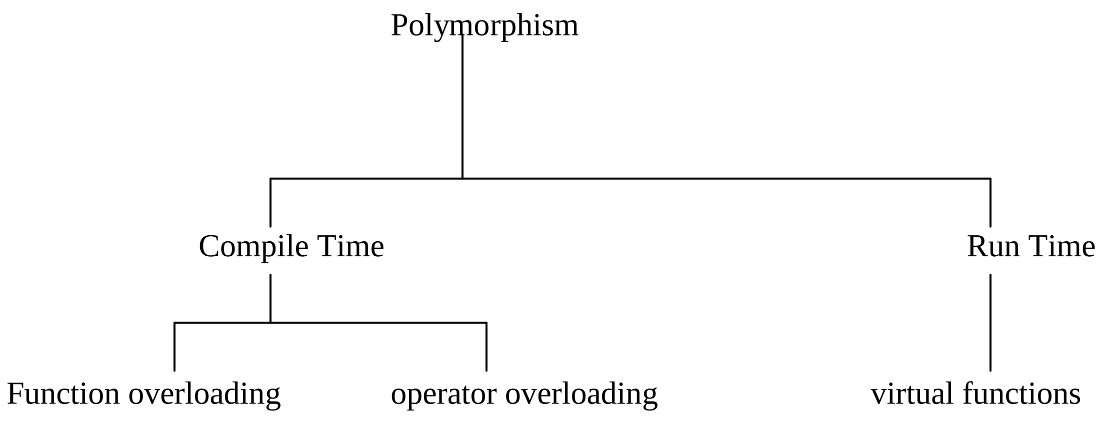
Phymorphism: "one name but different forms"

- Polymorphism is implemented by function over loading and operator overloading
- Giving same name to functions with different no or types of arguments
- This information is known to compiler at compile time and compiler is able to select the appropriate function at call time
- It is called compile time polymorphism-**Early binding**.

# If appropriate member function are chooses at run time rather than compile time, this is known as runtime polymorphism or **Late binding.**
# For this mechanism we chose member function as virtual.

In C++, polymorphism indicates the form of a member function that can be changed at runtime. Such member functions are called virtual functions and the corresponding class is called polymorphic class. The objects of the polymorphic class, address by pointers, change at runtime and respond differently for the same message. Such mechanism requires postponement of binding of a function call to the member function declared as virtual until runtime.



Virtual Function: A virtual function is a function that appears to exists but does not exist in reality.

Declaration is done in base class as.

```
      class Test
{
      public:
            ...............
            virtual retrun_type function_name(args……..)
            {
                  //function body
            }
};
```

- When we use same function in both base class and derived class and access them using pointer of base class objects assigning the address of objects of derived class, we expect that the function should perform the operation of which base pointer is assigned the address but we can not get such result using general member function. For this we can use virtual functions.

- Virtual function is declared in base class with keyword virtual and without keyword virtual in other derived class.
- An object pointer of base class is used for assigning the address of the object of derived class and to invoke the member function of same name for different task.

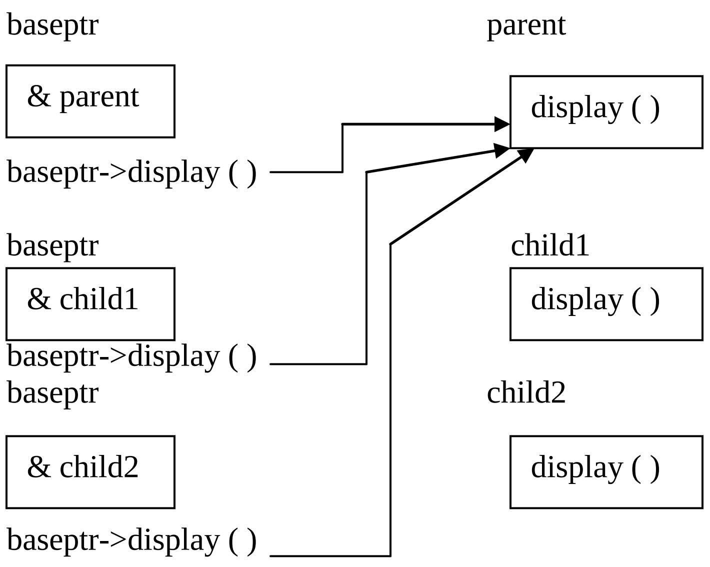
Look Example:

A normal function accessed with object pointer.

