

Polymorphism

Chapter 10



What is Covered ?

- What is Polymorphism?
- Types of Polymorphism
- Static Polymorphism using Function Overloading and Operator Overloading
- Dynamic Polymorphism
- Virtual Functions
- Pure Virtual Functions
- Abstract Classes



Introduction

- Polymorphism means one name – many forms
- When you call a function with a common name, the runtime decides which implementation to call depending on the current program context. This feature is known as polymorphism



Types of Polymorphism

- Static Polymorphism
- Runtime Polymorphism



Static Polymorphism

- Implemented in C++ by way of
 - function and
 - operator overloading



Function Overloading

- **Function Declarations**

```
void Show (int i);  
void Show (float f)  
void Show (char *ptr);
```

- **Function Calling**

```
Show (5);  
Show ((float)5.0);  
char *ptr = "Test";  
Show (ptr);
```



Rules for Function Overloading

- All the overloaded functions have the same name
- The number of arguments may differ
- The data type of arguments may differ
- The order of arguments may differ
- The return type may differ



Rule: The number of arguments may differ

```
void Show (int i);
```

```
void Show (int i, float f);
```

```
void Show (int i, float f, double d);
```



Rule: The data type of arguments may differ

```
void Show (int i);
```

```
void Show (float i);
```



Rule: The order of arguments may differ

```
void Show (int i, float f);  
void Show (float f, int i);
```



Program Example

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class Complex
{
public:
    int real, imag;
public:
    Complex (int r, int i)
    {
        real = r;
        imag = i;
    }
    void dump()
    {
        cout << "Real = " << real << " Imag = " << imag << endl;
    }
};
```



Program Example - Continued

```
class Calculator
{
    public:
    int Add(int a, int b)
    {
        return (a + b);
    }
    Complex Add(Complex a, Complex b)
    {
        return (Complex (a.real + b.real, a.imag + b.imag));
    }
};
```



Program Example - Continued

```
void main()
{
    Calculator calc;
    cout << "Real Number Addition: " << endl;
    cout << "50 + 70 = " << calc.Add(50, 70) << endl;
    cout << endl;
    cout << "Complex Number Addition: " << endl;
    Complex c1 (10, 5);
    Complex c2 (2, 4);
```



Program Example - Continued

```
Complex c3 = calc.Add (c1, c2);  
cout << "C1: ";  
c1.dump();  
cout << "C2: ";  
c2.dump();  
cout << "C3 = C1 + C2 : " ;  
c3.dump();  
}
```



Program Output

Real Number Addition:

$$50 + 70 = 120$$

Complex Number Addition:

$$C1: \text{Real} = 10 \text{ Imag} = 5$$

$$C2: \text{Real} = 2 \text{ Imag} = 4$$

$$C3 = C1 + C2 : \text{Real} = 12 \text{ Imag} = 9$$



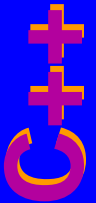
Merits/Demerits of Static Polymorphism

The calls to the overloaded methods are resolved at the compile time. This helps in creating faster programs. However, since all such calls must be resolved at compile time, it deprives us from the flexibility of plugging the new code at the runtime

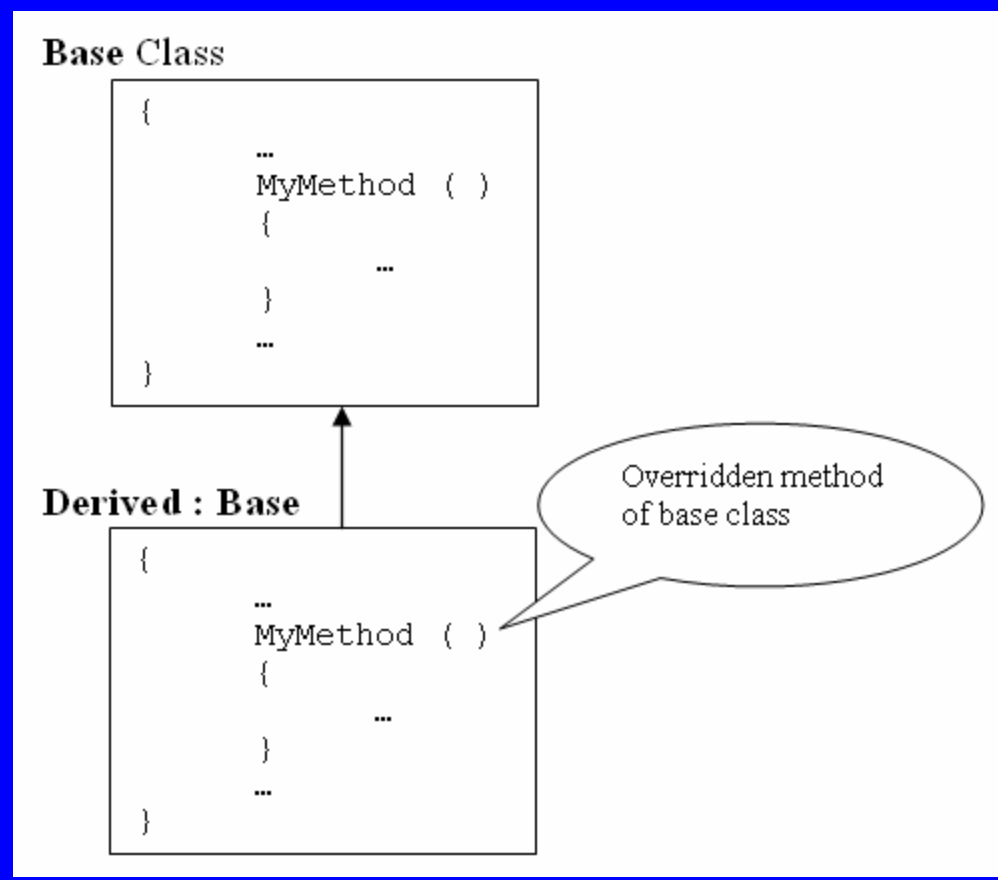


Dynamic Polymorphism

- In case of static polymorphism, for the overloaded functions, the compiler resolves which function to call at the compile time
- In case of dynamic polymorphism, this decision is delayed until the runtime



Method Overriding





Using Pointers for calling overridden methods

