

FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

Lecture16

Lecture 16: Polymorphism

Welcome to lecture 16, Polymorphism. So, during the basic introduction of polymorphism, we saw what you mean by polymorphism. Poly, which is nothing but many, and morph is nothing but forms. So, the entity will take multiple forms, right?

So, for example, a function, the member function, or even a function that takes many forms. So, in C++ and Java, you have the polymorphism, right? The property of polymorphism means the properties of one class to be used in different classes in various ways. So, you call this polymorphism. For example, I have, let us say, a function, int, right?

So, maximum of int A, int B. Assume that I am using this function, right? So, similarly, let us assume I have the same name. Right, assume that I have the same name, but here the parameters or the arguments are float A and float B, right? So, in that case, in the main program, what exactly are you doing? Assume that I have written these two functions. So, let us assume I have properly defined. The dot dot dot indicates that I have properly defined.

So, in that case Which function will be called? For example, if I am passing, let us say, max of 2, 5 and I am storing this in some integer I, right? So, 2 and 5, since it is an integer. So, by default, right, you have passed an integer.

Polymorphism

- Poly means 'many,' and morph means 'forms.'
- Polymorphism refers to a single entity taking on multiple forms.
- In C++ and Java, polymorphism allows the properties of one class to be used in different classes in various ways.

```
int maximum(int a, int b) { ... }  
float maximum(float a, float b) { ... }
```



So, this function will be called, right? So, I put 1. So, 1 will be called, and the resultant will be stored in I. Suppose I am passing, let us say, max of 2.5, 3.4, right? So, in that case, this function will be called, right? And whichever, right, whichever is the result, that will be stored in, let us say, float m, something like that, right?

So, that means, from this example, the entities can take multiple forms. So, here it is the member function. So, member function, single entity means they are having the same name. So, the same name which is taking multiple forms, right? One is you are finding out the maximum of two integers, or let us say, you have a maximum of two characters based on the ASCII value, something will be printed, right?

So, which one will be called? So, you call it polymorphism. So, this is in the case of, let us say, procedural-oriented language, right? So, in C++ and Java, since we have already seen the classes and also about the inheritance, Suppose class A has the function.

Assume that it is the same function. In fact, in the case of inheritance, we are going to call the concept overriding. We can have the same entities. The name of the function and arguments are also the same. Let us say int A, int B. The return type is also the same.

But the question is, which function will be called? So, which member function will be called? So now, there are two cases. So, one I talked about is the procedural-oriented way. The procedural-oriented language way.

Another one is, let us say, object-oriented. C++ or Java. So, in this case, the first one is the compile-time polymorphism. The previous example that we had seen. So, assume that, without a class, we are writing this.

I have written two functions of the same name. One is this, and the second one is this. So, this one, when I am calling this in the main, right? When I am calling this in the main, which one will be called? So, this will be decided during compile time, right?

So, which function will be called? So, this will be decided during compile time. So, therefore, this is called the compile-time polymorphism, right? Or you call it as static binding or static polymorphism. Whereas, when we talk about the inheritance,

So, in the base class, you have the same function name, right. Let us assume the name of the function is int FUN1, and then you are passing something. Let us say integer, comma, integer, right. So, this is in the base class, and in the derived class, you also have the same name, right. So, assume that I am defining this int FUN1.

In comma int. So, in the derived class, you also have the same name. Right. So, in that case, which function will be called? Assume that the functionalities are different.

Right. In Function 1, I am putting cout. Welcome to IIT Roorkee. Right. And then here, in the derived class, I am writing cout.

Welcome to IIT Madras. So, in that case, assume that the object is created under, let us say, the derived class. Right. So, which function will be called? So, this will be decided during runtime.

