5. Polymorphism

Introduction:

- Polymorphism is the ability to use an operator or function in different ways. It gives different meaning or function to the operators or functions. Poly refers too many, signifies the many uses of the operators and functions. A single function uses or an operator functioning in many ways can be called polymorphism. It refers to codes, in different context. An operator having same name, suppose '+' operator which can add two integers as well as concatenates two strings.
- The information about the same name and different forms or operation is known at the compile time so compiler is able to select the appropriate function for a particular call at compile time. This is called early binding or static binding. The compile time polymorphism means that an object is bound to its function call at the compile time.

...

- It would be nice if the appropriate member function could be selected while the program is running. This is known as run time polymorphism. C++ supports a mechanism known as virtual function to achieve runtime polymorphism. At run time, when it is known what class object are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as late binding. It is also known as dynamic binding because the selection of the appropriate function is done dynamically at run time which requires the use of pointer to objects.

Static Vs Dynamics Polymorphism

- Static polymorphism is concerned more efficient, while dynamic polymorphism is more flexible.

...

- In dynamic polymorphism, the function calls to its various forms are resolved dynamically when the program is executed. As we already know, the term late binding refers to the resolution of the function at runtime instead of at compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context.
- Statically bound methods are methods that are bound to their calls at compile time. Dynamic function calls are bound to the functions during run time. On the other hand, no run time search is required for statically bound functions.
- In the case of dynamic binding, the function calls are resolved at run time, thereby providing users with the flexibility to alter a call without having to modify the code. To a programmer, efficiency and performance would be probably be a primary concern but to a user, flexibility and maintainability may be much more important.

...

Advantages of Polymorphism

i) Once the application is written using the concept of polymorphism, it can be easily extended, provided new objects that confirms to the original interfaces.

ii) It helps in reusability of code

iii) Provides easier maintenance of application.

Types of polymorphism

1) Operator Overloading

- As we already mention, same operator can perform different task or form, so operator overloading means giving additional meaning of operators when they apply to user defined data types. The concept of operator overloading is not only applicable for user defined data type but also used for normal application.

...

- Some examples are: * operator is used to multiply two numbers as well as it is used to assign the pointer variable, + operator is used to add two number as well as it is used to concatenate two strings. By overloading operators, its original meaning will not be lost.
- To define an additional task to an operator, we have to define the relation of operator with class and OOP provides a special function called operator function and the operator function is defined as follows:

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

- Where the return-type is the type of the value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator.

...

- Operator function must be either member function or friend function. The basic difference between them is that a friend function will have only one argument list in unary operator and two for binary operator, while a member function has no argument for unary operator and only one for binary operator. This is because the object used to invoke member function is passed implicitly and therefore is available for the member function. This is not in the case of friend function. Argument may be passed either by value or by reference.
- The process of overloading involves the following steps:

i) Create class that defines the data types that is to be used in overloading operation.

ii) Declare the operator function operator op() in the public part of the class. It may be either member function or a friend function.

iii) Define the operator function to implement the required operation.

...

```
//Overloading Unary (minus) Operator
#include<iostream.h>
#include<conio.h>
class unary
{int x,y;
public:
void getdata(int a, int b)
{x=a; y=b;}
void display()
{cout<<"x="<<x<<endl;
cout<<"y="<<y<<endl;}
void operator-();};
void unary :: operator-()
{x=-x; y=-y;}
```

```
void main()
{
unary u;
u.getdata(10,-20);
u.display();
-u;
cout<<"After overloading –
    operator:"<<endl;
u.display();
getch();
}
```

Output:

x=10
y=-20

After overloading – operator:
x=-10
y=20

...

