

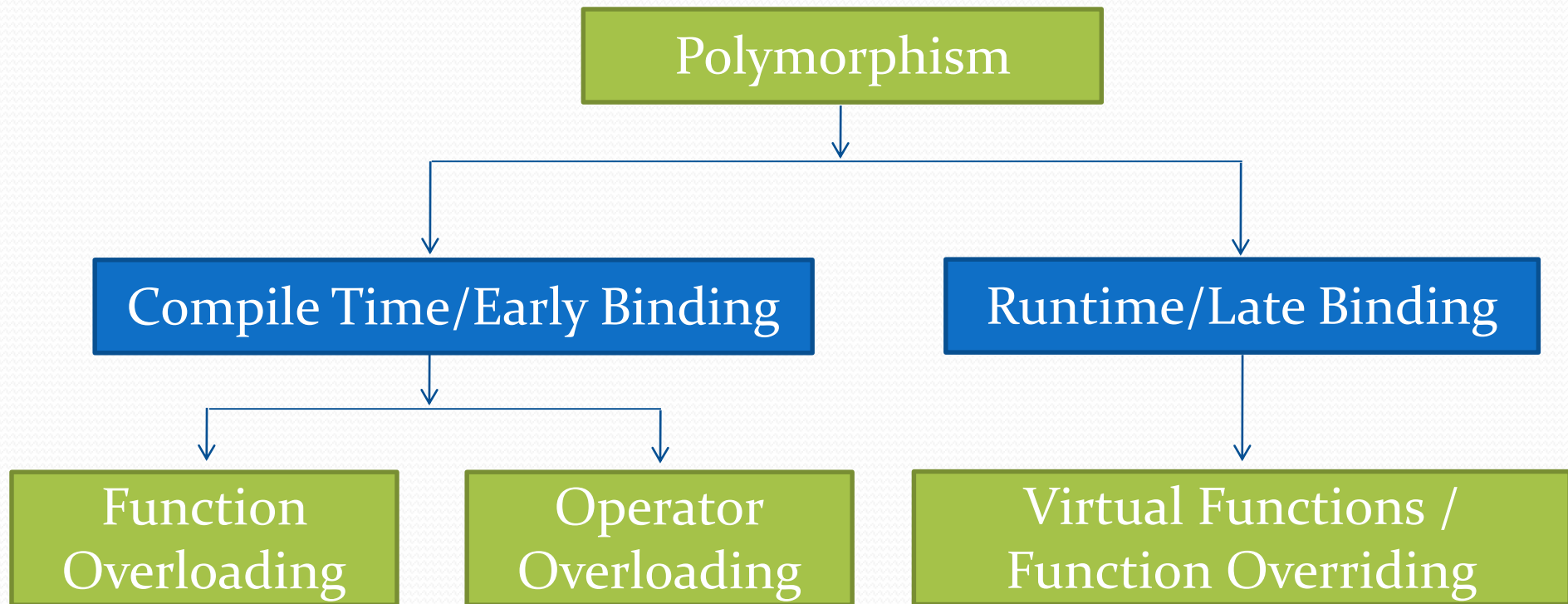
Virtual Functions

Haresh Jaiswal

Rising Technologies, Jalna.

Introduction

- Virtual functions belongs to the branch of Runtime Polymorphism in C++



Introduction

- Polymorphism is classified into 2 branches
 - Compile Time Polymorphism/**Early Binding**/Static Binding
 - Runtime Polymorphism/**Late Binding**/Dynamic Binding

What is Binding?

- For every function call; compiler binds or links the call to one function definition.
- This linking can happen at 2 different time
 - At the time of compiling program, or
 - At Runtime

