Object Oriented Programming

6 Operator Overloading &Polymorphism

Overloading in C++

• C++ allows function overloading, whether for *member functions* or *independant functions*.

Example:

```
class ratio
{ private : int num, den ;
public :
ratio ( int n, int d=1) ;
ratio(int n) ;
} ;
```

• Assigning the same name to *different functions* allows the compiler to choose the appropriate one at **call time**, based on the **number** and **types of arguments**.

Operator Overloading in C++

- C++ also supports operator overloading.

Example: **a+b**

- The symbol **+** can represent different operations depending on the types of **a** and **b**:
- addition of two integers,
- addition of two float numbers,
- or addition of two double values.

The **+** operator is interpreted according to the context.

- In C++, any existing operator—whether *unary* or *binary*—can be overloaded. This allows us to define custom types using classes, with fully integrated operators.

✎ For instance, it is possible to define how expressions like a + b, a - b, a * b, and a / b behave for a ratio class.

Overdefinition mechanism

- Operator Overloading Syntax:

```
return_type  operator  operator_symbol (parameters) { statement1; statement2; }
```

- The keyword operator defines a new action or operation to the operator.

- operator_symbol denotes the symbol of the operator we want to overload (for example: +, <, -, ++, etc.).

Operator Overloading *(Example 1)*


```
#include <iostream>
using namespace std;
class Count {
private:
int i;
public:
Count(int n) { i = n; }
void operator +() { i = i+1; }
void display() { cout << "Count = " << i; } };
int main() { Count c(2);
+c; // call of the “void operator+” function
c.display();
return 0; }
```

Count=3

Operator Overloading *(Example 2)*

```
class point {
private :
 int x, y ;
 public : point (int x =0, int y =0){ …} ;
friend point operator+ (point , point) ;
void display( ){cout <<"x="<<x<<endl;
y="<<y<<endl;}; } ;
point operator+ (point a, point b)
{ point p;
p.x =a.x +b.x; p.y = a.y + b.y;
 return p; }
int main() {
point a (1,2);
point b (2,5);
point c;
c= a+b;
c.display();
c= a+b+c;
c.display(); }
```

?????

NB.

c=a+b; =>

c= operator+(a,b);

Operator Overloading *(Example 3)*

```
class point {
private :
 int x, y ;
 public : point (int x =0, int y =0){ …} ;
void display( ){cout <<‘’x=‘’<<x<<endl;
y=‘’<<y<<endl;};
point operator+ (point a)
{ point p;
p.x =x +a.x; p.y = y + a.y;
 return p; }
} ;
int main() {
point a (1,2);
point b (2,5);
point c;
c= a+b;
c.display();
c= a+b+c;
c.display(); }
```

?????

NB.

c = a+b; =>

c = a.operator+(b);

Operator Overloading (Some Remarks)

- The symbol following the keyword operator **must be** an **operator already defined** for the base types =>We are not allowed to create new symbols.
- Some operators cannot be redefined at all (.)
- The **plurality** of the **operator must be** preserved: **unary** or **binary**.
  - Binary: + - / *
  - Unary: -- ++ new delete
- We **must use a class context**:An operator can only be overridden if it has at least one class-type argument (a member function) and for an independent function with at least one class-type argument (friend function).

Operator Overloading -> Polymorphism

- Redefining (**overloading**) an operator does not change its natural meaning. It can be used for both variables of built-in data type and objects of user-defined data type.

- Operator overloading is one of the most valuable concepts introduced by C++ language.

