Polymorphism

- Polymorphism is composed of two greek words where poly means many and morph means form.
- Polymorphism is defined as one of the important features of oop which means one name multiple forms.

Types of polymorphism

1. **Compile time polymorphism**

- It is defined as the polymorphism where the object is bound to its function call at the compile time.
- It is also known as early binding/static binding/static linking.
- It can be achieved either by function overloading or by operator overloading.

2. **Run time polymorphism**

- It is defined as the polymorphism where,during the run time it is known that which class object are under consideration and according to that the appropriate version of the function is invoked.
- It is also known as the dynamic binding/late binding.
- It can be achieved either by using pointer,virtual function or by function overriding.

Operator overloading

- It is defined as the mechanism of using same operator to do multiple operations like ' + 'do addition of int,float and can also be used to concatenate(i.e. add) two strings.

- Using overloading features we can add two user defined data types such as objects ,with same syntax just as the basic data type i.e. a+b; where ,a and b are the normal int type variables. t1+t2;where, t1 and t2 are objects of class time.

- Operator overloading enhances exhaustability i.e. implementation of operator for multiple purposes.

General form/syntax for Defining operator function

1.If the operator function definition is inside the class:

```
return-type operator op(argument-list){
//body of operator function
}
```

2.If the operator function definition is outside the class:

```
return-type class-name:: operator op(argument-list){
//body of operator function
}
```

Here**,<u>op</u>** is the operator being overloaded and <u>operator op</u> is the function name ,where <u>operator</u> is the keyword.
Operator overloading is done with the help of the special function called <u>operator function</u>.
Operator function might be either the member function or the friend function.
Overloading operator must have at least one operand that is of user-defined type.

Note:

- Friend function will have only one argument for unary operator and two arguments for binary operator.
- But a member function has no arguments for unary operators and only one argument for the binary operator.This is because the object used to call the member function is passed implicitly and therefore is available for the member function ,but this is not the case for the friend function.

The process of operator overloading involves the following steps:

- Create a class that defines the data type that is to be used in the overloading operation.
- Declare the operator op() in the public section of the class.
- Define the operator function to implement the required operations.

Overloading Unary Operator

->>Overloading unary – operator

1.Using operator function as a member function without return type.

```
#include<iostream>
using namespace std;
class demo{
	int x,y,z;
	public:
		void get(int a,int b,int c){
			x=a;
			y=b;
			z=c;
```

```
                }
                void display(){
              cout<<"x="<<x<<endl<<"y="<<y<<endl<<"z="<<z<<endl;
                }
                void operator -(){
                        x=-x;
                        y=-y;
                        z=-z;
                }
};
main(){
        demo d;
        d.get(1,2,3);
        d.display();
        -d;//d.operator -()
        d.display();
}
```

2.Using operator function as a member function with return type.

```
#include<iostream>
using namespace std;
class demo{
        int x, y,z;
        public:
                void get(int a,int b,int c){
                        x=a;
                        y=b;
                        z=c;
                }
                void display(){
```

```
        cout<<"x="<<x<<endl<<"y="<<y<<endl<<"z="<<z<<endl;
                }
                demo operator -(){
                        demo d2;
                        d2.x=-x;
                        d2.y=-y;
                         d2.z=-z;
                return d2;
                }
};
main(){
        demo d,d3;
        d.get(1,2,3);
        d.display();
        d3=-d;//d.operator -()
        d3.display();

}
```

3.Using operator function as a friend function without return type.

```
#include<iostream>
using namespace std;
class demo{
        int x,y,z;
        public:
                void get(int a,int b,int c){
                        x=a;
                        y=b;
```

```
                    z=c;
                }
                void display(){
               cout<<"x="<<x<<endl<<"y="<<y<<endl<<"z="<<z<<endl;
                }
                friend void operator -(demo &d);
        };
                void operator -(demo &d){
                    d.x=-d.x;
                    d.y=-d.y;
                    d.z=-d.z;
                }

    main(){
        demo d;
        d.get(1,2,3);
        d.display();
        -d;//operator -(d)
        d.display();
    }
```

4.Using operator function as a friend function with return type.

```
#include<iostream>

using namespace std;

class demo{

    int x,y,z;

    public:

        void get(int a,int b,int c){
```

```
                    x=a;
                    y=b;
                    z=c;
          }
          void display(){
         cout<<"x="<<x<<endl<<"y="<<y<<endl<<"z="<<z<<endl;
          }
          friend demo operator -(demo d);
     };
          demo operator -(demo d){
                    demo d2;
                    d2.x=-d.x;
                    d2.y=-d.y;
                    d2.z=-d.z;
                    return d2;
          }
main(){
     demo d,d3;
     d.get(1,2,3);
     d.display();
     d3=-d;//operator -(d)
     d3.display();
```

```
}
```