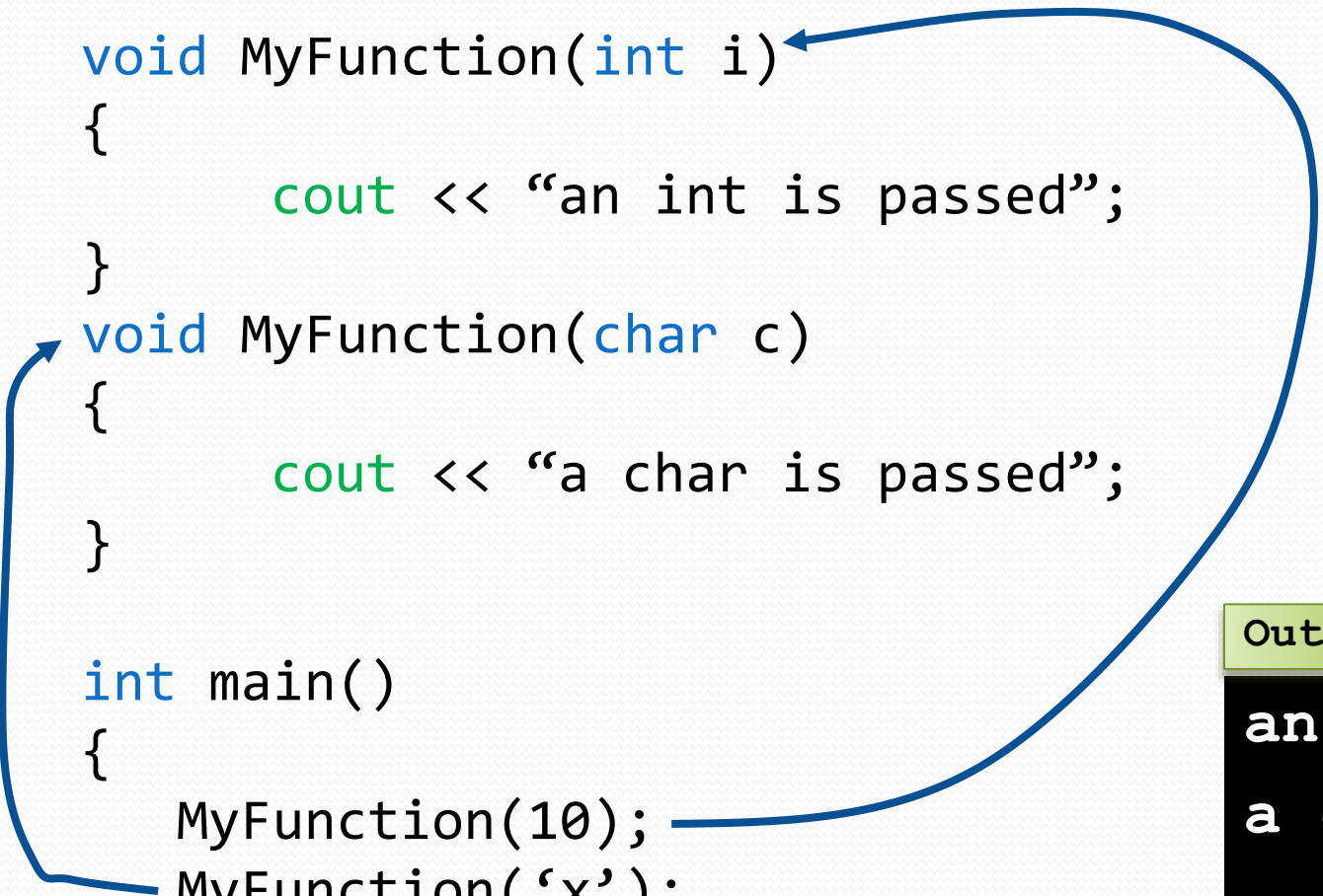
Compile Time Polymorphism

- Function Overloading is an example of Compile Time Polymorphism.
- This decision of binding among several functions is taken by considering formal arguments of the function, their data type and their sequence.

Example of compile time polymorphism

```
void MyFunction(int i)
{
    cout << "an int is passed";
}
void MyFunction(char c)
{
    cout << "a char is passed";
}

int main()
{
    MyFunction(10);
    MyFunction('x');
}
```



Output

```
an int is passed
a char is passed
```

Runtime Polymorphism

- In late binding; call to a function is resolved at Runtime, the compiler determines the type of object at execution time and then binds the function call to a function definition.
- Late binding is also called as Dynamic Binding or Runtime Binding.
- Virtual Functions are example of Late Binding in C++
- Runtime polymorphism is achieved using pointers.

Pointers behaviour in Polymorphism

- A base class pointer variable can hold address of derived class object, but it can access only members of base class.

Pointers behaviour in Polymorphism

```
class base
{
public:
    void show()
    {
        cout << "Show from base";
    }
};

class derived : public base
{
public:
    void show()
    {
        cout << "Show from derived";
    }
};
```

```
int main()
{
    base *ptr;
    derived ob;

    ptr = &ob;

    ptr -> show();
}
```

Output

Show from base

Pointers behaviour in Polymorphism

- You can see that even the pointer holds address of derived class object; it has called the base version of show() method.
- The problem is even a base class pointer holds address of derived type of object, it can access only members of base class, because the base pointer variable doesn't have any idea about the structure of derived class.
- A base class can't have any idea about what derived class has added to it.