In the above program, - operator is overloaded: used to make negative value of built in data types of x and y and same – operator is also applied to user defined data type 'u' which calls the function operator-().

```
/*add 2 complex numbers by
    using operator overloading */
#include<iostream.h>
#include<conio.h>
class complex
{float x,y;
public:
complex(){}
complex(float r, float i)
{x=r; y=i;}
complex operator +(complex );
void display();
};
complex complex ::
    operator+(complex c)
{
complex temp;
temp.x=x+c.x;
temp.y=y+c.y;
return(temp);}
```


...

```
void complex:: display()
{cout<<x<<"+j"<<y<<endl;}
void main()
{
complex c1,c2,c3;
c1=complex(4.1,3.5);
c2=complex(1.6,2.3);
c3=c1+c2;
cout<<"c1=";
c1.display();
cout<<"\nc2=";
c2.display();
cout<<"\nc3=";
c3.display(); getch(); }
```

Output:

c1=4.1+j3.5
c2=1.6+j2.3
c3=5.7+j5.8

- In the above example,

+ operator is overloaded.

+ Operator is applying to add defined variables c1 and c2. The data member of c1 is directly accessed and c2 is accessed by dot operator.

...

Type Casting

- If an expression contains different types of data like integer and float in left and right side then the compiler automatically convert from one type to another type by applying the type conversion rule provided by the compiler. The automatic type conversion means that the right side of the data type is automatically converted to the left side type. This type of conversion rules cannot be applied in case of user defined data types. In user defined data types, we must define conversion routine ourselves. The possible types are:
  i) Conversion from basic type to class type
  ii) Conversion from class type to basic type
  iii) Conversion from one class type to another class type

...

Conversion between user defined (class type) and basic types

- To convert from basic type (int, float, double etc.) to class type constructor is used
- To convert from class type to basic type, C++ allows us an overloaded casting operator. The general form of an overloaded casting operator function is usually referred to as a conversion function, is:

```
operator typename()
{ ..........
..........//(function statement)
.......... }
```

- This function converts class type data to typename. The casting operator function should satisfy the following conditions:

 It must be a class member.
 It must not specify a return type. It must not have any arguments.

...

```
//basic to class type conversion
#include<iostream>
#include<conio.h>
class time
{
int hrs, mins;
public:
time()
{hrs=0;
mins=0;}
time (int t)
{hrs=t/60;
mins=t%60;}
void display()
{
cout<<"Hour: "<<hrs;
cout<<" Minutes: "<<mins;
} };
void main()
{
time t1;
int duration=85;
t1=duration;    //int to class type
t1.display();
getch();}
```

Output:
Hour: 1 Minutes: 25

…

```
//class to basic type conversion
#include<iostream>
#include<conio.h>
class time
{int hrs, mins;
public:
time (int a,int b)
{hrs=a;
mins=b;}
operator int()
{int m;
m=hrs*60+mins;
return m;
} };
```

```
void main()
{
time t1(1,25);
int minutes=t1;   //class type to int
cout<<"Minutes: "<<minutes;
getch();}
```

Output:

Minutes: 85

…

- Another example which converts distance in meter to feet and inches and feet and inches to meter. Two types of conversion techniques are applied which converts basic type to class type (user defined) and class type to basic type.
- 1m = 3.28084 ft
- 1ft = 12 inch

```
//Example basic to class type
#include<iostream.h>
#include<conio.h>
const float m=3.28084;
```

```
class distance
{
int feet;
float inches;
public:
distance()
{feet=0;
inches=0;}
distance(float meter)//basic to class type
{
float ft=m*meter;
feet=(int)ft;
inches=12 *( ft-feet);
}
```


…

```
distance(int f, float i)
{feet=f;
inches=i;}
void showdistance()
{
cout<<feet<<"' "<<inches<<"
   " "<<endl;
}
operator float()//class to basic type
{
float fractfeet= inches/12;
fractfeet += (float)feet;
return(fractfeet/m);}
};
```

```
void main()
{
distance d(2.35);
cout<<"Distance 1 = ";
d.showdistance();
distance d2(10,10.25);
float mtrs=(float) d2;
cout<<"Distance 2
   ="<<mtrs<<"m"<<endl;
getch();}
```

