=30;
        cout<<"\n member of A:"<<b.x;
        cout<<"\n       Member       of
        B:"<<b.y; return 0;
}
```

Output:

```
        Member of A : 20
        Member of B : 30
```

However, in private derivation, an object of the derived class has no permission to directly access even public members of the base class. In such a case, the public members of the base class can be accessed using public member functions of the derived class.

In case the base class has private member variables and a class derived publicly, the derived class can access the member variables of the base class using only member functions of the base class. The public derivation does not allow the derived class to access the private member variable of the class directly as is possible for public member variables. The following example illustrates public inheritance where base class members are declared public and private.

/* **Write a program to derive a class publicly from base class. Declare the base class member under private section.*/**

```
// PUBLIC DERIVATION //
class A // BASE CLASS
{
    private:
            int x;
    public:
            A() {x=20;}
            void showx()
            {
                    cout<<"\n x="<<x;
            }
};
class B : public A    // DERIVED CLASS
{
    public:
            int y;
            B() {y=30;}
            void showy()
            {
                    showx();
                    cout<<"\n y="<<y;
            }
};
int main()
{
            B b; // DECLARATION OF OBJECT
            b.showy();
            return 0;
}
```

Private Inheritance: The objects of the privately derived class cannot access the public members of the base class directly. Hence, the member functions are used to access the members.

/* **Write a program to derive a class privately. Declare the member of base class under public section.**

```
class A    // BASE CLASS
{
        public:
                int x;
};
```

```
class B : private A    // DERIVED CLASS
{
        public:
                int y;
                B()
                {
                        x=20;
                        y=40;
                }
                void show()
                {
                        cout<<"\n x="<<x;
                        cout<<"\n y="<<y;
                }
};
int main()
{
        B b; // DECLARATION OF OBJECT
        b.show();
        return 0;
}
```

/*Write a program to derive a class privately.*/

```
class A    // BASE CLASS
{
        int x;
        public:
                A()
                {
                        x=20;
                }
                void showx()
                {
                        cout<<"\n x="<<x;
                }
};
class B : private A // DERIVED CLASS
{
        public:
        int y;
        B()
        {
                 y=40;
        }
```

```
        void showy()
        {
                showx();
                cout<<"\n y="<<y;
        }
};
int main()
{
                B b; // DECLARATION OF OBJECT
                b.showy();
                return 0;
}
```

Access Specifiers and their Scope:

Sr.No.	Base class access mode	Derived class access mode		
		`Private` derivation	`Public` derivation	`Protected` derivation
A	`public`	`private`	`public`	`protected`
B	`private`	Not inherited	Not inherited	Not inherited
C	`protected`	`private`	`protected`	`protected`

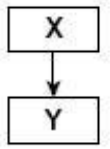
1. All private members of the class are accessible to public members of the same class. They cannot be inherited.
2. The derived class can access the private members of the base class using the member function of the base class.
3. All the protected members of the class are available to its derived classes and can be accessed without the use of the member function of the base class. In other words, we can say that all protected members act as public for the derived class.
4. If any class is prepared for deriving classes, it is advisable to declare all members of the base class as protected, so that derived classes can access the members directly.
5. All the public members of the class are accessible to its derived class. There is no restriction for accessing elements.
6. The access specifier required while deriving class is either private or public. If not specified, private is default for classes and public is default for structures.
7. Constructors and destructors are declared in the public section of the class. If declared in the private section, the object declared will not be initialized and the compiler will flag an error.

Types of Inheritance

Inheritance is classified as follows:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multi-path Inheritance

Single Inheritance: This occurs when only one base class is used for the derivation of a derived class. Further, derived class is not used as a base class, such a type of inheritance that has one base and derived class is known as single inheritance.



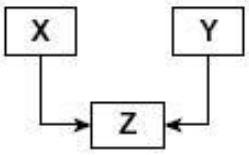
X is a base class. Y is a derived class. This type involves one base and derived class. Further, no class is derived from Y.

Example:

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

class Publisher
{
        string pname;
        string place;
public:
        void getdata()
        {
                cout<<"Enter name and place of publisher:"<<endl;
                cin>>pname>>place;
        }
        void show()
        {
                cout<<"Publisher Name:"<<pname<<endl;
                cout<<"Place:"<<place<<endl;
        }
};
class Book:public Publisher
{
        string title;
        float price;
```

```
        int pages;
public:
        void getdata()
        {
                Publisher::getdata();
                cout<<"Enter Book Title, Price and No. of pages"<<endl;
                cin>>title>>price>>pages;
        }
        void show()
        {
                Publisher:: show ();
                cout<<"Title:"<<title<<endl;
                cout<<"Price:"<<price<<endl;
                cout<<"No. of Pages:"<<pages<<endl;
        }
};
int main() {

        Book b;
        b.getdata();
        b.show();
        return 0;
}
```

Multiple Inheritance: When two or more base classes are used for the derivation of a class, it is called multiple inheritance.