Using virtual Keyword

```
class base
{
public:
    virtual void show()
    {
        cout << "Show from base";
    }
};

class derived : public base
{
public:
    void show()
    {
        cout << "Show from derived";
    }
};
```

```
int main()
{
    base *ptr;
    derived ob;

    ptr = &ob;

    ptr -> show();
}
```

Output

Show from derived

Using virtual Keyword

- Using **virtual** keyword with base class version of show function; late binding takes place and derived version of the function will be called, because base pointer points an derived type of object.
- We know that **in runtime polymorphism the call to a function is resolved at runtime depending upon the type of object.**

How virtual function works?

- Base class pointer can point to derived class object.
- In this case, using base class pointer if we call some function which is in both classes, then base class function is invoked.
- But if we want to invoke derived class function using base class pointer, it can be achieved by defining the function as virtual in base class, this is how **virtual functions** support runtime polymorphism.

Virtual functions.

- A virtual function is a member function that is declared as virtual within a base class and redefined by a derived class.
- To create virtual function, precede the base version of function's declaration with the keyword **virtual**.
- When a class containing virtual function is inherited, the **derived class can redefine (Override) the virtual function to suit its own unique needs.**
- The method name and type signature should be same for both base and derived version of function.

Using virtual Keyword

```
class circle
{
protected:
    float radius;

public:
    circle(float r)
    {
        radius = r;
    }

    virtual float area()
    {
        float a;
        a = 3.14 * radius * radius;
        return a;
    }
};
```

```
class cylinder : public circle
{
private:
    float height;

public:
    cylinder(float r, float h) : circle(r)
    {
        height = h;
    }

    float area() // overriding area method
    {
        float a;
        a = (2 * 3.14 * radius * radius)
            + (2 * 3.14 * radius * height);
        return a;
    }
};
```

Using virtual Keyword

```
int main()
{
    circle *ptr;

    circle CircleOb(8);
    cylinder CylOb(8, 4);

    ptr = &CircleOb;
    cout << endl << "Area of Circle : " << ptr -> area();

    ptr = &CylOb;
    cout << endl << "Area of Cylinder : " << ptr -> area();

    return 0;
}
```

Output

```
Area of Circle : 200.96
Area of Cylinder : 602.88
```


Why method overriding?

- **Method overriding** allows a derived class to provide a specific implementation of a method that is already provided by one of its base class.
- The implementation in the derived class overrides (replaces) the implementation in the base class by providing a method that has same name, same parameters (signature), and same return type as the method in the parent class has.
- Using a base class pointer variable, we can point to object of any child class, depending upon the type of object at runtime a particular method will be called if that method is made virtual in base class & overridden in derived class.

Method Overloading vs. Overriding?

- **Overloading** occurs when two or more methods in one class have the same method name but different parameters (signature), **Overriding** means having two methods with the same method name and parameters (*method signature*), one of the method is in the parent class and the other is in child class.
- Call to an **Overloaded** method is resolved at compile time, while call to an **Overridden** method is resolved at runtime depending upon the type of object.

Method Overloading vs. Overriding?

	Overloading	Overriding
Definition	Methods having same name but each must have different number of parameters or parameters having different types & order.	Sub class have method with same name and exactly the same number and type of parameters and same return type as super class method.
Meaning	More than one method shares the same name in the class but having different signature.	Method of base class is re-defined in the derived class having same signature.
Behaviour	To Add/Extend more to method's behaviour.	To Change existing behaviour of method.
Polymorphism	Compile Time	Run Time
Inheritance	Not Required	Always Required
Method Signature	Must have different signature	Must have same signature.

Method Overloading vs. Overriding?

	Overloading	Overriding
Method Relationship	Relationship is between methods of same class.	Relationship is between methods of super class and sub class.
No of Classes	Does not require more than one class for overloading.	Requires at least 2 classes for overriding.
Example	<pre>class Sample { public: void MyFunction() { cout << "MyFunction Called"; } void MyFunction(int param) { cout << "MyFunction Called : " << param; } };</pre>	<pre>class Base { public: virtual void MyFunction() { cout << "Base MyFunction"; } }; class Derived : public Base { public: void MyFunction() { cout << "Derived MyFunction"; } };</pre>