****Overloading prefix and post fix ++ unary operator****

1.Overloading prefix ++ operator:

```
#include<iostream>
using namespace std;
class demo{
        int a,b;
        public:
                void get(int x,int y){
                        a=x;
                        b=y;
                }
                void operator ++(){
                        ++a;
                        ++b;
                }
                void display(){
                        cout<<"a="<<a<<"b="<<b;
                }
};
main(){
        demo d;
        d.get(1,2);

        d.display();

        ++d;//d.operator ++()

        d.display();

}
```

2.Overloading postfix ++ operator:

```
        #include<iostream>
        using namespace std;
        class demo{
                int a,b;
                public:
                demo(int x,int y){
                        a=x;
                        b=y;
                }
                void operator ++ (int){
                        a++;
                        b++;
                }
                void display(){
                        cout<<"a="<<a<<"b="<<b;
                }
        };
        main(){

                demo d(1,2);
                d.display();
                d++;//d.operator ++()
        d.display();

    }
```

Here,in case of postfix ++ ,c++ uses a dummy variable,dummy argument which is a fake integer parameter that only serves to distinguish the postfix version from the prefix version.

Overloading Binary Operator

1.Overloading binary + operator:

Previously we have done

c4=c3.addcomplex(c1,c2)

Now, by overloading binary operator it can be done as:

c4=c1+c2.

Eg:

Adding two complex number object by overloading binary + operator by:

a)using operator function as a member function with no return type.

```
#include<iostream>

using namespace std;

class complex{

    int real,imag;

    public:

        complex(){

            real=0;
```

```
                imag=0;
        }
        complex(int r,int i){
                real=r;
                imag=i;
        }
        void display(){
                cout<<"real="<<real<<"imag="<<imag;
        }
        void operator +(complex c){
                real=real+c.real;
                imag=imag+c.imag;
        }
};
main(){
    complex c1(1,2);
    complex c2(3,4);
    c1+c2;//c1.operator +(c2)
    c1.display();
}
```

b)using operator function as a member function with return type

```
#include<iostream>
```

```
using namespace std;
class complex{
        int real,imag;
        public:
                complex(){
                        real=0;
                        imag=0;
                }
                complex(int r,int i){
                        real=r;
                        imag=i;
                }
                void display(){
                        cout<<"real="<<real<<"imag="<<imag;
                }
                complex operator +(complex c){
                        complex c4;
                        c4.real=real+c.real;
                        c4.imag=imag+c.imag;
                        return c4;
                }
};
```

```
main(){
        complex c1(1,2);
        c1.display();
        complex c2(3,4);
        c2.display();
        complex c3;
        c3=c1+c2;//c1.operator +(c2)
        c3.display();
}
```

c)using operator function as a friend function with return type

```
#include<iostream>
using namespace std;
class complex{
        int real,imag;
        public:
                complex(){
                        real=0;
                        imag=0;
                }
                complex(int r,int i){
                        real=r;
                        imag=i;
                }
                void display(){
                        cout<<"real="<<real<<"imag="<<imag;
                }
                friend complex operator +(complex c1,complex c2);
        };
                complex operator +(complex c1,complex c2){
                        complex c;
                        c.real=c1.real+c2.real;
                        c.imag=c1.imag+c2.imag;
                        return c;
                }
main(){
        complex c1(1,2);
        c1.display();
        complex c2(3,4);
         c2.display();
        complex c3;
        c3=c1+c2;//operator +(c1,c2)
        c3.display();
}
```

d)using operator function as a friend function with no return type.

```
#include<iostream>
using namespace std;
class complex{
	int real,imag;
	public:
		complex(){
			real=0;
			imag=0;
		}
		complex(int r,int i){
			real=r;
			imag=i;
		}
		void display(){
			cout<<"real="<<real<<"imag="<<imag<<endl;
		}
		friend void operator +(complex &c1,complex &c2);
	};
		void operator +(complex  &c1,complex  &c2){
			c1.real=c1.real+c2.real;
			c1.imag=c1.imag+c2.imag;
```

```
        }
main(){
    complex c1(1,2);
    c1.display();
    complex c2(3,4);
    c2.display();
    complex c3;
    c1+c2;//operator +(c1,c2)
    c1.display();
}
```

Some practice questions:

1)WAP to overload binary operator – so that the statement c3=c1-c2 holds true.

```
#include<iostream>
using namespace std;
class complex{
    int real,imag;
    public:
        complex(){
            real=0;
            imag=0;
        }
        complex(int r,int i){
            real=r;
            imag=i;
        }
        void display(){
            cout<<"real="<<real<<"imag="<<imag;
        }
        complex operator -(complex c){
            complex c4;
            c4.real=real-c.real;
            c4.imag=imag-c.imag;
            return c4;
        }
};
main(){
    complex c1(1,2);
    c1.display();
    complex c2(3,4);
    c2.display();
    complex c3;
    c3=c1-c2;//c1.operator +(c2)
    c3.display();
```

```
}
```

2)WAP to overload binary + operator to add two times in hr and min ,where operator function is a friend function with return type.

```
#include<iostream>
using namespace std;
class time{
        int hr,min;
        public:
                time(){
                        hr=0;
                        min=0;
                }
                time(int h,int m){
                        hr=h;
                        min=m;
                }
                void display(){
                        cout<<"hour="<<hr<<"minutes="<<min<<endl;
                }
                friend time operator +(time t1,time t2);
};
time operator +(time t1,time t2){
        time t;
        t.min=t1.min+t2.min;
        t.hr=t.min/60;
        t.min=t.min%60;
        t.hr=t.hr+t1.hr+t2.hr;
        return t;
}
main(){
        time t1(20,30);
        t1.display();
        time t2(10,40);
        t2.display();
        time t3;
        t3=t1+t2;
        t3.display();
}
```

3)WAP to overload binary + operator to add two height in feet and inch.

```
#include<iostream>
using namespace std;
class height{
        int ft,inch;
        public:
```

```
		height(){
			ft=0;
			inch=0;
		}
		height(int f,int i){
			ft=f;
			inch=i;
		}
		void display(){
			cout<<"feet="<<ft<<"inch="<<inch<<endl;
		}
		height operator +(height h2){
			height h;
			h.inch=inch+h2.inch;
			h.ft=h.inch/12+ft+h2.ft;
			h.inch=h.inch%12;
			return h;
		}
};
main(){
	height h1(2,11);
	h1.display();
```

```
        height h2(5,11);
        h2.display();
        height h3;
        h3=h1+h2;
        h3.display();
}
```

4)WAP to overload += operator to add height of two objects and put the final sum value in first object.

```
#include<iostream>
using namespace std;
class height{
        int ft,inch;
        public:
                height(){
                        ft=0;
                        inch=0;
                }
                height(int f,int i){
                        ft=f;
                        inch=i;
                }
                void display(){
```

```
                cout<<"feet="<<ft<<"inch="<<inch<<endl;
        }
        void operator +(height h2){
                inch=inch+h2.inch;
                ft=inch/12+ft+h2.ft;
                inch=inch%12;

        }
};
main(){
    height h1(2,11);
    h1.display();
    height h2(5,11);
    h2.display();
    h1+h2;
    h1.display();
}
```

5)WAP to generate Fibonacci series using ++ operator overloading.

```
#include<iostream>
using namespace std;
class fibo{
    int a,b,c;
```

```
        public:
                fibo(){
                        a=0;
                        b=1;
                        c=a+b;
                }
                void operator ++(){

                        a=b;
                        b=c;
                        c=a+b;
                }
                void display(){
                        cout<<c<<" ";
                }
};
main(){
        fibo f;
        int i,n;
        cout<<"enter total no of terms:";
        cin>>n;
        cout<<"0"<<" "<<"1"<<" ";
        for(i=0;i<n-2;i++){
                f.display();
                ++f;
        }
}
```

Run time polymorphism

Pointer

->><u>For creating an object pointer</u>

Syntax:

class-name *pointer-variable=new class-name;

->><u>For creating array of object using pointers</u>

Syntax:

class-name *pointer-variable=new class-name[size];

Eg of object pointer(or pointer to an object)

```
#include<iostream>
using namespace std;
class item{
        int code;
        float price;
        public:
                void get(int c,float p){
                    code=c;
                    price=p;
            }
            void display(){
                    cout<<"code="<<code<<"price="<<price;
            }

};
main(){
        item *ptr=new item;
        ptr->get(5,5.3);
        ptr->display();
}
```

Eg for creating array of object using pointer

```
#include<iostream>
using namespace std;
class item{
        int code;
        float price;
        public:
            void get(int c,float p){
                    code=c;
```

```
                        price=p;
                }
                void display(){
                        cout<<"code="<<code<<"price="<<price;
                }

};
main(){
        int size,i;
        float pr;
        int co;
        cout<<"enter the size";
        cin>>size;
        item *ptr=new item[size];
        item *d=&ptr[0];//or item *d=ptr;

        for(i=0;i<size;i++){
        cout<<"enter the code and price";
        cin>>co>>pr;
        ptr->get(co,pr);
         ptr++;
}
```

```
for(i=0;i<size;i++){
        d->display();
        d++;
}
return 0;
}
```

Q1)WAP to input two variable and display their sum using pointer to an object.