X and Y are base classes. Z is a derived class.
Class Z inherits properties of both X and Y.
Further, Z is not used as a base class.

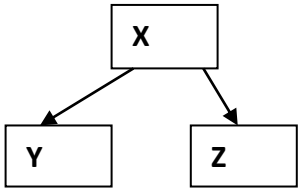
Example:

```
class Publisher
{
        string pname;
        string place;
public:
        void getdata()
        {
                cout<<"Enter name and place of publisher:"<<endl;
                cin>>pname>>place;
        }
        void show ()
```

```
        {
                cout<<"Publisher Name:"<<pname<<endl;
                cout<<"Place:"<<place<<endl;
        }
};
class Author
{
        string aname;
public:
        void getdata()
        {
                cout<<"Enter Author
                name:"<<endl; cin>>aname;
        }
        void show ()
        {
                cout<<"Author Name:"<<aname<<endl;
        }

};
class Book:public Publisher, public Author
{
        string  title;
        float price;
        int pages;
public:
        void getdata()
        {
                Publisher::getdata();
                Author::getdata();
                cout<<"Enter Book Title, Price and No. of pages"<<endl;
                cin>>title>>price>>pages;
        }
        void show()
        {
                Publisher:: show ();
                Author:: show ();
                cout<<"Title:"<<title<<endl;
                cout<<"Price:"<<price<<endl;
                cout<<"No. of Pages:"<<pages<<endl;
        }
};
int main() {

        Book b;
        b.getdata();
```

```
        b.show();
        return 0;
}
```

Hierarchical Inheritance: When a single base class is used for the derivation of two or more classes, it is known as hierarchical inheritance.

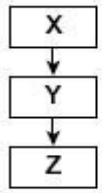


Example:

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

class Account
{
        int act_no; string
        cust_name;
        public:
        void getdata()
        {
                cout<<"Enter Accout number and Customer name:"<<endl;
                cin>>act_no>>cust_name;
        }
        void show ()
        {
                cout<<"Account Number:"<<act_no<<endl;
                cout<<"Customer Name:"<<cust_name<<endl;
        }
};
class SB_Act: public Account
{
        float roi;
public:
        void getdata()
        {
                Account::getdata();
                cout<<"Enter Rate of Interest"<<endl;
                cin>>roi;
        }
```

```
        void show ()
        {
                cout<<"***** SAVINGS ACCOUNT********"<<endl;
                Account:: show ();
                cout<<"Rate of Interest:"<<roi<<endl;
        }

};
class Current_Act: public Account
{
        float roi;
public:
        void getdata()
        {
                Account::getdata();
                cout<<"Enter Rate of Interest"<<endl;
                cin>>roi;
        }
        void show ()
        {
                cout<<"***** CURRENT ACCOUNT********"<<endl;
                Account:: show ();
                cout<<"Rate of Interest:"<<roi<<endl;
        }

};
int main() {
        /* Saving Account*/
        SB_Act s;
        s.getdata();
        s. show ();
        /* Current Account*/
        Current_Act c;
        c.getdata();
        c. show ();
        return 0;
}
```

Multilevel Inheritance: When a class is derived from another derived class, that is, the derived class acts as a base class, such a type of inheritance is known as multilevel inheritance.



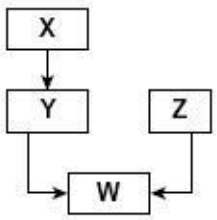
X is a base class. Y is derived from X. Further, Z is derived from Y. Here, Y is not only a derived class but also a base class. Further, Z can be used as a base class.

Example:

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

class Publisher
{
        string pname;
        string place;
public:
        void getdata()
        {
                cout<<"Enter name and place of publisher:"<<endl;
                cin>>pname>>place;
        }
        void show ()
        {
                cout<<"Publisher Name:"<<pname<<endl;
                cout<<"Place:"<<place<<endl;
        }
};
class Author:public Publisher
{
        string aname;
public:
        void getdata()
        {
                Publisher::getdata();
                cout<<"Enter Author
                name:"<<endl; cin>>aname;
        }
        void show ()
        {
                Publisher:: show ();
                cout<<"Author Name:"<<aname<<endl;
        }
```

```
};
class Book:public Author
{
        string  title;
        float price;
        int pages;
public:
        void getdata()
        {
                Author::getdata();
                cout<<"Enter Book Title, Price and No. of pages"<<endl;
                cin>>title>>price>>pages;
        }
        void show()
        {
                Author:: show ();
                cout<<"Title:"<<title<<endl;
                cout<<"Price:"<<price<<endl;
                cout<<"No. of Pages:"<<pages<<endl;
        }
};
int main() {

        Book b;
        b.getdata();
        b.show();
        return 0;
}
```

