POINTERS TO OBJECTS

To create a pointer to an object of class demo

we write, demo *ptr;

Which creates a pointer of type demo class type. Now for an object say d of demo class declared as demo d; we can store address of this object into pointer ptr as :

ptr = &d;

Now any data member or function of demo class can be accessed using pointer as

ptr->func_name( ); and ptr->data_member;

As the pointer ptr contains the address of object d, *ptr denotes object d so we can also write

(*ptr).func_name and (*ptr).data_member;

We can also create objects dynamically and can store the address into pointer as

```
demo *ptr = new demo; OR
demo *ptr;
ptr = new demo;
```

Here, we don't have any object d as in the earlier case. Object will be referred only by pointer ptr.

Similarly to pointer to objects we can have a pointer to an array of objects or we can have an array of pointers to objects. To create pointer to an array of objects we can write as :

```
demo d[5];
demo * ptr = d;
```

And dynamically creating an array of object we can write demo *ptr = new demo [5]; which creates an array of objects of size 5 and returns the base address of this array.

The first object will be referred as ptr [0], second as ptr [1] and so on.

```
#include <iostream>
using namespace std;
class demo
{
   public :
        void show( )
        {
                cout<<"Hello from show"<<endl;
        }
        void bye_bye( )
        {
                cout<<"BYE BYE"<<endl;
        }
};
int main( )
{
        demo *ptr= new demo;
        ptr->show( );
        ptr->bye_bye( );
        return 0;
}
```

OUTPUT :

Hello from show

BYE BYE

we have 4 different ways to call functions of demo class using object d and pointer ptr.

(a) d.show( );

(b) (&d)->show( );

(c) Ptr->show( );

(d) (*ptr).show( );

THE THIS POINTER

The this pointer is a special pointer which is a built-in pointer. It is a keyword. It stores the address of current object in context. That is the current object which can be referred using this pointer anywhere in the class. The this pointer can be used only inside the class i.e., only inside the member function of the class and cannot be used outside the class. The this pointer is a constant pointer.

For example, the following function call

```
myDate.setMonth (3);
```

Can be interpreted this ways :

```
setMonth (&myDate, 3);
```

The object's address is available from within the member function as the this pointer. It is legal, through unnecessary, to use this pointer when referring to members of the class. The expression (*this) is commonly used to return the current object from a member function.

Important Points About this Pointer

(a) It is an implicit pointer used by the system.
(b) It stores the address of the current object in reference.
(c) It is a constant pointer to an object.
(d) The object pointed to by the 'this' pointer can be de-referenced and modified.
(e) It can only be used within non static functions of the class.
(f) The this pointer is non modifiable, assignment to this are not allowed.

```
#include <iostream>
using namespace std;
#include <string.h>
class Item
{
        char iname[10];
        int icode;
        float iprice;
  public :
        Item(char iname[10], int icode, float iprice)
        {
        strcpy(iname,iname);
        icode = icode;
        price = iprice;
        }
        void show( )
        {
        cout<<"Item Details"<<endl;
        cout<<"Name ="<<this ->iname <<endl;
        cout<<"Code ="<<(*this).icode <<endl;
        cout<<"Price ="<<this->iprice<<endl;
        }
};

int main( )
{
        Item I1("Mouse",301,280);
        I1.show( );
}
```

OUTPUT :

Item Details

Name=

Code =

Price =

Another example

```
#include <iostream>
using namespace std;
class demo
{
        public :
        demo( )
        {
        cout<<"Default constructor is called"<<endl;
        cout<<"Address of current object= "<<this<<endl;
        }
};
int main( )
{
        demo d1;
        cout<<"Address of object d1 in main="<<&d1<<endl;
        demo d2;
        cout<<"Address of object d2 in main="<<&d2<<endl;
}
```

OUTPUT :