Types of Polymorphism

There are two types of polymorphism in C++

1. Static/Compile-time polymorphism.
2. Dynamic/Run-time polymorphism.


Base

`int fun1(int, int);`

Derived

`int fun1(int, int);`

↓



So, therefore, this polymorphism is called runtime polymorphism. So, we have already seen inheritors, the base class, derived class, and assume that the derived class is inheriting the base class. But the problem is you have the same function name. So, in fact, one of the programs we had seen before, right? So, this will be decided during runtime.


So, you call it dynamic binding or runtime polymorphism. So, as I said, In the case of compile-time polymorphism, in C++, we can achieve this kind of polymorphism by doing function overloading. So, you can do function overloading. So, what do you mean by overloading?

So, here, as I said, the number of arguments, right? So, the same maximum program, return type is integer, right? The same maximum. Suppose I have int a, int b, int c, right? We have already used what?

Static/Compile-time Polymorphism

- This type of polymorphism in C++ can be achieved by function Overloading.
- Overloading by changing number of arguments or type of arguments is called compile-time polymorphism.

`int max;`



Int, let us say maximum, int a, int b, right? So one is the maximum of two numbers. Another logic we are going to write is the maximum of three numbers,

right? So now your number of arguments, here it is three. And the number of arguments here is two.

So in the main program, when you are calling, let us say, the maximum of, let us say, 2, 3. Right. When you are calling the maximum of 2, 3. So what will happen? This function will be called.

Right. So maximum of 2, 3. Two arguments. Two values you are passing. So that means this function will be called.

Right. Suppose I am doing, say, max, let us say 20, 30, 40. Right. Three values I am passing. So in that case, this will be called.

Alright. So, there is a change in the number of arguments. So, during compile time, it will be decided which one will be called. Or, like a previous example, we had given the type. Alright.

So, the type we had given is float A, float B. And the return type is also float. Alright. In that case, it is completely different from the one you had written here. Let us say 1, 2, and the third one is float. The return type is float.

Same maximum function. But inside the argument, you have float A, float B. Correct? So in that case, assume that you are calling the maximum of 2.0, 3.0, or 20.0, 30.0. Both are floats. Right?

So in this case, you have different types of arguments, either the number of arguments or different types of arguments, the third one, right? So in that case, whatever you are calling in the main, correct? So which will be decided during compile time. Therefore, this is called compile-time polymorphism or static polymorphism or static binding. All are the same names. So let us see how function overloading is working in C++, right? So let us consider the area. The name of the function is area. So you have one argument, radius, and area returns πr^2 . 3.14 into radius into radius.

Another function return type is double. I have an area, but there are two arguments. One is length and width. One is length, and the other is width. Length and width.


So now you are returning length times width. Two arguments. Now the third one has the same return type, double area function name is the same. So you look

here; the function names are the same, right? So here you have double base height and triangle, right?

So now it is returning 0.5 times base times height, right? 0.5 times base times height. So now you have right: some radius 5.0, length 10.0, and width is 4.0. Coming to the main program, base is 6.0 and height is 3.0. Now I am calling the function area. And then what am I doing?

```
1  #include <iostream>
2  using namespace std;
3
4  // Overloaded function to calculate the area of a circle
5  double area(double radius) {
6      return 3.14159265358979323846 * radius * radius;
7  }
8
9  // Overloaded function to calculate the area of a rectangle
10 double area(double length, double width) {
11     return length * width;
12 }
13
14 // Overloaded function to calculate the area of a triangle
15 double area(double base, double height, bool isTriangle) {
16     return 0.5 * base * height;
17 }
18
```

Function Overloading



I am passing one value: radius. So when I am passing one value, radius, only one value. So, it is a one-argument function. So that means the radius will be passed over here. Assume that the radius is what?

5.0. So, 3.14 multiplied by 5.0 multiplied by 5.0 will be calculated. And that will be displayed. Okay. Because this is a one-argument function.

And in the second case, it is a two-argument function. So, length and width are being passed. So, the function name is the same. So, this particular function will be called, line number 10, right? You are passing length and width, correct?