Ans:

```
#include<iostream>
using namespace std;
class demo{
        int a,b,sum;
        public:
                void get(int x,int y){
                        a=x;
                        b=y;
                }
                void display(){
                        sum=a+b;
                        cout<<"sum="<<sum;
                }
        };
        main(){
                int a,b;
                cout <<"enter two numbers:"<<endl;
                cin>>a>>b;
                demo *ptr=new demo;
                ptr->get(a,b);
                ptr->display();
                delete ptr;
        }
```

Q2)WAP to input name,roll and id of ‘n’ number of students by creating array of object using pointer.

Ans:

```
#include<iostream>
#include<string.h>
using namespace std;
class demo{
    int r,i;
    char na[50];
    public:
        void get(char a[50],int b,int c){
            strcpy(na,a);
            r=b;
            i=c;
        }
        void display(){
            cout<<"name="<<na<<"id="<<i<<"roll="<<r;;
        }
    };
    main(){
        char name[50];
        int n,i,roll,id;
        cout<<"enter total number of students:";
        cin>>n;
        demo *ptr=new demo[n];
        demo *d=&ptr[0];
        for(i=0;i<n;i++){
            cout<<"enter name,roll and id of"<<i+1<<"student:";
            cin>>name>>roll>>id;
            ptr->get(name,roll,id);
            ptr++;
        }
        cout<<"details of students are:"<<endl;
        for(i=0;i<n;i++){
                d->display();
                d++;
        }
 delete ptr,d;
 return 0;
}
```

Pointer to derived class

- Pointers to object of a base class are type-compatible with pointers to object of derived class.So,a single pointer can be made to point to objects belonging to different classes.

  If B be the parent class and D be the child class then:

  ```
  B *cptr;//pointer to class B
  B b;//base class object
  D d;//child classs object
  cptr=&b;//cptr points to object b
  cptr=&d;//cptr points to object d
  ```
- Using cptr we can only access those members which are inherited from B and not the members that originally belong to D.
- If D and B have the same name function then any call to that function by cptr will always access the base class function.
- Although ,a base pointer can be used to point to any object derived from that base(i.e. derived class object),the pointer cannot be directly used to access all members of derived

class.Here,we should use another pointer declared as pointer to the derived class type.

Eg:

```
#include<iostream>
using namespace std;
class A{

        public:
                int a;
                void show(){
                        cout<<"value of a="<<a;
                }
};
class B:public A{

        public:
                int b;
                void show(){
                        cout<<"value of b="<<b;
                }
};
main(){
        A ob;//object of base class
        A *ptr1=&ob;//ptr1 points to base object
        ptr1->a=2;
        ptr1->show();//show() of class A is executed
         //ptr1->b=5;//cant be done
         B obj1;//derived class object
        ptr1=&obj1;//ptr1 points to child object
         ptr1->a=10;//initialize a=10 through base pointer
         ptr1->show();//show() of base class is executed
         //ptr1->b=6;//cant be done
         B *ptr2;//base class pointer
         ptr2=&obj1;//ptr2 points to base class object
        ptr2->a=3;//initialize a=3 through child pointer
         ptr2->b=9;//initilaize b=9 of child class
         ptr2->show();//show() of child class
         //casting ptr2 to pointer of parent class
        ((A*)ptr2)->a=5;
        ((A*)ptr2)->show();
         //casting ptr1 to pointer of derived class i.e ptr2
         ((B*)ptr1)->b=20;
         ((B*)ptr1)->show();
}
```

Output

```
Value of a=2

Value of a=10

Value of b=9

Value of a=5

 Value of b=20
```

Virtual Function

- A base pointer even when it is made to contain the address of a derived class,it always executes the function in the base class.Here,the compiler simply ignores the content of the pointer and chooses the member function that matches the type of the pointer .In this case polymorphism is achieved by using properties known as virtual function.

- When we use the same function name in both the base and derived classes,the function in base class is declared as virtual using keyword “virtual”.
- When a function is made virtual ,c++ determines which function to use at run time based on the type of object pointed by the base pointer,rather than the type of the pointer.