Default constructor is called

Address of current object =0x8fd5fff4

Address of object d1 in main=0x8fd5fff4

Default constructor is called

Address of current object =0x8fd5fff2

Address of object d2 in main=0x8fd5fff2

```
#include <iostream>
using namespace std;
class Emp
{
   float sal;
   public :
        Emp( )
        {}
Emp(float s)
{
        sal=s;
}
Emp compare(Emp);
void show(char *s)
{
        cout<<s<<" "<<sal<<endl;
}
};
Emp Emp ::compare(Emp E)
{
        if (this -> sal>E.sal)
        return *this;
        else
        return E;
}
int main( )
{
        Emp E1(9500), E2(14000), E3;
        E3=E1.compare(E2);
        E1.show("Sal is=");
        E2.show("Sal is=");
        E3.show("Max sal=");
}
```

OUTPUT :

Sal is= 9500

Sal is= 14000

Max sal= 14000

The alternative code for compare function can be written as :

```
1. Emp Emp : :compare(Emp E)
  {
       Emp temp;
       if (sal>E.sal)
       temp.sal=sal;
       else
       temp.sal=E.sal;
       return temp;
   }

2. Emp Emp : :compare(Emp E)
  {
       if(this-> sal> E.sal)
       return Emp(this->sal);
       else
       return Emp(E.sal);
  }
```

OBJECT SLICING

Object slicing is a process of removing off the derived portion of the object when an object of derived class is assigned to a base class object. Only the base class data are copied to derived class object. Consider the following two classes :

```
class A
{
public :
        int Ax, Ay;
};
class B :public A
{
public :
        int Bx;
};
```

Through inheritance public data members Ax and Ay of class A are copied to class B. When in the main we write as :

```
        B b1;
        A a1;
        a1=b1;
```

Only the data members of object b1 which were inherited from class A are assigned to object a1. The data member Bx of object b1 is not copied. That is we say that object b1 was sliced off.

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

class demo
{
```

```
public :
        void show_demo( )
        {
                cout<<“Hello from show of demo”<<endl;
        }
};
class der_demo :public demo
{
public :
        void show_der( )
        {
        cout<<“Hello from show of der_demo”<<endl;
        }
};
int main( )
{
        demo d1;
        der_demo d2;
        d1=d2;
        d1.show_der( );
        return 0;
}
```

OUTPUT :

ERROR MESSAGE

error: ‘class demo’ has no member named ‘show_der’;

```
#include <iostream.h>
#include <conio.h>
class demo
{
  public :
        int bx,by;
demo(int x,int y)
```

```
{
        bx=y;
        by=y;
}
demo( ){}
};
class der_demo :public demo
{
  public :
        int dx;
der_demo(int x,int y,int z) :demo(x,y),dx(z)
{
}
};
int  main( )
{
        der_demo d2(10,20,30);
        demo d1;
        d1=d2;
        cout<<d1.bx<<“\t”<<d1.by<<“\t”<<d1.dx;
        return 0;
}
```

OUTPUT :

ERROR MESSAGE

‘dx’;is not a member of demo

VIRTUAL DESTRUCTOR

There is no concept of virtual constructor in C++ but a virtual destructor can be in C++. In the normal call sequence of constructor and destructor, they follows the basic rules which have been discussed earlier i.e., constructor of base class is called first and destructor of derived class calls first even the destructor in the base class is virtual. See the program given below for better understanding point of view.

```
#include <iostream>
using namespace std;
class first
{
public :
    first( )
{
cout<<"First constructor called"<<endl;
}
 ~first( )
{
cout<<"First destructor called"<<endl;
}
};
class second :public first
{
char *ptr;
   public :
second( )
{
        ptr=new char[10];
        cout<<"Second constructor called"<<endl;
}
```

```
~second( )
{
        cout<<"Second destructor called"<<endl;
        delete ptr;
}
};
int main( )
{
        second s;
        return 0;
}
```

OUTPUT :