```
class Base
{
public:
    void Show ()
    {
        cout << "In Base class Show method" << endl;
    }
};
```

The Derived class overrides the Show method of Base class

```
class Derived:public Base
{
public:
    void Show ()
    {
        cout << "In Derived class Show method" << endl;
    }
};
```



Example

```
Base base;
```

```
Derived derived;
```

```
Base *baseptr = &base;
```

```
baseptr->Show();
```

```
baseptr = &derived;
```

```
baseptr->Show();
```



Program Example - Continued

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class Base
{
public:
    void Show ()
    {
        cout << "In Base class Show method" << endl;
    }
};
```



Program Example - Continued

```
class Derived:public Base
{
public:
    void Show ()
    {
        cout << "In Derived class Show method" << endl;
    }
};
void main()
{
    Base base;
    Derived derived;
    cout << "Using pointer to base class" << endl;
```



Program Example - Continued

```
Base *baseptr = &base;
baseptr->Show();
baseptr = &derived;
baseptr->Show();
cout << endl;
cout << "Using pointer to derived class" << endl;
Derived *derivedptr = &derived;
derivedptr->Show();
derivedptr = (Derived *) &base;
derivedptr->Show();
}
```



Program Output

```
Using pointer to base class
```

```
In Base class Show method
```

```
In Base class Show method
```

```
Using pointer to derived class
```

```
In Derived class Show method
```

```
In Derived class Show method
```



Virtual Functions

If we prefix the function declaration with the virtual keyword

```
virtual void Show ()
```

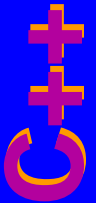
It is observed that the runtime now calls the implementation of the **Show** method as expected



Pure Virtual Functions

- The methods declared in the base class are overridden in the derived classes
- The derived classes provide the implementations for these methods
- A pure virtual function does not provide the implementation and is declared as follows:

```
virtual void Show () = 0;
```



Program Example

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class Base
{
public:
    virtual void Show () = 0;
};
```



Program Example - Continued

```
class Derived1:public Base
{
public:
    void Show ()
    {
        cout << "In Derived1 Show method" << endl;
    }
};
class Derived2:public Base
{
public:
    void Show ()
    {
        cout << "In Derived2 Show method" << endl;
    }
};
```



Program Example - Continued

```
void main()
{
    Derived1 derived1;
    Derived2 derived2;
    Base *baseptr = &derived1;
    cout << "Using pointer to Base class" << endl;
    cout << "Calling Show method in Derived1 class" << endl;
    baseptr->Show();
    cout << "Calling Show method in Derived2 class" << endl;
    baseptr = &derived2;
    baseptr->Show();
}
```



Program Output

```
Using pointer to Base class
```

```
Calling Show method in Derived1 class
```

```
In Derived1 Show method
```

```
Calling Show method in Derived2 class
```

```
In Derived2 Show method
```



Merits/Demerits of Dynamic Polymorphism

- In Dynamic polymorphism the call to a particular form of function is determined at runtime, this process is called late binding
- Late binding helps in development of large applications where code flexibility is desired
- By using late binding, code changes can be easily accomplished without major modifications



Abstract Classes

- If a class contains one or more pure virtual functions, it is called an Abstract class
- Abstract class cannot be instantiated

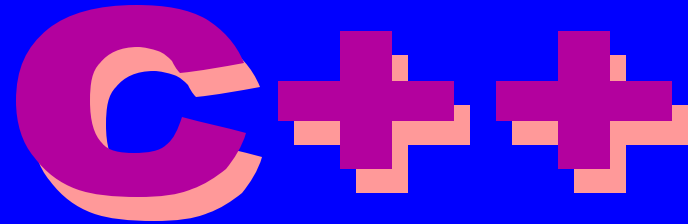
```
class Base
{
public:
    virtual void Show () = 0;
};
```

```
class Derived:public Base
{
};
Derived derived; // does not compile
```



Summary

- **There are two types of polymorphism supported in C++**
 - static and
 - dynamic polymorphism
- **Static polymorphism is implemented using**
 - function overloading or
 - operator overloading
- **Dynamic polymorphism defers the function binding to the runtime**
- **By using “virtual keyword” compiler calls the appropriate implementation depending on the object type to which the pointer currently points to**
- **A pure virtual function is created by assigning zero in the function declaration**
- **A class containing a pure virtual function is treated as an abstract class, as it cannot be instantiated**



Conclusion