Hybrid Inheritance: A combination of one or more types of inheritance is known as hybrid inheritance.



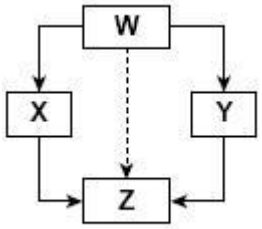
In this type, two types of inheritance is used, i.e. single and multiple inheritance. Class Y is derived from class X. It is single type of inheritance. Further, the derived class Y acts as a base class. The class W is derived from base classes Y and Z. This type of inheritance that uses more than one base class is known as multiple inheritances. Thus, combination of one or more type of inheritance is called as Hybrid inheritance.

Example:

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

class Publisher
{
    string pname;
    string place;
public:
    void getdata()
    {
            cout<<"Enter name and place of publisher:"<<endl;
            cin>>pname>>place;
    }
    void show ()
    {
            cout<<"Publisher Name:"<<pname<<endl;
            cout<<"Place:"<<place<<endl;
    }
};
class Author:public Publisher
{
    string aname;
public:
    void getdata()
    {
            Publisher::getdata();
            cout<<"Enter Author
            name:"<<endl; cin>>aname;
    }
    void show ()
    {
            Publisher:: show ();
            cout<<"Author Name:"<<aname<<endl;
    }

};
class Distributor
{
    string dname;
public:
    void getdata()
    {
            cout<<"Enter Distributor name:"<<endl;
```

```
            cin>>dname;
        }
        void show ()
        {
            cout<<"Distributor Name:"<<dname<<endl;
        }

};
class Book:public Author, public Distributor
{
    string  title;
    float price;
    int pages;
public:
    void getdata()
    {
            Author::getdata();
            Distributor::getdata();
            cout<<"Enter Book Title, Price and No. of pages"<<endl;
            cin>>title>>price>>pages;
    }
    void show()
    {
            Author:: show (); Distributor::
            show ();
            cout<<"Title:"<<title<<endl;
            cout<<"Price:"<<price<<endl;
            cout<<"No. of Pages:"<<pages<<endl;
    }
};
int main() {

    Book b;
    b.getdata();
    b.show();
    return 0;
}
```

Multipath Inheritance: When a class is derived from two or more classes, those are derived from the same base class. Such a type of inheritance is known as multipath inheritance. The multipath inheritance also consists of many types of inheritance, such as multiple, multilevel, and hierarchical, as shown in the figure.



But the disadvantage is ambiguity in classes. Consider the following example:

```
class A1
{
        protected: int
                a1;
};
class A2 : public A1
{
        protected: int
                a2;
};
class A3: public A1
{
        protected: int
                a3;
};
class A4: public A2,A3
{
        int a4;
};
```

In the above example, classes `A2` and `A3` are derived from class `A1`; that is, their base class is similar to class A1 (hierarchical inheritance). Both classes A2 and A3 can access the variable a1 of class A1. The class A4 is derived from classes A2 and A3 by multiple inheritance. If we try to access the variable a1 of class A1, the compiler shows error

In the above example, we can observe all types of inheritance, that is, multiple, multilevel, and hierarchical. The derived class A4 has two sets of data members of class A1 through the middle base classes A2 and A3. The class A1 is inherited twice.

Virtual Base Classes: To overcome the ambiguity occurring due to multipath inheritance, the C++ provides the keyword `virtual`. The keyword `virtual` declares the specified classes virtual. The example given below illustrates the virtual classes:

```
 class Publisher
{
        string pname;
        string place;
public:
        void getdata()
        {
                cout<<"Enter name and place of publisher:"<<endl;
                cin>>pname>>place;
        }
        void show ()
        {
                cout<<"Publisher Name:"<<pname<<endl;
                cout<<"Place:"<<place<<endl;
        }
};
class Author:virtual public Publisher
{
        string aname;
public:
        void getdata()
        {
                cout<<"Enter Author
                name:"<<endl; cin>>aname;
        }
        void show ()
        {
                cout<<"Author Name:"<<aname<<endl;
        }

};
class Distributor:virtual public Publisher
{
        string dname;
public:
        void getdata()
        {
                cout<<"Enter Distributor name:"<<endl;
                cin>>dname;
        }
```

```
        void show ()
        {
                cout<<"Distributor Name:"<<dname<<endl;
        }

};

class Book:public Author, public Distributor
{
        string  title;
        float price;
        int pages;
public:
        void getdata()
        {
                Publisher::getdata();
                Author::getdata();
                Distributor::getdata();
                cout<<"Enter Book Title, Price and No. of pages"<<endl;
                cin>>title>>price>>pages;
        }
        void show()
        {
                Publisher:: show (); Author::
                show (); Distributor:: show ();
                cout<<"Title:"<<title<<endl;

                cout<<"Price:"<<price<<endl;
                cout<<"No. of Pages:"<<pages<<endl;
        }
};
int main() {

        Book b;
        b.getdata();
        b.show();
        return 0;
}
```

Pointers

A pointer is a memory variable that stores a memory address. Pointers can have any name, and it is declared in the same fashion as other variables, but it is always denoted by '*' operator.

Features of Pointers

2. Pointers save memory space.
3. Execution time with pointers is faster, because data are manipulated with the address, that is, direct access to memory location.
4. Memory is accessed efficiently with the pointers. The pointer assigns as well as releases the memory space. Memory is dynamically allocated.
5. Pointers are used with data structures. They are useful for representing two-dimensional and multi-dimensional arrays.
6. We can access the elements of any type of array, irrespective of its subscript range.
7. Pointers are used for file handling.
8. Pointers are used to allocate memory in a dynamic manner.
9. In C++, a pointer declared to a base class could access the object of a derived class. However, a pointer to a derived class cannot access the object of a base class.

Pointer Declaration

Pointer variables can be declared as follows:

Example:

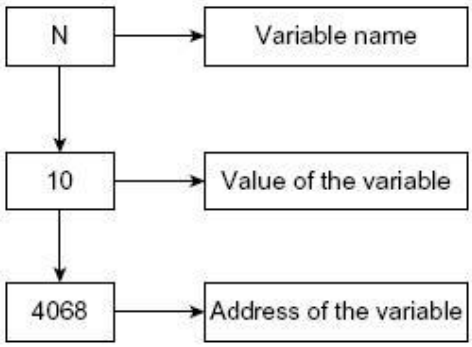
```
int *x;
float *f;
char *y;
```

In the first statement, 'x' is an integer pointer, and it informs the compiler that it holds the address of any integer variable. In the same way, 'f' is a float pointer that stores the address of any float variable, and 'y' is a character pointer which stores the address of any character variable.

The indirection operator (*) is also called the *dereference operator*. When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.

/* Write a program to display the address of the variable.*/

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

int main()
{
        int n;

        cout<<"Enter a Number = ";
        cin>>n;
        cout<<"Value of n = "<<n;
        cout<<"Address of n= " <<(unsigned)&n;
        return 0;
}
```



Output:

```
Enter a Number = 10
Value of n = 10
Address of n=4068
```

/* Write a program to declare a pointer. Display the value and address of the variable-using pointer. */

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

int main()
{
    int *p;
    int x=10;

    p=&x;

    cout<<"\n x="<<x <<" &x="<<&x;
    cout<<"\n x="<<*p<<" &x="<<p;

    return 0;
}
```

Output:

x=10 &x=0x22ff18
x=10 &x=0x22ff18

Arithmetic Operations with Pointers

We can perform different arithmetic operations by using pointers. Increment, decrement, prefix, and postfix operations can be performed with pointers. The effects of these operations are shown in Table.

Data Type	Initial Address	Operation		Address After Operations		Required Bytes
int i = 2	4046	++	--	4048	4044	2
char c = 'x'	4053	++	--	4054	4052	1
float f = 2.2	4058	++	--	4062	4054	4
long l = 2	4060	++	--	4064	4056	4

From the above table, while referring to the first entry, we can observe that on increment of the pointer variable for integers, the address is incremented by two; that is, 4046 is the original address and on increment, its value will be 4048, because integers require two bytes. Similarly, when the pointer variable for integer is decreased, its address 4048 becomes 4046.

/* Program on pointer incrementation and decrementation.*/