- **Output:**

Distance 1 = 7' 8.55197"

Distance 2 = 3.30835m

…

Conversion from one user defined types to another

- We know that the casting operator function operator typename() converts the class of which it is a member to typename. The typename may be built-in type or a user-defined one (another class type). In the case of conversions between objects, typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used (i.e. source class). The conversion takes place in the source class and the result is given to the destination class object. However constructor can also be used to convert from one user defined types to another, for this constructor is used in destination class that takes object (of source class) as an argument
- Example: Define two classes named Rectangle and Polar to represent the point in polar and rectangle form. Use conversion routine to convert from one form to another.

…

```
//using operator function
#include<iostream.h>
#include<conio.h>
#include<math.h>
class Rectangle
{float xw, yw;
public:
Rectangle(){}
Rectangle(float x, float y)
{xw=x;
yw=y;}
void display1()
{cout<<"("<<xw<<","<<yw<<
   ")"<<endl;}};
```

```
class Polar
{
float radius,angle;
public:
Polar(){}
Polar(float a, float i)
{
radius=a;
angle=i;
}
void display()
{
cout<<"("<<radius<<","<<angl
   e<<")"<<endl;}
```


…

```
operator Rectangle()
{
float x=radius * cos(angle);
float y=radius * sin(angle);
return Rectangle(x,y);}};
void main()
{Rectangle rec;
Polar pol(10.0,0.78);
rec=pol;
cout<<"Pol"<<endl;
pol.display();
cout<<"Rec"<<endl;
rec.display1();
getch();}
```

Output:

Pol

(10, 0.78)

Rec

(7.10914, 7.03279)

…

```
//using constructor
#include<iostream.h>
#include<conio.h>
#include<math.h>
class Polar //source class
{ public:
float radius,angle;
public:
Polar(){}
Polar(float a, float i)
{radius=a; angle=i;}
void display()
{cout<<"("<<radius<<","<<angle<<")"<<endl;}};
class Rectangle //destination class
{float xw, yw;
public:
Rectangle(){xw=0; yw=0;}
Rectangle(Polar p) //constructor for conversion
{
xw=p.radius * cos(p.angle);
yw=p.radius * sin(p.angle);}
void display1()
{cout<<"("<<xw<<","<<yw<<")"<<endl;}
```


…

```
};
void main()
{Rectangle rec;
Polar pol(10.0,0.78);
rec=pol;
cout<<"Pol"<<endl;
pol.display();
cout<<"Rec"<<endl;
rec.display1(); getch();}
```

Output:

Pol

(10, 0.78)

Rec

(7.10914, 7.03279)

…

2) Function Overloading

- It is defined as the features of OOP in which a symbol or a function have same name but with different forms. Mainly used overloading is operator overloading and function overloading. In operator overloading same operator can be used for different purpose and in function and constructor overloading, same function name can perform different operation or calculation.

```
#include<iostream.h>
#include<conio.h>
void print(int i)
{cout<<"Here is int"<<i<<endl;}
void print(double f)
{cout<<"Here is float"<<f<<endl;}
void print(char a[20])
{cout<<"Here is character"<<a<<endl;}
void main()
{
print(10);
print(10.01);
print("PU");
getch();}
```

Output:

Here is int 10

Here is float 10.01

Here is character PU

…

Overriding

- If a method is defined in a base class for particular message then that is inherited by the derived class and the subclass defines another method with the same name then the instance of super class is replaced. So the process of declaration of method with same name in the super and derived class which replace the instance of super class is called overriding. The method overriding contributes to shape the code. if the method defined in super class is not appropriate for the derived class and should slightly change, at that time we can override the method. Without overriding, it would be necessary for all the subclasses to provide their own method to respond to message.