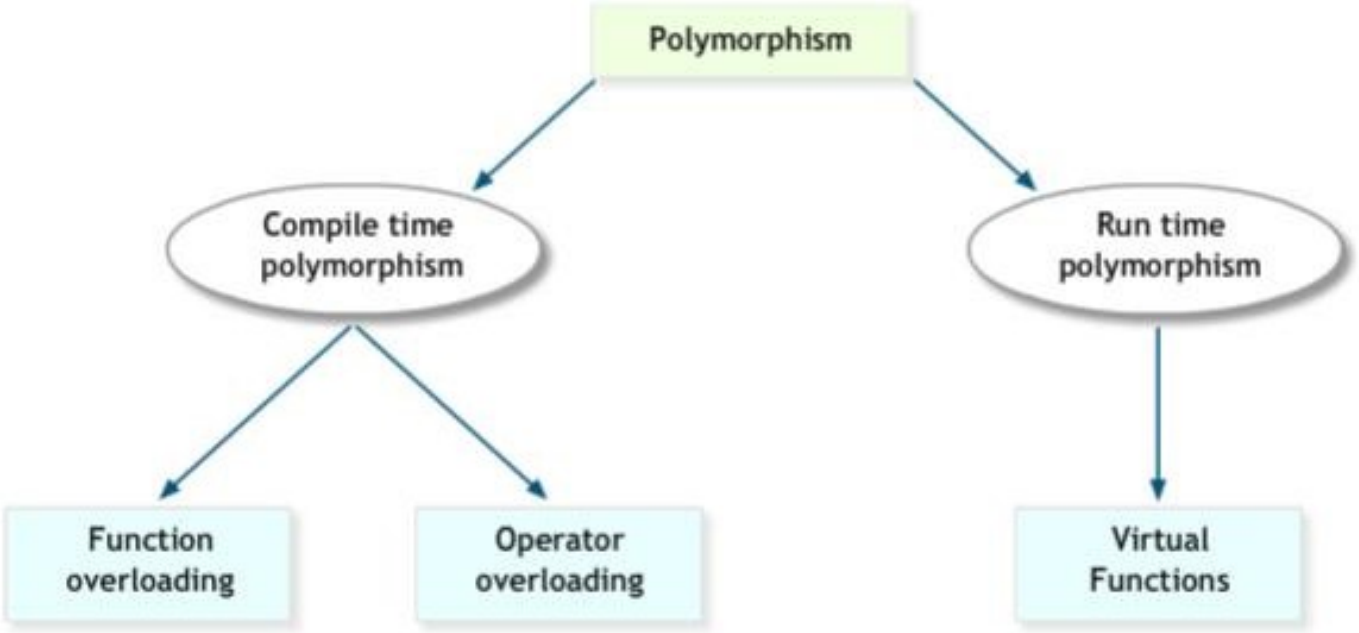
- It is a type of **polymorphism**.

What is Polymorphism?

- The word **polymorphism** means having **many forms**.
- A real life example (A woman as a sister, a mother and an employee at the same time).
- In C++, polymorphism concept can be applied to **functions** and **operators**.
  - A single function can work differently in different situations.
  - An operator works different when used in different context.

Types of Polymorphism

- Polymorphism in C++ can be classified into two types:
  - ✓ Compile-time Polymorphism
  - ✓ Runtime Polymorphism



Compile-time Polymorphism

- Known as early binding and static polymorphism => the compiler determines how the function or operator will work depending on the context.
- It is achieved by overloading functions and operators.

Compile-time Polymorphism: Function Overloading

- Function overloading is a feature of OOP where two or more functions can have the same name but behave differently for different parameters.

- Such functions are said to be overloaded; hence, this is known as Function Overloading.

- Functions can be overloaded **either by changing** the **number of arguments** or **changing the type of arguments**.

Compile-time Polymorphism: Function Overloading


```
class Example1 {
public:
void add(int a, int b) {
        cout << "Integer Sum = " << a + b
        << endl;
   }
void add(double a, double b) {
        cout << "Float Sum = " << a + b
        << endl ;    }
};
```

```
int main() {
    Example1 E;

    E.add(10, 2);

    E.add(5.3, 6.2);
```

Runtime Polymorphism

- Known as late binding and dynamic polymorphism => the function call in runtime polymorphism is resolved at runtime.

- It is implemented using function overriding with virtual functions.

Runtime Polymorphism: Virtual and Overriding Function

- A virtual function is a member function of a class defined with the keyword virtual and its functionality can be overrided in that class derived classes.

- Virtual functions are used to give different meanings to a function, and this is part of the concept of polymorphism.