```
int main()
{
        int x=10;
        int *p;
        p=&x;
        cout<<"\n Address of p:"<<unsigned(p);
        p=p+4;
        cout<<"\n Address of p:"<<unsigned(p);
        p=p-2;
        cout<<"\n Address of p:"<<unsigned(p);
        return 0;
}
```

Output:

Address of p:65524
Address of p:65532
Address of p:65528

/* Program on changing the values of variables using pointer.*/

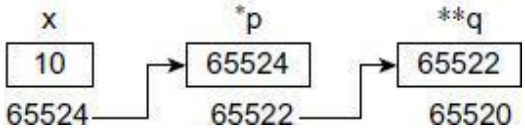
```
#include <iostream>
using namespace std;

int main() {
        int x=10;
        int *p;
        p=&x;
        cout<<"\n Value of x:"<<*p;
        *p=*p+10;
        cout<<"\n Value of x:"<<*p;
        *p=*p-2;
        cout<<"\n Value of x:"<<*p;
        return 0;
}
```

Output:

```
Value of x: 10
Value of x: 20
Value of x: 18
```

Pointer to Pointer

Pointer to pointer is a pointer that stores the address of another pointer. There can be a chain of pointers depending on applications/requirements. In the Figure,, x is a simple variable, p is a pointer to the variable x, and q is a pointer to p. The values of variables 'x,* p, and **q' are shown in the boxes, and their addresses are shown outside the boxes.



/*Program to demonstrate the concept of pointer to pointer.*/

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

int main() {
        int x=10;
        int *p;
        int **q;
```

```
        int ***z;
        p=&x;
        q=&p;
        z=&q;
        cout<<"\n Value of x = "<<x;
        cout<<"\n Value of x = "<<*p;
        cout<<"\n Value of x = "<<**q;
        cout<<"\n Value of x = "<<***z;
        cout<<"\n Adderss of x "<<unsigned(&x);
        cout<<"\n Adderss of p "<<unsigned(&p);
        cout<<"\n Adderss of q "<<unsigned(&q);

        return 0;

}
```

OUTPUT

```
Value of x = 10
 Value of x = 10
 Value of x = 10
 Value of x = 10

 Adderss of x 2293528
 Adderss of p 2293524
 Adderss of q 2293520
```

void Pointers

When a variable is declared as being a pointer to type **void** it is known as a *generic pointer*. Since you cannot have a variable of type **void**, the pointer will not point to any data and therefore cannot be dereferenced. It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term *Generic pointer*. This is very useful when you want a pointer to point to data of different types at different times.

/* Program to illustrate the use of void pointer */

```
int main()
{
        int i;
        char c;
        void *the_data;
```

```
        i = 6;
        c = 'a';

        the_data = &i;
        cout<<"The data points to the integer value :"<< *(int*) the_data;

        the_data = &c;
        cout<<"\nThe data now points to the character:"<< *(char*) the_data;

        return 0;
}
```

Output:

```
The data points to the integer value : 6
The data now points to the character: a
```

wild Pointers or Dangling Pointers

Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location and may cause a program to crash or behave badly.

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

int main()
{
        int *p; /* wild pointer */
        *p = 12; /* Some unknown memory location is being corrupted. This
                    should never be done. */
        cout<<unsigned(p);

}
```

Please note that if a pointer p points to a known variable then it's not a wild pointer. In the below program, p is a wild pointer till this points to a.

```
int main()
{
        int *p; /* wild pointer */
        int a = 10;
        p = &a; /* p is not a wild pointer now*/
        *p = 12; /* This is fine. Value of a is changed */
}
```

The pointer becomes a `wild` pointer due to the following reasons:

- Pointer declared but not initialized
- Pointer alteration
- Accessing destroyed data

this Pointer

When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object (i.e., the object on which the function is invoked). This pointer is known as *this pointer*. It is internally created at the time of function call.

/* Program to print the address of the object using this pointer*/

```
#include  <iostream>
using namespace std;
class integer
{
        int x;
public:
        void show_addr();
};
void integer::show_addr()
{
        cout<<"My Object's Address="<<this<<"\n";
}
int main() {
        integer a,b,c;
        cout<<"A  addr:  "<<&a<<endl;
        cout<<"B  addr:  "<<&b<<endl;
        cout<<"C  addr:  "<<&c<<endl;
        a.show_addr();
        b.show_addr();
        c.show_addr();
        return 0;