…

```
#include<iostream.h>
#include<conio.h>
class A
{protected:
int a;
public:
void geta(int x)
{a=x;}
void display()
{cout<<"a="<<a<<endl;}};
class B : public A
{
int a,b;
public:
void getb(int x,int y)
{a=x; b=y;}
void display()
{
cout<<"a="<<a<<" and "<<"b="<<b<<endl;
}};
void main()
{
B b;
b.geta(5);
b.getb(10,20);
b.display();
b.A::display();
getch();}
```

Output:

a=10 and b=20

a=5

…

Pointer to Object

- Pointer can use to point an object
- Let us take: item x;

Here, x is an object of class item, using pointer this can be declare as: **item *x;**

Here x is a **pointer object** and pointer object is useful for create run time object

- An arrow (->) operator is used instead of dot (.) operator for accessing class member in case of pointer object

```
#include<iostream>
class item
{int no; float price;
public:
void getdata(int a, float b)
{no=a; price=b;}
void display()
{cout<<no<<":"<<price<<endl;}};
void main()
{item x;
item *ptr;
ptr=&x;
ptr->getdata(101,50.5);
ptr->display();}
```

…

Polymorphic Variables

- Polymorphism is not only possible through overloading (operator and function) but also possible through polymorphic variables. Polymorphic variables are those variables which have same name but can hold different types of values. For example, a variable can hold integer as well as float values. It also satisfies principle of substitutability. The polymorphic variables are used in subtype. In C++ polymorphic variables and the class which use the declared variable is different. In C++ polymorphic variables are possible to use only by using pointer or references.

Deferred Method

- The deferred methods are members of a super class which is null in the super class and redefined in the child class whenever it is used. This method is also called pure virtual function. The deferred methods are declared in abstract class which is not used to create the object.

146

…

- If a common method is declared and defined in base class which has several derived classes without deferred method then it is more difficult to collect all the information related to that common method. To remove this difficulty, the deferred method is used.
- Suppose we have a collection of classes to draw circle, rectangle, square which have a base class called shape. If we declare a method which is inherited in all child classes as deferred method then new child class can define this method in its own way.

3) Virtual Function

- The object of different classes can respond to same message in different forms. We can access the function by using the object which is declared as single pointer variable. To access the member of a class having same name, C++ provides pointer object of base class which can independently access the members. The pointer to base class refers to all the object of derived class. While choosing member function, pointer object ignore all the contents and match with members and access.

147

…

- The function which is accessed by pointer object having same name in both base class and derived class is called virtual function. The function is declared using keyword 'virtual'. When a function is made virtual, C++ determines which function to used at runtime based on the type of object pointed on the base pointer.
- The vital reason for having a virtual function is to implement a different functionality in the derived class.
- Syntax:

```
class classname
{
public:
virtual void memberfunction()
{
...........
........... }};
```

148

…

Properties of Virtual Function

a) Dynamic Binding Property

- Virtual functions are resolved during runtime or dynamic binding. The main difference between non-virtual member function and virtual member function is in the way they are both solved. A non-virtual member function is resolved during compile time (static binding) & virtual functions are resolved during run-time (dynamic binding).

b) Virtual functions are member function of a class

c) Virtual functions are declared using the keyword **virtual**

d) Virtual function takes a different functionality in a derived class.

149

…

```
#include<iostream.h>
#include<conio.h>
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"<<endl;}
void show()
{cout<<"Show Derived"<<endl;}};
```

```
void main()
{
base B;
derived D;
base *bptr;
cout<<"bptr points to Base"<<endl;
bptr=&B;
bptr->display(); //calls BaseVersion
bptr->show(); //calls Base Version
cout<<"bptr points to Derived"<<endl;
bptr = &D;
bptr ->display(); //calls Base Version
bptr -> show(); //calls Derived version
getch();}
```

Output:

```
bptr points to Base
Display base
Show base
bptr points to Derived
Display base
Show Derived
```

150

…

Note: When bptr is made to point to the object D, the statement

bptr -> display()

calls only the function associated with the Base (i.e. Base :: display()), whereas the statement

bptr -> show();

calls the Derived version of show(). This is because the function display() has not been made virtual in the Base class.