So, then, length into width: what is length into width? 10.0 and 4.0, 10 into 4, 40, right? So, in this case, 40 will be displayed, the second case. And the third case, Same function name.

But you are passing three parameters. Base, height, and true value. Boolean. Right. This is what exactly.

```

19 int main() {
20     double radius = 5.0;
21     double length = 10.0, width = 4.0;
22     double base = 6.0, height = 3.0;
23
24     // Call the overloaded functions for different shapes
25     cout << "Area of the circle: " << area(radius) << endl;
26     cout << "Area of the rectangle: " << area(length, width) << endl;
27     cout << "Area of the triangle: " << area(base, height, true) << endl;
28
29     return 0;
30 }
31

```



This triangle is Boolean. And here you go. It is true. So, you are passing base and height. Correct.

So, it will be multiplied by half. 0.5 multiplied by base multiplied by height. Whatever you are passing. Right. So, three arguments you are passing.

Three values you are passing. It is a three-argument function. So, three parameters, and then the third function will be called, right? So, these things will be decided during compile time. So, therefore, it is called static binding or compile-time polymorphism.

The name polymorphism because if you look at line numbers 25, 26, and 27, all the function names are same, right? So, therefore, this is called function overloading, or you call it compile-time polymorphism. Or also you call it static binding, right? This is the concept of function overloading, all right? So, when I run the code, I will get the output like this, right?


So, these are the outputs you are getting. Area of the circle 5.0 into pi r square and then length into width 10 into 4 is 40 and of bh 6 into 3 is 18 by 2 is 9, right? So, all the outputs you are getting are correct. So, with this example, I hope you might have understood what is meant by function overloading, or you also call it static binding. Maybe we can see one more example, right.



So, that the concept will be still clear, right. So, here I use volume. The last program we used area. Let us assume volume. Right?

Volume. So, here you have double side. Right? One argument. So, only one argument here you have.

```
1 #include <iostream>
2 using namespace std;
3
4 // Overloaded function to calculate the volume of a cube
5 double volume(double side) {
6     return side * side * side;
7 }
8
9 // Overloaded function to calculate the volume of a rectangular box
10 double volume(double length, double width, double height) {
11     return length * width * height;
12 }
13
```

Example 2



 Oswaal 

And then it will return side into side into side. A cube. Right? One volume. Another volume: length into width into height.

So, three arguments. One is one argument. Another one is you have three arguments. Right? So, here, what is happening?

You are multiplying length into width into height. Return length into width into height. And the first case returns side into side into side. Right. One is nothing but the volume of a cube.

Another one is nothing but the volume of a rectangular box. So now, the question is, I am going to use the same function name. All right. Maybe let us see if you have. Yes.

One more volume. All right. Here you have the radius and height. Two arguments. Right.

So here, the first one is one argument. The second one is three arguments. And the third one is two arguments. So here you have pi times pi r squared h, right? For the cylinder, the volume of the cylinder is pi r squared h. So pi times radius times radius times height, right?

So, in all the cases, if you look, you have one function name. So, the function name is nothing but volume. So, let us consider your side is 5.0, length is 7.0, width is 6.0, and height is 8.0, all right. And the radius is 4.0, 9.0. Now, the volume of the cube.


```

14 // Overloaded function to calculate the volume of a cylinder
15 double volume(double radius, double height) {
16     return 3.14159265358979323846 * radius * radius * height;
17 }
18
19 int main() {
20     double side = 5.0;
21     double length = 7.0, width = 6.0, height = 8.0;
22     double radius = 4.0, cylHeight = 9.0;
23
24     // Call the overloaded functions for different shapes
25     cout << "Volume of the cube: " << volume(side) << endl;
26     cout << "Volume of the rectangular box: "
27         << volume(length, width, height) << endl;
28     cout << "Volume of the cylinder: " << volume(radius, cylHeight) << endl;
29
30     return 0;
31 }

```

$$\pi r^2 h$$



So, first, I am calling with the one value that I am passing, right. So, that means it has one argument. So, that means, obviously, this will call line number 5. So, line number 5 you have One argument function, and if you look at line number 6, it is returning side into side into side.