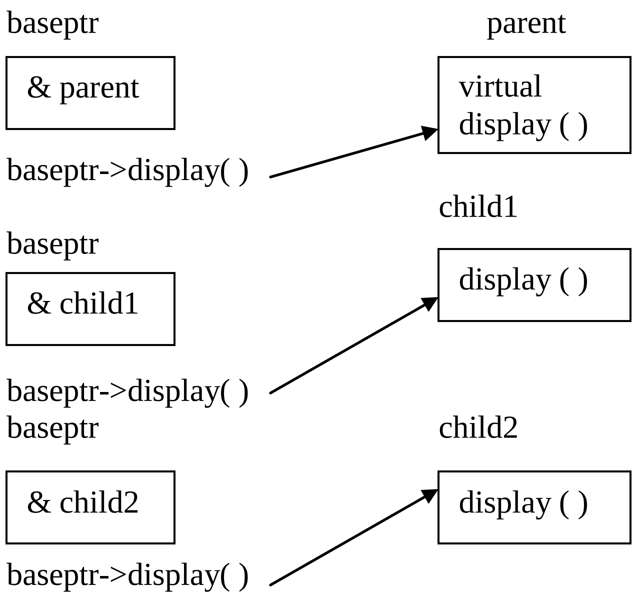
```
//normal function
#include <iostream.h>
class parent
{
     private :
          int a ;
     public:
          parent ( ) { a = 1 ; }
          void display()
          {
            cout <<" \n value from parent class :" << a ;
          }
};
class child1: public parent
{
     int a ;
     public:
          child1( ) { a = 2;}
          void display()
          { cout <<" \nValue from child1 is :" << a;}
};

class child2 : public parent
{    int a;
     public:
     child2() { a = 3;}
     void display()
      { cout <<" \n Value from child2 is:"<<a;}
};
void main ( )
{
     parent  * baseptr;
     parent    p;
     baseptr = &p;
     baseptr->display();//execute display ( ) from parent
     child1 c1;
     child2 c2;
     baseptr = &c1 ;
     baseptr->display(); // expect  for execution of
                         //display ( ) of child1
     baseptr = &c2 ;
     baseptr->display(); // expectation for execution of
}                        //display ( ) of child2
```

Run:
```
Value from parent class 1
Value from parent class 1
Value from parent class 1
```
Which is not our expectation, our expectation was
```
Value from parent class 1
Value from child1 class 2
Value from child2 class 3
```

Normal – nonvirtual pointer access



Virtual pointer access



To achieve correct result we can use virtual function

```
#include <iostream.h>
class parent
{
     private :
     int a ;
     public:
     parent() { a = 1 ; }
     virtual void display()
     { cout <<" \n value from parent class :" << a ; }
};
class child1: public parent
{
     int a ;
     public:
     child1() { a = 2;}
     void display()
      { cout <<" \nValue from child1 is :" << a;}
};
class child2 : public parent
{    int a;
     public:
     child2() { a = 3;}
     void display()
     { cout <<" \n Value from child2 is:"<<a;}
} ;
void main ( )
{
     parent  * baseptr;
     parent    p;
     baseptr = &p;
     baseptr->display(); // execute display ( ) from parent

     child1 c1;
     child2 c2;
     baseptr = &c1 ;
     baseptr->display(); // execute display ( ) of child1
     baseptr = &c2 ;
     baseptr->display(); // execute display ( ) from child2
}
```

Run:

```
Value from parent class : 1
Value from child1 class : 2
Value from child2 class : 3
```

Which is correct expectation.

Abstract classes and pure virtual function:

When the objects of base class are never instantiated, such a class is called abstract base class or simply or simply abstract class. Such a class only exists to act as a parent of derived classes from which objects are instantiated. It may also provide interface for class hierarchy.

To make a class abstract so that object instantiation is not allowed and derivation of child classes are allowed, at least one pure virtual function should be placed in the class.