- **Virtual destructor**: Destructor can also be declared as virtual in parent class if so, then both parent as well as child destructors will be executed otherwise only the parent destructor will be invoked.

Pure Polymorphism (Polymorphic method)

- In case of C++, same function can perform different task and may have different parameters. This situation is called overloading and the overloading feature is fall into polymorphism. However the situation where a single function can be executed by arguments of a variety of types is called polymorphic function(pure polymorphism).

151

…

pure virtual function: A do-nothing function may be defined as virtual void display()=0. Such function is called **pure virtual function** (also called deferred method). *A pure virtual function is a virtual function without a body, created by a value 0 being assigned to the function declared in a base class*. In such cases, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function. Pure virtual function cannot be used to declare any objects of its own. The main objective of abstract base class is to provide some facilities to the derived class and to create a base pointer required for achieving run time polymorphism.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class media
{protected:
char title[50]; float price;
public:
media(char *s, float a)
{
strcpy(title,s);
price=a;
}
```

152

…

```
virtual void display()=0;
//pure virtual function
};
class book: public media
{protected:
int pages;
public:
book(char *s,float a, int p):media(s,a)
{pages=p;}
void display()
{cout<<"Title:"<<title<<endl;
cout<<"Pages:"<<pages<<endl;
cout<<"Price:"<<price<<endl;}};
class tape: public media
{protected: float time;
public:
tape(char *s,float a, float
    t):media(s,a)
{time=t;}
void display()
{cout<<"Title:"<<title<<endl;
cout<<"Play Time:
    "<<time<<endl;
cout<<"Price:"<<price<<endl;
}
};
```

153

…

```
int main()
{media *mptr;
book b("C++",34.5,123);
tape ta("Jerry",45.7,41.5);
mptr=&b;
mptr->display();
mptr=&ta;
mptr->display();
getch();
return 0;}
```

154

Miscellaneous…

Static Data Member:

- If a data item in a class is defined as static, then only one such item is created for the entire class, no matter how many objects are there. A static data item is useful when all objects of the same class must share a common item of information. A member variable defined as static has similar characteristics to a normal variable. It is visible within the class, but its lifetime is the entire program. A static member variable has certain special characteristic, these are

a) It is initialized to zero when the first object of its class is created.

b) Only one copy of that member is created for the entire class and shared by all the object of that class. No matter how many object are created.

c) It is visible only within class, but its life time is the entire program

155

…

```
//Example static member
# include <iostream>
class item
{static int count;
int number;
public:
void inccount(int a)
{
number=a;
count++;
}
void getcount()
{cout<<"Count:"<<count<<endl;
}};
int item::count;
void main()
{
item I1,I2;
I1.getcount();
I2.getcount();
I1.inccount(100);
I2.inccount(200);
cout<<"After reading data:\n";
I1.getcount();
I2.getcount();
cin.get();}
```

Output:
Count:0
Count:0
After reading data:
Count:2
Count:2

156

…

Static Member Functions:

A member function declared **static** has following characteristics:

- Can have access to only other static members (functions/ variables) declared in the same class
- Can be called using class name (instead of object) as:

cname::fname;

```
//Example static member
# include <iostream.h>
class test
{int code;
static int count;
public:
void setcode(void)
{
code=++count;
}
void showcode(void)
{
cout<<"Object
    number:"<<code<<endl;}
static void showcount(void)
{cout<<"\nCount:
    "<<count<<endl;}};
int test::count;
```

157

…