Eg:

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

                }
                    virtual void show(){
                         cout<<"show base"<<endl;
                    }
};
class derived:public base{
                public:
                    void display(){
                         cout<<"display derived";
                    }
                    void show(){
                         cout<<"show derived";
                    }
};
main(){
 base b;
```

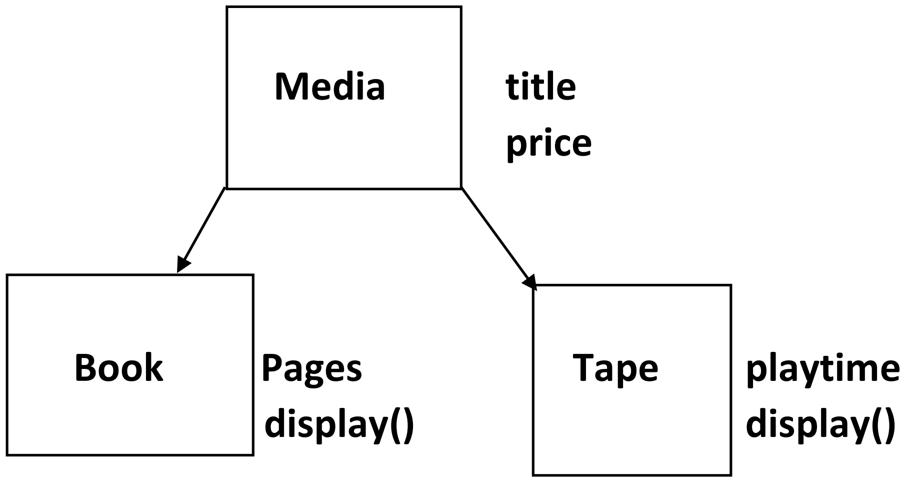
```
  derived d;
  base *bptr;
  bptr=&b;
  bptr->display();
  bptr->show();
  bptr=&d;
  bptr->display();
}
```

Note:

- We cannot use a pointer to a derived class to access an object of the base class by putting the address of the base class object to that child class pointer.
- Virtual function are always accessed by using the object pointers.
- A virtual function in a base must be defined even though it may not be used.
- We cannot have the virtual constructors but we can have virtual destructors.

****Run time polymorphism is achieved only when virtual function is accessed through a pointer to the base class.**

Eg:



```
};
```

Ans:

```
#include<iostream>
#include<string.h>
using namespace std;
class Media{
                        protected:
                              char title[50];
                              float price;
                              public:
                                     Media(char t[50],float a){
                                            strcpy(title,t);
                                            price=a;
                                     }
                                     virtual void display(){

                                     }
};
class Book:public Media{
                        int pages;
                        public:
                              Book(char s[50],float a,int p):media(s,a){
                                     pages=p;
                              }
                              void display(){
                                     cout<<"title:"<<title;
                                     cout<<"pages="<<pages;
                                     cout<<"price="<<price;
                              }

};
```

```
class Tape:public Media{
                    float playtime;
                    public:
                         Tape(char s[50],float a,float t):media(s,a){
                               playtime=t;
                         }
                         void display(){
                               cout<<"title="<<title;
                               cout<<"playtime="<<playtime;
                               cout<<"price="<<price;
                         }
                    };
                    main(){
                    char title[50];
                    int pages;
                    float time,price;
                    cout<<"enter books details:";
                    cout<<"title:";
                    cin>>title;
                    cout<<"price:";
                    cin>>price;
                    cout<<"pages:";
                    cin>>pages;

                    Book book1(title,price,pages);

                    //tape details
                    cout<<"enter tape details:";
                    cout<<"title:";
                    cin>>title;
```

```
                cout<<"price:";
                cin>>price;
                cout<<"play time(mins)";
                cin>>time;
                Tape tape1(title,price, time);

                Media *list[2];
                list[0]=&book1;
                list[1]=&tape1;

                cout<<"media details:";
                cout<<"book::";
                list[0]->display();
                cout<<"tape::";
                list[1]->display();
}
```

Pure virtual function/Deferred method/Abstract method

- A virtual function declared in a base class that has no definition relative to the base class is known as a **pure virtual functions**.It is also known as **do-nothing functions**.
  Syntax:
  virtual return-type function-name()=0;
  eg:
  virtual void display()=0;

- If the base class contains pure virtual function ,the compiler requires each derived class of that base class to either define the function or redeclare it as a pure virtual function.
- A class containing pure virtual functions cannot be used to declare any objects of its own,such class that contains at least one pure virtual function is known as **abstract class.**
- This abstract classes is one that is not used to create any object but the main **objective/advantage/use of the abstract class** is to provide some traits/features to the derived classes and to create a base pointer required for achieving run-time polymorphism.

  Eg:

```
#include<iostream>
using namespace std;
class balaguruswamy{
            public:
                virtual void display()=0;
};
class c:public balaguruswamy{
            public:
                void display(){
                    cout<<"c text book written by
balaguruswamy"<<endl;
                }
};
class oops:public balaguruswamy{
            public:
                void display(){
                    cout<<"c++ text written by
balaguruswamy";
```

```
                    }
        };
        main(){
                    balaguruswamy *bptr[2];
                    c obj;
                    oops obj1;
                    bptr[0]=&obj;
                    bptr[1]=&obj1;
                    bptr[0]->display();
                    bptr[1]->display();
        }
```

Virtual constructors and virtual destructors

A constructor cannot be virtual because of the following reasons:

1. To create an object the constructor of the object class must be of the same type of the class.But,it is not possible with virtually implemented constructor.
2. At the time of calling a constructor,the virtual table would not have been created to resolve any virtual functions calls.

But a destructor can be virtual.

Virtual destructor is required in an inheritance when the base class pointer is holding the address of a derived class object.

Situation:

```
class A{

public:

~A(){
```

```
cout<<”base class destructor”;
}
};
class B:public A{
public:
~B(){
cout<<”child class destructor”;
}
};
main(){
A *ptr=new B;//base class pointer points to the derived class object
//body of the main
delete ptr;
}
```

Here,in this code delete ptr statement calls the base class destructor only and the derived class destructor can’t be called.So,to solve this problem base class destructor is made virtual i.e.

```
virtual ~A(){
cout<<”base class destructor”;
}
```

Now,the derived class destructor is also called.

Output:

```
child class destructor
```

base class destructor

Function Overriding

Function overriding is **defined** as a feature that allows us to have a same function in child class which is already present in the parent class. It is like creating a new version of an old function, in the child class.

To override a function, you must have the same signature in child class. By signature it means the data type and number of parameters.

Here,in the given code ,display() is overridden we don't have any parameter in **the parent function so we didn't use any parameter in the child function.**

Eg:

```
#include<iostream>

using namespace std;

class base{

    protected:

    int a;

    public:

        void geta(){

            cout<<"enter a:";

            cin>>a;
```

```
		}
		void display(){
			cout<<"a="<<a;
		}
};
class child:public base{
	int b;
	public:
		void getb(){
			cout<<"enter b:";
			cin>>b;
		}
		void display(){
			cout<<"b="<<b;
		}
};
main(){
	child c;
	c.geta();
```

```
        c.getb();

        c.display();

}
```

Output:

```
Enter a:5

Enter b:4

b=4
```

Here ,in this case display() is overridden .It means when the child class object calls the display() then the child class display() is executed but not of the parent class.

Now to access the overridden display() of parent class we use:

```
Base b;

b.display();
```

this displays a=5

Type Conversion

Eg:

```
int m;

float x=5.34;

m=x;
```

Here,5.34 is reduced to 5 and then it is stored in m.In this case the compiler automatically performs the type conversion which is known as the implicit conversion.

- Type conversion are implicit as long as the data types involved are built-in-basic data type.
- But when we try to perform the conversion like c3=c1+C2,where the one of the operand is basic type and the other is an object or what if they belong to different classes?In such cases compiler donot perform the implicit conversion,we should manually handle it(i.e. we should design the conversion steps by ourself).

Type conversion are of three types:

1. Basic to class type.
2. Class to basic type.
3. Class to class type.

1)Basic to Class type

This conversion can be done by using constructor in a class.Here, the constructor used for the type conversion takes a single argument whose type is to be converted.

Eg:

A program to convert minutes(basic type) to the hour and minute of the class by using type conversion.

Ans:

```
#include<iostream>

using namespace std;

class time

{
     int hr,min;
    public:
           time(int m){
                     hr=m/60;
                    min=m%60;
                  }
          void display(){
                cout<<"hour="<<hr<<"minute="<<min;
                     }
};
main(){
    int timeinmins=80;
    time t1=timeinmins;//time t1(timeinmins)
     t1.display();
}
```

Questions:

1)WAP to convert height in inches to feet and inch using type conversion.

Ans:
```
#include<iostream>
using namespace std;
class height{
                                int inch,ft;
```

```
                public:
                        height(int i){
                                ft=i/12;
                                inch=i%12;
                        }
                        void display(){
                                cout<<"feet="<<ft<<"inch="<<inch;
                        }
};
main(){
                int totalinch;
                cout<<"enter the height in inch:";
                cin>>totalinch;
                height h=totalinch;
                h.display();


}
```

2)WAP to convert total distance in meter to km and m using type conversion.

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