A pure virtual class is one with the expression = 0 is added to the declaration of virtual function. The syntax of declaration of pure virtual function and making a class abstract is:

```
class A
{
      public:
            virtual void show() = 0 ; //pure virtual function
};
```

Here class A become abstract since there is presence of pure virtual function show ( ). The expression = 0 has no any other meaning the equal sign = 0 does not assign 0 to function show (). It is only method to tell the compiler that show () is pure virtual function hence class A is abstract class.

All classes with pure virtual function are known as concrete classes.

A pure virtual function has following properties.

1. A pure virtual function has no implementation in the base class.
2. It acts as an empty bucket (virtual function is partially filled bucket) that the derived class are supposed to fill it.
3. A pure virtual function can be invoked by its derived class.

```
//pure virtual function
#include<iostream.h>
#include<conio.h>

class Base
{
      public:
            virtual void show()=0;//pure virtual function
};

class Derv1: public Base
{
      public:
      void show()
      {cout<<"\n I am Derv1 class";}
};

class Derv2: public Base
{
      public:
      void show()
      {cout<<"\n I am Derv2 class";}
};

void main()
```

```
{
      Derv1 dv1;
      Derv2 dv2;
      Base *ptr;//pointer to base class

      ptr=&dv1;//address of dv1 in pointer
      ptr->show();//called from Derv1

      ptr=&dv2;
      ptr->show();//called from Dev2
      getch();
}
```

The pure virtual function in the base class must be override in all its derived class from which we want to instantiate objects. If a class doesn't override pure virtual function, it itself becomes abstract and objects cannot be instantiated.

Array of Pointers to Base class objects

```
//array of pointers to base class object
#include<iostream.h>
#include<conio.h>
class Base
{
      public:
            virtual void show()=0 ;  // pure virtual function
};
class Derived1 : public Base
{
      public :
            void show() { cout <<" Derived1 \n"; }
};
class Derived2 : public Base  // derived class 2
{
      public:
            void show() { cout <<" Derived2 \n" ; }
};
class Derived3 : public Base
{
      public:
            void show() { cout <<" Derived3 \n" ; }
};
class Derived4 : public Base
{
      public:
            void show() { cout <<" Derived4 \n" ; }
};
void main()
{
      // Base baseobj ; // can't make object of abstract class

      Derived1 dv1 ;          // object of derived1
      Derived2 dv2 ;          //  object of derived2.
      Derived3 dv3 ;
      Derived4 dv4 ;
      Base *ptr[]={&dv1,&dv2,&dv3, &dv4} ;// array of ptr of base
                                          //class.
      cout<<"Calling functions..."<<endl;
      for(int i=0;i<4;i++)
            ptr[i]->show();
      getch();
}
```

Virtual destructors: Since destructors are member functions, they can be made virtual with placing keyword virtual before it. The syntax is

virtual ~ classname ( ) ; // virtual destructor.

The destructor in base class should always be virtual. If we use **delete** with a base class object to destroy the derived class object, then it calls the **delete** calls the member function destructor for base class. This causes the base class object to be destroyed. Hence making destructor of base class virtual, we can prevent such miss-operation.

Example:

```
#include<iostream.h>
class Base
{
 public:
       ~Base ( ) ;      // non virtual
     //virtual ~Base()
     {cout<<"Base Destryed\n";}
};
 class Derv1:public Base
 {
     public:
     ~Derv1()
     {
     cout<<"Derived1 destroyed\n";
     }
 };
 class Derv2:public Base
 {
     public:
     ~Derv2()
     {
     cout<<"Derived2 destroyed\n";
     }
 };

 void main()
 {
  Base * pBase = new Derv1;
  delete pBase;
 }
     The output for it is :
          Base destroyed.
```

- pBase stores address of object of Derv1 class.
- Delete pBase destroy the Base object i.e. calls the destructor of base class.
- If the destructor is made virtual by the line virtual ~Base ( ) ;

then,

```
        delete pBase ;
```

Simply calls the destructor of Derv class first and the output is now

Derv destroyed.
Base destroyed.