}
```

Output:

1. addr:0x22ff1c
2. addr: 0x22ff18
3. addr: 0x22ff14

```
My Object's Address=0x22ff1c
My Object's Address=0x22ff18
My Object's Address=0x22ff14
```

/* Program to add two object contents using this pointer*/

```
#include  <iostream>
using namespace std;
class Add
{
        int val;
public:
        void setdata(int val)
        {
                this->val=val;
        }
        void display()
        {
                cout<<val<<endl;
        }
        Add sum(Add v2)
        {
                val=val+v2.val;
                return *this;
        }
};
int main() {
        Add v1,v2;
        v1.setdata(3);
        v2.setdata(4);

        Add s;
        s=v1.sum(v2);
        cout<<"Sum is=";
        s.display(); return
        0;

}
```

Output:

```
Sum is=7
```

Pointer to Derived Classes and Base Class

It is possible to declare a pointer that points to the base class as well as the derived class. One pointer can point to different classes. For example, X is a base class and Y is a derived class. The pointer pointing to X can also point to Y.

/* Program to declare a pointer to the base class and access the member variable of base calss.*/

```
class B
{
      public :
      int b;
      void display()
      {
      cout<<"b = " <<b <<endl;
      }
};
class D : public B
{
          public :
                    int d;
                    void display()
                    {
                              cout<<"b= " <<b <<"\n" <<" d="<<d <<endl;
                    }
  };
int main()
{
             B *cp; B
             base;
             cp=&base;
             cp->b=100;
             // cp->d=200; Not Accessible
             cout<<"\n cp points to the base object \n";
             cp->display();
             D d;
             cout<<"\n cp points to the derived class \n";
             cp=&d;
             cp->b=150;
              //cp->d=300; Not accessible
             cp->display();
             return 0;

}
```

Output:

```
 cp points to the base object
b = 100
cp points to the derived class
b = 150
```

/* Program to declare a pointer to the derived class and access the member variable of base and derived class.*/

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

class B
{
    public :
    int b;
    void display()
    {
        cout<<"b = "<<b <<endl;
    }
};
class D : public B
{
        public:
                int d;
                void display()
                {
                        cout<<"b= "<<b <<endl;
                        cout<<"d= "<<d <<endl;
                }
};
int main()
{
        D *cp;
        D d;
        cp=&d;
        cp->b=100;
        cp->d=350;
        cout<<"\n cp points to the derived object \n";
        cp->display();
        return 0;
}
```

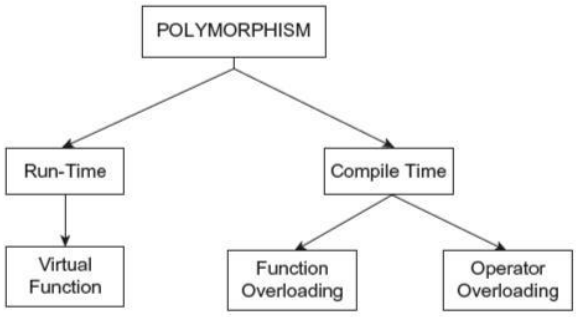
Output:

```
cp points to the derived object
b= 100
d= 350
```

Polymorphism

The word *poly* means many, and *morphism* means several forms. Both the words are derived from Greek language. Thus, by combining these two words, a new whole word called polymorphism is created, which means various forms.

In C++, the function can be bound at either compile time or run time. Deciding a function call at compile time is called compile time or early or static binding. Deciding a function call at run time is called run time or late or dynamic binding. Dynamic binding permits to suspend the decision of choosing a suitable member function until run time. Two types of polymorphism are shown in Figure.



A polymorphism is a technique in which various forms of a single function can be defined and shared by various objects to perform an operation.

Binding in C++

Binding refers to the process that is to be used for converting functions and variables into machine language addresses. The C++ supports two types of binding: static or early binding and dynamic or late binding.

Static (Early) Binding: By default, matching of function call with the correct function definition happens at compile time. This is called static binding or early binding or compile-time binding. Static binding is achieved using function overloading and operator overloading. Even though there are two or more functions with same name, compiler uniquely identifies each function depending on the parameters passed to those functions. Consider the following example.