```
                    int mt,km;
                    public:
                            dist(int m){
                                    km=m/1000;
                                    mt=m%1000;
                            }
                            void display(){

                    cout<<"kilometer="<<km<<"meter="<<mt;
                            }
};
main(){
                    int m;
                    cout<<"enter total distance in meter:";
                    cin>>m;
                    dist d=m;
                    d.display();
}
```

3)WAP to convert the total memory in bytes into mb,kb and bytes using type conversion.

Ans:

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

```
class memory{
                    long int mb,kb,b;
                    public:
                        memory(long int by){
                            int rem;
                            mb=by/(1024*1024);
                            rem=by%(1024*1024);
                            kb=rem/1024;
                            b=rem%1024;
                        }
                        void display(){
                            cout<<"megabytes="<<mb<<"kilo
bytes="<<kb<<"bytes="<<b;
                        }
};
main(){
                    long int bytes;
                    cout<<"enter total bytes:";
                    cin>>bytes;
                    memory m=bytes;//distance d=m
                    m.display();
}
```

2)Class to Basic type

Here,instead of using constructor, overloaded operator function is used which is known as the **conversion function**.

General form:

operator typename(){

//body of function

}

This function converts a class type to the data of the typename used in the conversion function.

Casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify any return type.
- It must not contain any arguments/parameters.

Conversion function is a member function ,so it is invoked/called by the object and therefore,the values used for conversion inside the function belongs to the object that called the function.This means conversion function donot need any arguments.

Eg:

A program to convert time in hr and min to total time in minutes using type conversion.

Ans: #include<iostream>

using namespace std;

```
class time{
                              int hr,min;
                              public:
                                    time(int h,int m){
                                           hr=h;
                                           min=m;
                                    }
                                    void display(){

                              cout<<"hour="<<hr<<"minute="<<min;
                                    }
                                    operator int(){
                                           int minutes;
                                           minutes=hr*60+min;
                                           return minutes;
                                    }
};
main(){
                              time t1(1,30);
                              int total;
                              total=t1;//total=t1.int()
                              cout<<"total time in minutes="<<total;
}
```

Questions:

1)WAP to convert the height in ft and inches to the total inches using type conversion.

Ans:
```
#include<iostream>
using namespace std;
class height{
                int ft,inch;
                public:
                    height(int f,int i){
                        ft=f;
                        inch=i;
                    }
                    void display(){
                        cout<<"feet="<<ft<<"inches="<<inch;
                    }
                    operator int(){
                        int inches;
                        inches=ft*12+inch;
                        return inches;
                    }
};
```

```
main(){
                    height h1(1,30);
                    int totalinch;
                    totalinch=h1;//totalinch=h1.int()
                    cout<<"total height in inches="<<totalinch;
}
```

2)**WAP to convert distance in m and km to total distance in meter using type conversion.**

```
Ans: #include<iostream>
using namespace std;
class dist{
                    int km,m;
                    public:
                        dist(int a,int b){
                            km=a;
                            m=b;
                        }
                        void display(){

                    cout<<"kilometer="<<km<<"meter="<<m;
                        }
                        operator int(){
                            int meters;
                            meters=km*1000+m;
                            return meters;
                        }
};
main(){
                    dist d1(1,30);
                    int totalmeters;
                    totalmeters=d1;//totalmeters=d1.int()
                    cout<<"total distance in meters="<<totalmeters;
}
```

3)WAP to convert memory in bytes,kilobytes and megabytes into total bytes using type conversion.

Ans:

```
#include<iostream>
using namespace std;
class memory{
            long int mb,kb,by;
            public:
                memory(long int a,long int b ,long int c){
                    mb=a;
                    kb=b;
                    by=c;
                }
                operator long int(){
                        long int x;
                        x=mb*(1024*1024)+kb*1024+by;
                        return x;
                }
                void display(){

                        cout<<"megabytes="<<mb<<"kilo
                bytes="<<kb<<"bytes="<<by;
                }
};
        main(){
                memory m(2,3,4);
                m.display();
                long int bytes;
                  bytes=m;
                cout<<"total memory in bytes="<<bytes;
        }
```

3)Class to Class type

Conversion of data of one class to another class.

Eg:

Let X and Y be two classes then:

objX=objY;

Here,objX is an object of class X and objY is an object of class Y.Class Y type data is converted to the class X type .

Here,Y is known as the **source class** and X is known as the **destination class**.

This class to class type conversion can be done in any of the two ways:

a)Using casting conversion function in the source class

Syntax:

```
operator destination-class-name(){
//body of the conversion function
}
```

b)Using constructor in destination class

Synatx:

```
destination-class-name(source class-object){
//body of the constructor
}
```

In this case of using constructor in destination class for conversion we must use special access function in source class to access its private data by destination class constructor.

Eg:

A program to convert rectangle into polar by using type conversion.

->>by using conversion function in source class:

Ans:

```
#include<iostream>
using namespace std;
#include<conio.h>
#include<math.h>
class Polar{
  float r,a;
  public:
     Polar(){
     r=0;
     a=0;
  }
  Polar(float rr,float aa){
    r=rr;
    a=aa;
  }
  void display(){
      cout<<"r="<<r<<" a="<<a<<endl;
  }
```

```
};
class Rect{
 float x,y;
 public:
  Rect(float xx,float yy){
   x=xx;
    y=yy;
  }
  void display(){
    cout<<"x="<<x<<" y="<<y<<endl;
  }
  operator Polar(){
   float a=atan(y/x);
    float r=sqrt(x*x+y*y);
  return Polar(r,a);
  }
};
main(){
 Rect R(4,5);
  Polar P;
  P=R;
  P.display();
  R.display();
```

```
   getch();
}
```

→by using constructor in destination class:

Ans:

```
#include<iostream>
using namespace std;
#include<conio.h>
#include<math.h>
class Rect{
  float x,y;
  public:
   Rect(float xx,float yy){
   x=xx;
    y=yy;
   }
   float getx(){
      return x;
   }
   float gety(){
    return y;
   }
   void display(){
      cout<<"x="<<x<<" y="<<y<<endl;
   }
};
class Polar{
  float r,a;
   public:
         Polar(){
                 r=0;
                 a=0;
   }
   Polar(Rect re){
        r=sqrt(re.gety()*re.gety()+re.getx()*re.getx());
         a=atan(re.gety()/re.getx());
   }
   void display(){
      cout<<"r="<<r<<" a="<<a<<endl;
   }
};
main(){
   Rect R(4,5);
   Polar P;
   P=R;
   P.display();
   R.display();
    getch();
}
```

Questions:

1) **WAP to convert polar into rectangle by using type conversion. (By using constructor in destination class)**

   Ans:

```
#include<iostream>
using namespace std;
#include<math.h>
class polar{
                float r,a;
                public:
                        polar(float rr,float aa){
                                r=rr;
                                a=aa;
                        }
                        void display(){
                                cout<<"radius="<<r<<"angle="<<a;
                        }
                float getr(){
                        return r;
                }
                float geta(){
                        return r;
                }
};
class rect{
                    float x,y;
                    public:
                         rect(){
                               x=0;
                               y=0;
                                     }
                         rect(polar p){
                               x=p.getr()*cos(p.geta());
                               y=p.getr()*sin(p.geta());
                         }
                         void display(){
                               cout<<"x="<<x<<"y="<<y;
                         }
};
main(){
                    polar p(1,45);
                    p.display();
                    r=p;//rect r(p)
                    r.display();
}
```

2) WAP to convert polar into rectangle by using type conversion. By using conversion function in source class

Ans:

```
#include<iostream>
using namespace std;
#include<math.h>
class rect{
};
                    float x,y;
                    public:
                            rect(){
                                    x=0;
                                    y=0;
                                            }
                            rect(float a,float b){
                                    x=a;
                                    y=b;
                            }
                            void display(){
                                    cout<<"x="<<x<<"y="<<y;
                            }
};
class polar{
                    float r,a;
                    public:
                            polar(float rr,float aa){
                                    r=rr;
                                    a=aa;
                            }
                            void display(){
                                    cout<<"radius="<<r<<"angle="<<a;
                            }
                            operator rect(){
                                    float xx=r*cos(a);
                                    float yy=r*sin(a);
                                    return rect(xx,yy);
                            }
};
```

```
main(){
                    polar p(1,45);
                    p.display();
                    rect r;
                    r=p;
                    r.display();
}
```

this pointer

c++ uses a unique keyword that is used to represent an object that invokes a member function.this is a pointer to the object for which this function was called.For eg:the function A.max() will set the pointer this to the address of the object A.The starting address is the same as the address of the first variable in the class.

This pointer is automatically passed to a member function when it is called.It is considered as an implicit argument to all the member functions.

Eg:

```
#include<iostream>

using namespace std;

#include<string.h>

class person{

                    char name[20];

                    float age;

                    public:
```

```
            person(char n[20],float a){
                  strcpy(name,n);
                  age=a;
            }
            void display(){

      cout<<"name="<<name<<"age="<<age;
            }
            person compare(person p){
                  if(p.age>=age){
                        return p;
                  }
                  else{
                        return *this;
                  }
            }
};
main(){
      person p1("john",37.2);
      person p2("ram",23);
      person p=p1.compare(p2);
      cout<<"elder one is:";
```

```
                    p.display();
}
```

The End