```
First constructor called
Second constructor called
Second destructor called
First destructor called
```

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

class first
{
public :
~first( )
{
        cout<<"first destructor called"<<endl;
}
};

class second :public first
{
        char *ptr;
public :
```

```
 ~second( )
{
cout<<"Second destructor called"<<endl;
}
};
int main( )
{
        first *f=new second;
        delete f;
        return 0;
}
```

OUTPUT :

first destructor called

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

class first
{
public :
virtual ~first( )
{
        cout<<"first destructor called"<<endl;
}
};

class second :public first
{
        char *ptr;
public :
 ~second( )
{
cout<<"Second destructor called"<<endl;
```

```
}
};
int main( )
{
        first *f=new second;
        delete f;
        return 0;
}
```

OUTPUT :

Second destructor called

First destructor called

Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class

```
#include <iostream>
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
};
class Dog: public Animal
{
 public:  virtual void eat()
   {
     cout<<"Eating bread...";
   }
```

```
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
```

Output

Eating bread...

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

class Base
{
public:
   int _base = 2;
    virtual void display()
   {
      cout << "The value of _base is " << _base << endl;
   }
};

class Derived : public Base
{
public:
   int _derived = 4;
   void display()
   {
      cout << "The value of _base is " << _base << endl;
      cout << "The value of _derived is " << _derived << endl;
   }
};
```

```
int main()
{
    Base *p_b, b;
    Derived d;

    p_b = &d;
    p_b->display();

    return 0;
}
```

VIRTUAL BASE CLASSES

This can be done by making the base class a virtual class. This keyword makes the two classes share a single copy of their base class .

```
class base
{
:
:
};
class Aclass : virtual public base
{
:
:
};
class Bclass : virtual public base
{
:
:
};
class derived : public Aclass, public Bclass
```

```
{
:

};
```

This will resolve the ambiguity involved.

ABSTRACT CLASSES

Abstract classes are the classes, which are written just to act as base classes. Consider the following classes.

```
class base
{
:
};
class Aclass : public base
{
:
:
};
class Bclass : public base
{
:
};
class Cclass : public base
{
:
:
};
void main()
{
Aclass objA;
Bclass objB;
```

```
Cclass objC;
:
:
}
```

There are three classes – Aclass, Bclass, Cclass – each of which is derived from the class base. The main () function declares three objects of each of these three classes. However, it does not declare any object of the class base. This class is a general class whose sole purpose is to serve as a base class for the other three. Classes used only for the purpose of deriving other classes from them are called as abstract classes. They simply serve as base class , and no objects for such classes are created.

VIRTUAL FUNCTIONS

When we use the same function name in both the base and derived classes, the function in the base class is declared as virtual using the keyword virtual preceding its normal declaration. When a function is made virtual, C++ determines which function to use at runtime based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function.

```
class Shape
{
public :
virtual void print()
{
cout << " I am a Shape " << endl;
}
};
class Triangle : public Shape
{
```

```
public :
void print()
{
cout << " I am a Triangle " << endl;
}
};


class Circle : public Shape
{
public :
void print()
{
cout << " I am a Circle " << endl;
}
};
int main()
{
Shape S;
Triangle T;
Circle C;
S.print();
T.print();
C.print();
Shape *ptr;
ptr = &S;
ptr -> print();
ptr = &T;
ptr -> print();
ptr = &C;
ptr -> print();
return 0;
}
```

Output

I am a Shape

I am a Triangle

I am a Circle

I am a Shape

I am a Triangle

I am a Circle

Now, the output of the derived classes are invoked correctly. When declared with the keyword virtual, the compiler selects the function to be invoked, based upon the contents of the pointer and not the type of the pointer. This facility can be very effectively used when many such classes are derived from one base class. Member functions of each of these can be, then, invoked using a pointer to the base class .

```
#include<iostream.h>
class Base
{
public:
void display()
{
cout<<”Display Base”;
}
virtual void show()
{
cout<<”Show Base”;
}
};
class Derived : public Base
{
public:
void display()
{
```

```
cout<<”Display Derived”;
}
void show()
{
cout<<”show derived”;
}
};
int main()
{
Base b;
Derived d;
Base *ptr;
cout<<”ptr points to Base”;
ptr=&b;
ptr->display();          //calls Base
ptr->show(); //calls Base
cout<<”ptr points to derive
ptr=&d;
ptr->display();          //calls Base
ptr->show();             //class Derived
}
```