Correct? So, assume that I am passing what? 5.0. So, what is the result I have to get? 5.0 into 5.0 into 5.0.

Correct? So, 125 will be the output. So, you have only one argument. So, when I have only one argument, the compiler will call line number 5, right, that particular function, and then it will return side into side into side. That means it will return 5 into 5 into 5.

125 will be the answer. And line number 26, you have volume, this is one line, right. For our display point of view, I have written like this. So, you... Write it in a single line.

When you are practicing this, write it in a single line. So, volume of length into width into height, right? When you are calling the function volume, right? So, you are passing three values. You are passing three values.

So that means it has to call a function with three arguments. So, a function with three arguments. So naturally, it will not go to line number 5. A function with 3 arguments you have at line number 10. Right.


A function with 3 arguments you have at line number 10. So, line number 10, if you look, it is length into width into height. Right. So, length into width into height means 7 into 6 into 8. Right.

So, the multiplication of these 3 will be the result for line number 26. And 27, right? You write it in a single line. And then, line number 28, you are passing two arguments. Radius and height.

See on the right, right? So, volume, again the same function name. You are passing two values. So when you are passing two values, it looks for a two-argument function. So, line number 5 is a one-argument function.

```
1  #include <iostream>
2  using namespace std;
3
4  // Overloaded function to calculate the volume of a cube
5  double volume(double side) {
6      return side * side * side;
7  }
8
9  // Overloaded function to calculate the volume of a rectangular box
10 double volume(double length, double width, double height) {
11     return length * width * height;
12 }
13
```

Example 2



Line number 10, you have three arguments. So, therefore, it will call line number 15, right? You have a two-argument function. So, you have radius and height, correct? So, radius and height when you are passing.

So, π times r squared times h will be computed and that will be displayed, right? So, this is how it is exactly happening. So, if you look at the output, first you have π . So, π times π times π . So, we are getting 125 as the output.

So, the second case is the volume of a rectangular box. So, the volume of a rectangular box. It is length multiplied by width multiplied by height. So, 7 multiplied by 6 multiplied by 8. 42 multiplied by 8.

You are getting 336. That is also correct. And the third one is, you are passing two values. What is that? Radius and height.

Cylinder height. So, that means you are passing 4 and 9. 4.0 and 9.0, two argument construct, two argument function, right. So, two argument function in the sense of line number 15. So, $\pi r^2 h$ will be computed, and you are getting the result like this, right, from these two examples.

So, now it is very clear, right. So, when you are using the same function name more than once, right. And then it has to be different, either in the arguments. For example, in the last two programs, you have seen the arguments or the number of arguments are different. All right.

If the number of arguments is the same, their data type should be different. All right. Also, the return type. All right. So, these things we have to be very careful about.

All right. So, that is what the definition I had given. So, the number of arguments should be different or the data type should be different. The return type should be different, or in the arguments, if you are using the data type, that should be different, okay. So, then during compilation time, it will be decided which function should be called, right.

```
14 // Overloaded function to calculate the volume of a cylinder
15 double volume(double radius, double height) {
16     return 3.14159265358979323846 * radius * radius * height;
17 }
18
19 int main() {
20     double side = 5.0;
21     double length = 7.0, width = 6.0, height = 8.0;
22     double radius = 4.0, cylHeight = 9.0;
23
24     // Call the overloaded functions for different shapes
25     cout << "Volume of the cube: " << volume(side) << endl;
26     cout << "Volume of the rectangular box: "
27         << volume(length, width, height) << endl;
28     cout << "Volume of the cylinder: " << volume(radius, cylHeight) << endl;
29
30     return 0;
31 }
```