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

class Base
{
        public:
        void display()
        {
                cout<<"Base"<<endl;
        }
};

class Derived:public Base
{
        public:
        void display()
        {
                cout<<"Derived"<<endl;
        }
};
int main()
{
        Derived d;
        d.Base::display(); // Invokes base class function
        d.display(); // Invokes derived class function
        return 0;
}
```

Output:

```
Base
Derived
```

Explanation: In the above program both the classes contain display() member function. Both the classes contain a similar function name. In function `main()` Hence, in order to invoke the `display()` function of the base class, the scope access operator is used. When base and derived classes have similar function names, in such a situation, it is very essential to provide information to the compiler at compile time about the member functions.

Dynamic (Late) Binding: C++ provides facility to specify that the compiler should match function calls with the correct definition at the run time; this is called dynamic binding or late binding or run-time binding. Dynamic binding is achieved using virtual functions, base class pointer points to derived class object and Inheritance.

Pointer to Base and Derived Class Objects:

In inheritance, the properties of existing classes are extended to the new classes. The new classes that can be created from the existing base class are called as derived classes. The inheritance provides the hierarchical organization of classes. It also provides the hierarchical relationship between two objects and indicates the shared properties between them. All derived classes inherit properties from the common base class. Pointers can be declared to the point base or derived class. Pointers to objects of the base class are type compatible with pointers to objects of the derived class. A base class pointer can point to objects of both the base and derived class. In other words, a pointer to the object of the base class can point to the object of the derived class; whereas a pointer to the object of the derived class cannot point to the object of the base class.

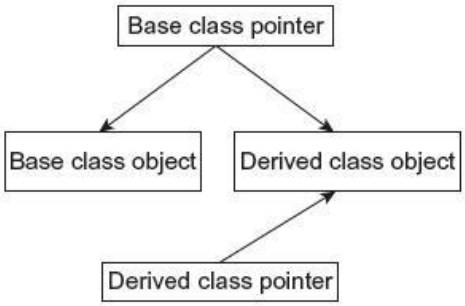


Fig. Type compatibility of base and derived class pointers

Virtual Functions:

A virtual function is a member function that is declared with in a base class and is redefined by derived class.

To create a virtual function, precede the functions declaration in the base class with the keworh '**virtual**'. It signals the compiler that we don't want static linkage for this function. So,

when a class containing virtual function is inherited, the derived class redefines the virtual function to fit its own needs. i.e., the definition creates a specific method.

Rules for Virtual Functions:

1. The virtual function should not be static and must be a member of a class.
2. The virtual function may be declared as a friend for another class. An object pointer can access the virtual functions.
3. A constructor cannot be declared as virtual, but a destructor can be declared as virtual.
4. The virtual function should be defined in the public section of the class. It is also possible to define the virtual function outside the class. In such a case, the declaration is done inside the class, and the definition is outside the class. The virtual keyword is used in the declaration and not in the function declaration.
5. It is also possible to return a value from virtual functions similar to other functions.
6. The prototype of the virtual function in the base class and derived class should be exactly the same. In case of a mismatch, the compiler neglects the virtual function mechanism and treats these functions as overloaded functions.
7. Arithmetic operations cannot be used with base class pointers.
8. If a base class contains a virtual function and if the same function is not redefined in the derived classes, in such a case, the base class function is invoked.

Example:

```
 class Base
{
        public:
        virtual void display()
        {
                        cout<<"Base"<<endl;
        }
};

class Derived:public Base
{
        public:
        void display()
```

```
        {
                cout<<"Derived"<<endl;
        }
};
int main()
{
        Derived b;
        Base *a=&b;
        a->display();
        return 0;

}
```

Output:

Derived

/* C++ program to use pointer for both base and derived class and call the member function. Use virtual keyword. */

```
#include <iostream>
using namespace std;
class super
{
        public:
        virtual void display()
        {
                cout<<"\n In function display() - class super";
        }
        virtual void show()
        {
                cout<<"\nIn function show() - class super";
        }
}
;
class sub: public super
{
        public:
        void display()
        {
                cout<<"\nIn function display() class sub";
        }
        void show()
        {
```

```
            cout<<"\nIn function show() class sub";
        }
};
int main()
{
        super sup;
        sub s;
        super *sp;
        cout<<"\n Super Pointer points to class super\n";
        sp=&sup;
        sp->display();
        sp->show();

        cout<<"\n\n Super Pointer points to derived class sub\n";
        sp=&s;
        sp->display();
        sp->show();
        return 0;

}
```

Output:

```
Super Pointer points to class super
        In function display() - class super
        In function show() - class super

Super Pointer points to derived class sub
        In function display() class sub
        In function show() class sub
```

/*C++ program to create array of pointers. Invoke functions using array objects.*/