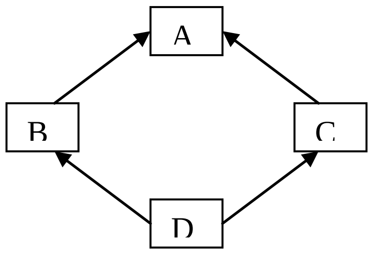
Virtual Base class:

In multiple inheritance, if a base class parent derives its two child class then another class is derived from two child, as

When member function of class D want to access data member of parent class A, then problem arises due to ambiguity.

To resolve such ambiguity we use virtual base class.

A virtual base class is one from which classes are derived virtually. as



```
class A.
{    // body of class A
};
class B: virtual public A
{
// Body of B
};
class C:virtual public A
{    // Body of class C
};
class D: public B; public C.
{
};
```

Example:

```
class parent                class child1: public parent
{ protected:                {  };
int basedata;               class child2: public parent
};                          { };
class grandchild: public  child1, child2
{    public:
     int getdata()
     { return basedata; }      // Error: ambiguous
};
```

When the member function of grandchild attempts to access base data in parent, each child$_1$ and child$_2$ inherits the copy of basedata. Since grandchild class is derived from both child$_1$ and child$_2$, so attempting to access base data becomes ambiguous in grandchild.

This ambiguity is overcome by making virtual base class as

class child$_1$: virtual public parent
{ };
class child$_2$: virtual public parent
{ };

The use of virtual in these two class causes them to share a single common copy of base data. So attempt to access base data in grandchild is not ambiguous.

Friend function and friend classes.

Private member of a class can not be accessed from outside the class. Non member function of a class cannot access the member of class. Using friend function we can achieve this. A fried function also acts as a bridging between two class. It can operate the object of two classes.

- Friend function is declared inside the class any where in private or public section with keyword friend.
- Friend function are defined outside the class as a normal function.

An example: Friend function operating objects of two classes.

```
#include<iostream.h>
class beta ;
class alpha
{
     private:
          int data ;
     public:
          alpha() { data=10;}
          friend int frifunc(alpha ,beta);
};
class beta
{    private:
          int data;
     public:
      beta() { data = 20; }
     friend int frifunc(alpha, beta);
};
int frifunc(alpha a, beta b)
{    return (a.data + b.data);
}
void main ( )
{
     alpha aa;
     beta  bb;
     cout<<frifunc(aa, bb)<<endl;
}

output:  30
```

Friend class:

When a class is to be made friend to another class, we should declare that class as friend inside another. as

```
     class A                  class B
     {    // body of A        {    // Body of B
     friend class B;
     };                        };
```

Here class B is friend class to A.

• When a class is declared as friend within another class entire function of friend class can access the private data of the class to which it becomes friend.

- Here in above example, all member function of B can access private data of class A.

Another way of declaration

```
     class B;
     class A
     {
     // Body of class A
     friend B;
     };

     class B
     {
       // Body
     };
```

//An Example:

```
#include<iostream.h>
class A
{
     private:
     int data;
     public:
     A(){data = 100;}
     friend class B;
};

class B
{
     public:
     void func1(A a)
     {cout<<"\ndata= "<<a.data;}
     void func2(A b)
     {
          cout<<"\ndata = "<<b.data<<endl;
     }
};

void main()
{
     A a;
     B b;
     b.func1(a);
     b.func2(a);
}
```

```
The output: data= 100
            data = 100
```

Static Function: A static function is one which is declared as static in a class. A static function can access only static member data and can be accessed by using class name rather than object name.

Following example shows the static function as class member

```
#include<iostream.h>
class staticfun
{
     private:
     static int count;    // count objects.
     int id;
     public:
     staticfun()          // constructor
     {    count++;
          id=count;
     }

     ~staticfun ()        // destructor
     {    count--;
          cout<<"Destroying ID number"<<id<<endl;
     }
     static void show()   // static function
     {
          cout<<"No of object is:"<<count<<endl;
     }
     void showid()
     {
     cout<<"ID number is:"<<id<<endl;
     }
};
int staticfun::count=0;   // defn of count.

void main()
{    staticfun s1;
     staticfun:: show();
     staticfun s2,s3;
     staticfun:: show();

     s1.showid();
     s2. showid();
     s3. showid();
     cout<<"---END---"<<endl;
}
```

• When a data member is declared as static, there is only one such value for the entire class. All objects of the class share the same data.

