Operator Overloading

Introduction

- Polymorphism is one of the crucial feature of OOP.
- It simply means ‘one name many forms’.
- The concept of polymorphism is already implemented using overloaded function and operator.

Operator Overloading is a specific case of polymorphism in which some or all of operators like +, =, or == have **different implementations** depending on the types of their arguments.

```
Point p1,p2;

p1=p1+p2;
```

- **Operators overloading in C++:**
- You can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.
- Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Syntax and Overview

- Operator overloading used for customised behaviour of different operators

```
class className {
   ... .. ...
   public
      returnType operator symbol (arguments) {
          ... .. ...
      }
   ... .. ...};
```

Operator overloading is used for customised functionality of a an operator in a class.

This a powerful tool in C++, where hundreds of lines of code can be slashed, if operator overloaded it done properly and efficiently.

Fundamentals of Operator Overloading

- Types
  - Built in (**`int`**, **`char`**) or user-defined
  - Can use existing operators with user-defined types
    - Cannot create new operators
- Overloading operators
  - Create a function for the class
  - Name function **`operator`** followed by symbol
    - **`Operator+`** for the addition operator **`+`**
- Obj*obj
- Obj%int

- Operator function must be either member function or friend function.
- Friend function will have only **one** argument for unary operators and **two** for binary operators.
- Member function will have **no** arguments for unary and **one** argument for binary operators.

example

- Suppose vector is a classname
  - Vector operator+(vector);
  - Vector operator-();
  - Int operator==(vector);

  - Friend vector operator+(vector, vector)
  - Friend int operator ==(vector,vector)

Operator Overloading

- C++ has the ability to provide the operators with the special meaning for a data type.
- The mechanism of giving such special meanings to an operator is known as **Operator overloading.**
- It provides a flexible option for creation of new definitions for most of C++ operators.
- Operator Overloading is a specific case of polymorphism in which some or all of operators like +, =, or == have different implementations depending on the types of their arguments.

Contt....

- We can overload all C++ operators **except** the following:

1. Class member access operator(.,.*).
2. Scope resolution operators(::);
3. Size operator(sizeof).
4. Conditional operators(?:)

Restrictions on Operator Overloading

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Fundamentals of Operator Overloading

- Types
  - Built in (**`int`**, **`char`**) or user-defined
  - Can use existing operators with user-defined types
    - Cannot create new operators
    - Semantics of an operator can be extended but syntax cannot be changed.
- Overloading operators
  - Create a function for the class
  - Name function **`operator`** followed by symbol
    - **`Operator+`** for the addition operator **`+`**

Syntax

return_type operator # (argument list)

{……}

Eg-

Complex operator +(complex);

Void operator >(distance);

- First of all we must specify the class to which the operator is applied.
- Function used for this is called operator function.

return-type classname:: operator op(arglist)

{

function body;

}

Steps in process of overloading

- Create a class that defines data types that is to be used in the overloading operation.
- Declare the operator function operator op() in public part of the class.
- Define the function to implement the required operation.

Types of Operator

- Unary operator
- Binary operator

Unary Operators

- Operators attached to a single operand (-a, +a, --a, a--, ++a, a++)

- The pre and post increment and decrement operators and overloading in different ways. This is because they have a different effect on objects and their values.

- e.g. a=14;
- cout << a++; // will print 14 and increment a
- cout << ++a; // will increment a and print 15

Prefix unary operators

Simple Prefix Unary Operators

- Are defined by either a member function that takes no parameter or a non-member function that takes one parameter

  Example:

```
i1.operator -()
or
 operator -(i1)
or
-i1
```

```
#include <iostream>
using std::cout;

class space {
private:
 int x; int y; int z;
public:
  space(int a, int b, int c) : x(a), y(b), z(c) {}

void display() {
 cout << x << " " << y << " " << z << "\n"; }

void operator-() {
       x = -x; y = -y; z = -z;
   }
};
```

```
void main()
{
space s(10, -20, 30);
cout << "s :";
s.display();
-s;
cout << "s :";
s.display();
}
```

Example: Unary Operators

```
class UnaryExample
{
    private:
        int m_LocalInt;
    public:
        UnaryExample(int j)
        {
            m_LocalInt = j;
        }
        int operator++ ()
        {
            return (m_LocalInt++);
        }
};
```

Example: Unary Operators (contd.)

```
void main()
{
     UnaryExample object1(10);
    cout << object1++; // overloaded operator results in value
                       // 11
}
```

Friend function

```
friend void operator –(space &s);  //declaration


 void operator –(space &s)
{
 s.x=-s.x;
 s.y=-s.y;
 s.z=-s.z;
}
```

Binary Operators

- Operators attached to two operands (a-b, a+b, a*b, a/b, a%b, a>b, a>=b, a<b, a<=b, a==b)

Example: Binary Operators (contd.)

```
void main()
{
    BinaryExample object1(10), object2(20);
   cout << object1 + object2; // overloaded operator called
}
```

- ob1 = ob1 + ob2;

  equivalent to

- ob1 = ob1.operator+(ob2);

```
class complex
{
        float x;
        float y;
public:
        complex() { x = 10; y = 20; }
        complex&  operator -(complex c)     {
                complex temp;
                temp.x = x - c.x;
                temp.y = y - c.y;
                return(temp); }

};
void main()
{
complex C1, C2, C3;
C3 = C1 - C2;
C3.show();
}
```

Restrictions on Operator Overloading

- Cannot change
  - How operators act on built-in data types
    - i.e., cannot change integer addition
    - When an operator is overloaded original meaning is not changed.
    - For eg: + operator can be used to add two vectors but it can also used to add two integers.
  - Precedence of operator (order of evaluation)
    - Use parentheses to force order-of-operations
  - Associativity (left-to-right or right-to-left)
  - Number of operands
    - **&** is unitary, only acts on one operand

Rules for overloading operators.

- Only existing operators can be overloaded. New operators cannot be created.
- Operators must be overloaded explicitly
  - Overloading **+** does not overload **+=**
- The overloaded operator must have at least one operand that is of user defined type.
- We cannot change the basic meaning of the operator.
- Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.

Rules for overloading operators.

- Unary operators overloaded by means of member function, take no explicit arguments and return no explicit values.
- Binary operators overloaded through a member function take two explicit arguments.
- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

Rules of thumb can help you determine which is best for a given situation:

- If you're overloading **assignment** (=), **subscript** ([]), **function** call (()), or **member** selection (->), do so as a ***member*** function.
- For **unary** operator, overload as a ***member*** function.
- For a **binary** operator that does not modify its left operand (e.g. operator+), overload as a ***normal*** function (preferred) or ***friend*** function.
- For a **binary** operator that modifies its left operand, but you can't add members to the class definition of the left operand (e.g. operator<<, which has a left operand of type ostream), overload as a ***normal*** function (preferred) or ***friend*** function.
- For a **binary** operator that modifies its left operand (e.g. operator+=), and you can modify the definition of the left operand, overload as a ***member*** function.