Virtual and Overriding Function (Example 1)


```
class Base_class
{
public:
   virtual void Create()
   {
       cout << "Base class Test" << endl; }
};
class Derived_class : public Base_class
{
public:
   void Create() override
   {
       cout << "Derived class Test" ;}
};
```

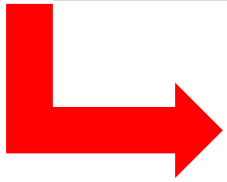
```
int main()
{
    Base_class  *x, *y;
    x = new Base_class();
    x->Create();

    y = new Derived_class();
    y->Create();
}
```

Virtual and Overriding Function (Example 2)


```
#include <iostream>
using namespace std;
class Base {
public:
virtual void print()
{ cout << "Base Function" << endl;
}
}; class Derived : public Base
 { public:
void print() override
{ cout << "Derived Function" << endl;
} };
```

```
int main()
 { Derived derived1;
// pointer of Base type that
points to derived1
Base* base1 = &derived1;
// calls member function of
Derived class
base1->print(); return 0;
 }
```



Output: Derived Function

Virtual and Overriding Function (Example 2)

The function print() is overridden even when we use a pointer of Base type that points to the Derived object derived1.

```
class Base {
    public:
      virtual void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1;
    Base* base1 = &derived1;

    base1->print();

    return 0;
}
```

print() of Derived class is called because print() of Base class is virtual

Virtual and Overriding Function: C++ Override Specifier

• **C++ 11** provides the new specifier override that is very useful to avoid **common mistakes** while using **virtual functions**.

NB. If we use a function prototype in Derived class and define that function outside of the class, then we use the following code:

```
class Derived :
public Base { public: // function prototype void print() override; };
 // function definition
void Derived::print() { // code }
```

Virtual and Overriding Function: Use of C++ Override (1)

- Using the **override specifier prompts** the compiler to display error messages when some mistakes are made.

- When using virtual functions, it is possible to make mistakes while declaring the member functions of the derived classes.

- The program will simply compile but the virtual function will not be overridden.

Virtual and Overriding Function: Use of C++ Override (2)

Some of possible mistakes when using virtual functions are:

1. Functions with incorrect names: For example display and diplay().

2. Functions with different return types

3. Functions with different parameters

4. No virtual function is declared in the base class.

Some Rules concerningVirtual Functions (1/3)

1. The keyword virtual should not be repeated in the definition if the definition occurs outside the class declaration.

```
class point { intx ; inty ;
public:
virtual void display ( ); };
virtual void point: : display ( ) //error
 { Function Body }
```

2. A virtual function cannot be a static member since a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.

```
class point { int x ; int y ;
public: virtual static int length ( ); //error
int point: : length ( )
 { Function body }
```

Some Rules concerningVirtual Functions (2/3)

3. A virtual function **cannot have a constructor** member function but it **can have the destructor** member function.

```
class point { int x ; int y ;
public:
 virtual point (int xx, int yy) ; // constructors, error
Virtual ~point(); //  It is valid
void display ( ) ;
 int length ( ) ; };
```

4. It is an error to redefine a virtual method with a change of return data type in the derived class with the same parameter types as those of a virtual method in the base class.

```
class base { int x,y ;
public: virtual int sum (int xx, int yy ) ; //error } ;
class derived: public base { intz ;
public: virtual float sum (int xx, int yy) ; };
```

Some Rules concerningVirtual Functions (3/3)

5. Only a member function of a class can be declared as virtual. A non member function (non-method) of a class cannot be declared virtual

```
virtual void display ( ) //error, non-member function
{ Function body }
```

Advantages of Polymorphism

- Reuse: polymorphism allows for more flexible and reusable code.
- Flexibility: instead of having to modify code, polymorphism allows programmers to easily extend functionality.
- Simplified Code: polymorphism allows programmers to create more concise code. Polymorphic codes are simple because they retain the exact initial instruction but have the ability to modify themselves.

Disadvantages of Polymorphism

- Performance Issues: A machine can struggle to execute polymorphic code, especially when it becomes too complex.

- Difficult to Implement: Polymorphic code is more difficult to implement due to the number of variables.