• To access such static data, we use static function that need not refer by any object and can be called by class name with scope resolution operator(::) as

class name:: static function();

The output of above program will be now:

```
No of object is: 1
No of object is: 3
ID number is: 1
ID number is: 2
ID number is: 3
 ——— END ———
Destroying ID number3
Destroying ID number2
Destroying ID number1
```

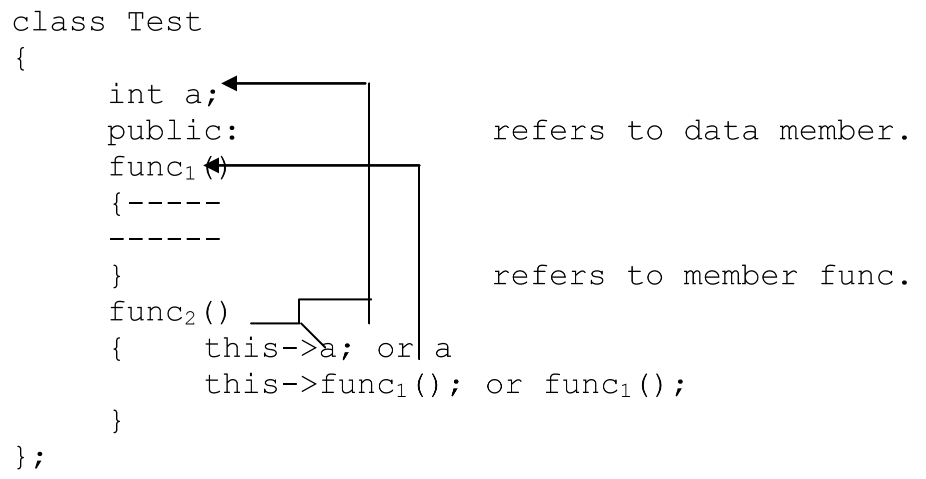
'this' pointer:

The member functions of every object have access to a magic pointer named '**this**' which points to the object itself. The **this** pointer is implicitly defined in each member function.

→ Every member function of a class is born with a pointer called **this**. Which points to the object with which the member function is associated.

→ When member function is invoked, it comes into existence with the value of **this** set to the address of the object for which it is called.

Method of accessing member of class with **'this'** is as in figure below.



```
class Test
{
        int a;
        public:                     refers to data member.
        func1()
        {-----
        ------
        }                           refers to member func.
        func2()
        {       this->a; or a
                this->func1(); or func1();
        }
};
```

Example:
```
#include<iostream.h>
class Test
{
        int a;
        public:
                void setdata(int x)
                { a=x;    //or this->a=x;
                  cout<<"Address of my object is :"<<this<<endl;
        }
        void showdata()
        {       // normal way.
        cout<<"Data accessed normally:"<<a<<endl;
        cout<<"Address of object in showdata():"
                                 <<this<endl;
        // data access through this
        cout<<"\nthis->a:"<<this->a;
    }
};
    void main()
    {
        Test myobj;
        myobj.setdata(20);
        myobj.showdata();
    }
```

Run: (Sample)

```
Address of my object is :0x22972446
Data accessed normally:20
Address of object in showdata():0x22972446
this->a:20
```

Using this for returning values:

For returning values from member functions, **this** can play important role. When an object is local to a function, the object will be destroyed when function terminates. So it is necessary for a more permanent object while returning it by reference.

Consider a function add() for addition of two complex object called as
`c3=c1. add(c2);` Where c_1, c_2, c_3 are complex no. objects. Our function will be as.

```
    complex  complex:: add(complex c2)
    {    real= real+ c2.real;
         imag=imag +c2.imag,
         return complex(real, imag);
    }
```

It adds c_2 to a default object and returns the updated default object by creating nameless temporary object by statement

```
              return complex(real, imag);
```

It can be replaced by the statement simply by

```
              return *this;  // returns object by value
```

Now definition of add will be

```
    complex add(complex c2)
    {    real=real+c2.real;
         imag=imag +c2.imag,
         return*this;
    }.
```