```
void main()
{test t1,t2;
t1.setcode();
t2.setcode();
test::showcount();
test t3;
t3.setcode();
test::showcount();
t1.showcode();
t2.showcode();
t3.showcode();
cin.get();}
```

Output:

Count: 2
Count: 3
Object number: 1
Object number: 2
Object number: 3

Default argument

- There is a feature of C++ that allows us to call a function without specifying all its arguments . In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared as:

  float amount (float principal, int period, float rate=0.15);

  – If the values are passed as:

  amount(5000,7);

  assign 5000 to principal, 7 to period default value 0.15 to missing argument rate

  – Otherwise assign a given value

  amount(5000,7,0.12);

  Passes 5000 to principal, 7 to period default value 0.12 to rate
- Default argument is useful in situation where some argument always have the same value such as bank interest.

…

```
//Example default argument
#include <iostream>
class myclass
{
int x;
public:
myclass(int n = 0)
{ x = n; }
int getx( )
{ return x; }
};
```

```
void main( )
{
myclass O1(10);
myclass O2;
 cout << "O1: " << O1.getx( ) << "\n";
 cout << "O2: " << O2.getx( ) << "\n";
 cin.get(); }
```

Output:

O1 = 10
O2 = 0

Reference Variable

- C++ introduces a new kind of variable called reference variable
- Provide alternative name for previously defined variable and can be used interchangeably to represent that variable
- Syntax for creating reference variable:

d_type & r_name=v_name;

Eg: int total=100;

int & sum=total;

- Here, total is a already declared variable and sum is the alternative name to represent the variable total ie. Reference to the variable total.
- Both variables refer to the same data object in the memory

```
//Example reference variable
#include <iostream>
void main( )
{
int total=100;
int & sum=total;
total=total+10;
cout << "Total= " << total << "\n";
cout << "Sum= " << sum << "\n";
 cin.get(); }
```

- **Output:**

Total= 110
Sum= 110

this pointer

- C++ contains a special pointer that is called **this** to represent an object that invokes a member function. **this** is a pointer that is automatically passed to any member function when it is called, and it points to the object that generates the call. For example, this statement,

ob.f1( ); // assume that ob is an object

- the function f1( ) is automatically passed as a pointer to ob, which is the object that invokes the call. This pointer is referred to as this. It is important to understand that only member functions are passed a this pointer. For example a friend does not have a this pointer.
- The this pointer has several uses, including aiding in overloading operators .
- By default, all member functions are automatically passed a pointer to the invoking object. Let us see example:

…

```
// Demonstrate the this pointer
#include<iostream>
#include<string.h>
class inventory
{char item[20];
double cost;
int on_hand;
public:
inventory(char *i, double c, int o)
{//access members through the this pointer
strcpy(this->item, i);
this->cost = c;
this->on_hand = o;   }
void show( ); };
```

```
void inventory::show( )
{cout << this->item; //use this to access member
cout << ": " << this->cost;
cout << "\tOn hand: " << this->on_hand;}
int main( )
{inventory inv("Cream",23.45,12);
inv.show(); cin.get(); }
```

- Here the member variables are accessed explicitly through the this pointer. Thus, within show( ), these two statements are equivalent:

cost = 123.23; this->cost = 123.23;

In fact the first form is a shorthand for the second. Though the second form is usually not used for such simple case, it helps understand what the shorthand implies.

Assignments

19. Differentiate early binding and late binding with suitable example.
20. What is polymorphism? Explain run-time and compile time polymorphism.
21. Define the terms deferred methods, abstract class, virtual destructor and function overloading.
22. Create an abstract base class shape with two members base and height, a member function for initialization and a pure virtual function to compute area ( ). Derive two specific classes Triangle and Rectangle which override the function area ( ). Use these classes in a main function and display the area of a triangle and a rectangle.
23. What is operator overloading? How can you use operator overloading in C++? Give the syntax.
24. What is type conversion? How a class type can be converted into another class type explain with example.
25. What is a virtual function? When do we make a function virtual and when we make a function pure virtual? Explain with a suitable example.
26. Explain default argument with an example.