Output:

```
ptr points to Base
Display Base
Show Base
ptr points to Derived
Display Base
Show Derived
```

When ptr is made to point to the object d, the statement ptr->display(); calls only the function associated with the Base i.e.. Base::display() where as the statement, ptr->show(); calls the derived version of show(). This is because the function display() has not been made virtual in the Base class.

```
Class Animal
{
        public|:
                void food()
                {
                        cout<<”It eats generic food”;
                }
};
Class Cat : public Animal
{
        public|:
                void food()
                {
                        cout<<”It eats rat”;
                }
};

Animal *animal=new Animal
Cat *cat=new Cat;

animal->food();
cat->food();
```

Rules For Virtual Functions:

When virtual functions are created for imnecessarily redefined in the derived class. In such cases, calls will invoke the base function.

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.

6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer points to any type of the derived object, the reverse is not true. i.e. we cannot use a pointer to a derived class to access an object of the base class type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

PURE VIRTUAL FUNCTIONS

As discussed earlier, an abstract class is one, which is used just for deriving some other classes. No object of this class is declared and used in the program. Similarly, there are pure virtual functions which themselves won't be used. Consider the example:

```
class Shape
{
public :
virtual void print() = 0; // Pure virtual function
};
class Triangle : public Shape
{
public :
void print()
{
cout << " I am a Triangle " << endl;
```

```
}
};
class Circle : public Shape
{
public :
void print()
{
cout << " I am a Circle " << endl;
}
};
void main()
{
Shape S;
Triangle T;
Circle C;
Shape *ptr;
ptr = &T;
ptr -> print();
ptr = &C;
ptr -> print();
}
```

Output:

I am a Triangle

I am a Circle

It can be seen from the above example that , the print() function from the base class is not invoked at all . even though the function is not necessary, it cannot be avoided, because, the pointer of the class Shape must point to its members. Object oriented programming has altered the program design process. Exciting OOP concepts like polymorphism have given a big boost to all this. Inheritance has further enhanced the language. This session has covered some of the finer aspects of inheritance.

```
// virtual members
#include <iostream.h>
class CPolygon
{
protected:
int width, height;
public:
void set_values (int a, int b)
{
width=a; height=b;
}
virtual int area (void)
{
return (0);
}
};
class CRectangle: public CPolygon
{
public:
int area (void)
{
return (width * height);
}
};
class CTriangle: public CPolygon
{
public:
int area (void)
{
return (width * height / 2);
}
};
int main ()
{
```

```
CRectangle rect;
CTriangle trgl;
CPolygon poly;
CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
CPolygon * ppoly3 = &poly;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly3->set_values (4,5);
cout << ppoly1->area() << endl;
cout << ppoly2->area() << endl;
cout << ppoly3->area() << endl;
return 0;
}
```

Output:

```
20
10
0
```

Now the three classes (CPolygon, CRectangle and CTriangle) have the same members: width, height, set_values() and area(). area() has been defined as virtual because it is later redefined in derived classes.

If you remove this word (virtual) from the code and then you execute the program the result will be 0 for the three polygons instead of 20,10,0. That is because instead of calling the corresponding area() function for each object (CRectangle::area(), CTriangle::area() and CPolygon::area(), respectively), CPolygon::area() will be called for all of them since the calls are via a pointer to Cpolygon.

Therefore, what the word virtual does is to allow that a member of a derived class with the same name as one in the base class be suitably called when a pointer to it is used, as in the above example. Note that in spite of its virtuality we have also been able to declare an object of type CPolygon and to call its area() function, that always returns 0 as the result.