```
        #include <iostream>
        using namespace std;
        class A
        {
                public:
                virtual void show() {           cout<<"A\n"; }
        };
        class B : public A
        {
                public:
```

```
                void show() {cout<<"B\n";}
};
class C : public A
{
        public:
        void show()    {       cout<<"C\n"; }
};
class D : public A
{
        public:
        void show()    {       cout<<"D\n"; }
};
class E : public A
{
        public:
        void show()    {       cout<<"E";     }
};
int main()
{
        A a;
        B b;
        C c;
        D d;
        E e;
        A *pa[]={&a,&b,&c,&d,&e};
        for ( int j=0;j<5;j++)
                pa[j]->show();
        return 0;

}
```

Output:

```
A
B
C
D
E
```

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. Declaration of pure virtual function

virtual void display() =0; // pure function

In the above declaration of the function, the `display()` is a pure virtual function. The assignment operator is not used to assign zero to this function. It is used just to instruct the compiler that the function is a pure virtual function and that it will not have a definition.

A pure virtual function declared in the base class cannot be used for any operation. The class containing the pure virtual function cannot be used to declare objects. Such classes are known as **abstract classes** or **pure abstract classes.**

/* C++ program to declare pure virtual functions.*/

```
#include <iostream>
using namespace std;
class Base
{
        public:
        virtual void display()=0;
};

class Derived:public Base
{
        public:
        void display()
        {
                        cout<<"Derived"<<endl;
        }
};
int main()
{
        Derived b;
        Base *a=&b;
        a->display();
        return 0;
}
```

Output:

Derived

Explanation: In the above program, the **display()** function of the base class is declared a pure function. The pointer object *a holds the address of the object of the derived class and invokes the function display() of the derived class. Here, the function display() of the base class does nothing. If we try to invoke the pure function using the statement b->Base::display(), the program is terminated with the error "abnormal program termination."

Abstract Classes

Abstract Class is a class which contains at least one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

While defining such an abstract class, the following points should be kept in mind:

3. Do not declare an object of abstract class type.
4. An abstract class can be used as a base class.
5. The derived class should not have pure virtual functions. Objects of the derived class can be declared.

Example:

```
class Base          //Abstract base class
{
        public:
         virtual void show() = 0;          //Pure Virtual Function
};

class Derived:public Base
{
         public:
         void show() {
                cout << "Implementation of Virtual Function in Derived class";
         }
};

int main()
{
 //Base obj; //Compile Time Error
 Base *b;
 Derived d;
```

```
b = &d; b-
>show();
}
```

Virtual functions in derived classes:

In C++, once a member function is declared as a virtual function in a base class, it becomes virtual in every class derived from that base class. In other words, it is not necessary to use the keyword virtual in the derived class while declaring redefined versions of the virtual base class function.

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

class Base          //Abstract base class
{
        public:
                virtual void show()
                {
                        cout << "Implementation of Virtual Function in Derived class";
                }
};

class Derived:public Base
{
};

int main()
{
        Base *b;
        Derived d;
        b = &d; b-
        >show();
}
```

Output:

Implementation of Virtual Function in Derived class

Run-Time Polymorphism Example:

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

class CPolygon
{
        protected:
                int width, height;
        public:
                void get (int first, int second)
                {
                        width= first;
                        height= second;
                }
                virtual int area()=0;

};
class CRectangle: public CPolygon
{
        public:
                int area()
                {
                        return (width * height);
                }
};

class CTriangle: public CPolygon
{
        public:
                int area()
                {
                        return (width * height / 2);
                }
};
int main ()
{
        CRectangle rectangle;
        CTriangle triangle;
        CPolygon * ptr_polygon;

        ptr_polygon = &rectangle;
        ptr_polygon->get(2,2);
        cout << ptr_polygon->area () << endl;
```

```
        ptr_polygon = &triangle;
        ptr_polygon->get(2,2);
        cout << ptr_polygon->area () << endl;

        return 0;
}
```

Output:

```
4
2
```

Object Slicing: Virtual functions can be invoked using a pointer or reference. If we do so, object slicing takes place. The following program takes you to the real thing:

```
class Base
{
        public:
        virtual void display()=0;
};

class Derived:public Base
{
        public:
        void display()
        {
                        cout<<"Derived"<<endl;
        }
};
int main()
{
        Derived b;
        b. display();//object reference
        Base *a=&b;
        a->display(); //pointer
        return 0;

}
```

Output:

```
Derived
Derived
```