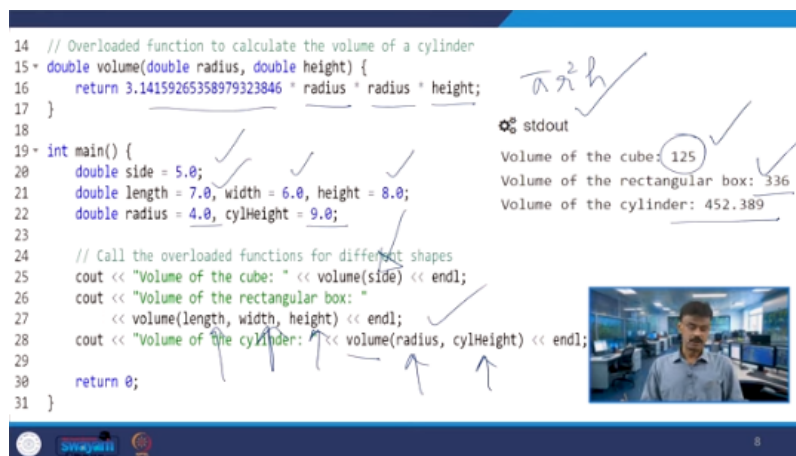
$\pi r^2 h$

stdout

Volume of the cube: 125

Volume of the rectangular box: 336

Volume of the cylinder: 452.389



So, this is what you call compile time polymorphism. So, now The theory is very clear, and you have seen two different examples. So, what you can do is practice with some other examples, okay? And the next concept is dynamic or runtime polymorphism, right?

So, what do you mean by dynamic or runtime polymorphism? So, here we talked about compile time polymorphism. So, during compilation time, which function will be chosen, right? So, here in this case, since we have already studied inheritance, correct?

Suppose you have class A; class A has the same function name, let us say function 1. Your dot dot dot means you have defined something, and class B is inheriting class A, let us say publicly inheriting class A. So, that is also having

function 1, right? So, in that case, which member function will be called, right? So, that is a challenge, right? So, this will be decided during runtime, right? The previous one was decided during compile time.

But in this case, this will be decided during runtime. So, therefore, it is called runtime polymorphism or dynamic binding, or you can also call this dynamic method dispatch, right? Dynamic method dispatch. So, several names, all are the same. The idea is the same, right?

So, when, which function will be called? That means it has to override some function. For example, the class B function is being called, right? It is calling the class B function. So that means it should be overridden, right?

So that is called overriding. In fact, we saw one example previously when we talked about inheritance. Anyway, we will consider the same example, right? So method overriding, the meaning of method overriding. So it is being done.

So, when you have the base class and derived class, assume that both functions exist, here you have function 1, let us say the return type is integer and here also the return type is integer. Both return types are the same and the number of parameters is the same. So, in that case, one is in the superclass, another one is in the subclass or base class or derived class. So, in that case, when you are creating an object, right, and then when it is invoking, the object is invoking the function, so which one will be called, right? So, in that case, the function should be overridden, right?

Dynamic/Run-time Polymorphism

- Run-time polymorphism is also known as Dynamic Method Dispatch.
- This type of polymorphism can be achieved using function overriding.
- In this approach a call to the overridden function is resolved during run-time.
- Method Overriding is done when a child or a derived class has a function with same name, parameters and return type as the parent or the superclass, then that function overrides the function in the base class.

Handwritten notes:

A ✓
int
fun1()
{.....}

B : A
fun1()
{.....}

Let us say, assume that the function in the base class is getting overridden, right? For example, this is being called. You have created an object, let us say

small b under capital B, right? So, let us say you have small b under capital B. So now this B is invoking let us say FUN1.

So here in this case this function should be called. So, that means this function in the base class should be overridden, right? The function overwrites the function in the base class. So, this function overwrites the function in the base class. So, you call this runtime polymorphism or dynamic binding.

So, what exactly happens in the case of runtime polymorphism? Assume that you have class P, right? So, the class P that has in P, right? The class P has one member data. And one member function.

That member function is display. Right. You have another class Q which is a derived class. It is publicly inheriting P. You have one member data Q. And another member function is also display. Like your base class.


So now the point is, when I am creating, let us say, an object. Let us say small Q1. Right. Under Q. And Q1 is invoking display. Right.

Run-time Polymorphism

```
class P
// base class declaration.
{
    int p; ✓
public:
    void display() ✓
    {
        cout<< "Class P ";
    }
};
```

q1

```
class Q : public P
// derived class
// declaration.
{
    int q; ✓
public:
    void display()
    {
        cout<<
    }
};
```



10



At one point in time, so now the point is it has to override the base class, and this function should be called, right? This function should be called, so then you call this runtime polymorphism, right? So maybe we will talk about one example. Let us say you have a class Animal, right? Assume that you have a class Animal. Animal is the base class, and you have one member function called Sound, right? Under this function, you have one print statement. This is a generic animal sound. One print statement. Some print statement.

And here in this case, we have this is a generic animal sound. And you have a derived class Dog, right? So, Animal is the base class. And you have a derived class called Dog, right? So, it is publicly inheriting Animal.

And then you have the same sound function. The member function is Sound. You have the same member function. See out the dog box. See out the dog box, right?

The class is over here. And then you have the third class. Right, which is cat, which is overriding, right, which is inheriting, in fact, which is inheriting animal, right, you have cat, right. So, cat also is having the same function name sound, right, the cat meows. In the case of a dog, the dog barks.

```
13 // Derived class - Dog
14 class Dog : public Animal {
15 public:
16     // Function with the same name in the
17     // derived class (hides base class function)
18     void sound() {
19         cout << "The dog barks" << endl;
20     }
21 };
22
23 // Derived class - Cat
24 class Cat : public Animal {
25 public:
26     // Function with the same name
27     // in the derived class (hides base class function)
28     void sound() {
29         cout << "The cat meows" << endl;
30     }
31 };
32
```



That is a cout statement. So, if you look at the diagram, all are having the same name, right? So, in this case, when you are creating an object, right? Assume that I am creating an object animal small a under the class animal capital A. I have an object dog small d under the class capital D. I have an object cat under the class cat, right? Capital C. So, in these cases, suppose I

The animal is invoking sound, right? Animal is invoking the sound member function, right? So, in that case, you have three sound functions. In fact, animal is the base class. So, in the base class, you have sound, right?

Function sound. So, this member function will be invoked, and this will be displayed first, correct? In the second case, the dog object is invoking sound. Right? The dog object is invoking sound.

So, in that case, what will happen? It will override the base class, right? Line numbers 8 to 10 will be overridden. So, this will call the dog class void sound, this particular member function. So, in that case, it will print the dog box.

The second output you will get is the dog box, right? And the third case, Cat dot sound, right. So, the cat is invoking the member function sound. So, in this case, what will happen, right?

So, under cat, right. So, this is a derived class; the base class is this. The base class will be overridden, and this particular function will be called. When it is calling this particular function, cat's meows will be displayed, right. So, in the order it is getting displayed in this case, right. You can try in a different order.

First, put doc.sound, correct? So, you may think it was initially going to the main, right? Main, in the sense, base class. I mean to say the base class, right? But in this case, so what is happening?

If you look at the output, so these are all the outputs, right? So, the first one is the corresponding object calling the corresponding class. The second one is it not calling; let us call it the overridden, right? So, here you can see that, right? So, in fact

Hiding the animal's version in the animal's version, you have the sound function, right? The sound member function is not being called, right? So, this is what exactly is happening over here. When you run the code, you are getting the output like this, right? So, the first one is for the animal case, the second one is for the dog, and the third one is for the cat, right? So, when it is taking many forms, right? When the entity is taking many forms, you call it polymorphism. So, under this, we have studied compile-time polymorphism, right? You call it static binding, or the example that we just saw is dynamic binding, or you call it runtime polymorphism. So, with this, I am concluding this particular lecture. Thank you.