Operator Overloading

Operators are by default defined in **built-in** types.
C++ allows us to re-define them for **user-defined types**.

syntax

> *type* `operator` *operator-symbol (parameter-list)*

- **`operator+`** is a formal function name that can be used like any other function name.
- The operator $+$
  - formal name, namely `operator+` (consisting of 2 keywords)
  - nick name, namely $+$
- The **nick name** can only be used when calling the function.
- The **formal name** can be used in any context, when declaring the function, defining it, calling it, or taking its address.

Redefinable Operator

Operator	Name	Type
,	Comma	Binary
!	Logical NOT	Unary
!=	Inequality	Binary
%	Modulus	Binary
%=	Modulus assignment	Binary
&	Bitwise AND	Binary
&	Address-of	Unary
&&	Logical AND	Binary

Operator	Name	Type
&=	Bitwise AND assignment	Binary
( )	Function call	—
( )	Cast Operator	Unary
*	Multiplication	Binary
*	Pointer dereference	Unary
*=	Multiplication assignment	Binary
+	Addition	Binary
+	Unary Plus	Unary
++	Increment 1	Unary
+=	Addition assignment	Binary
-	Subtraction	Binary
-	Unary negation	Unary
--	Decrement 1	Unary
-=	Subtraction assignment	Binary
->	Member selection	Binary
->*	Pointer-to-member selection	Binary
/	Division	Binary
/=	Division assignment	Binary
<	Less than	Binary
<<	Left shift	Binary
<<=	Left shift assignment	Binary
<=	Less than or equal to	Binary

Operator	Name	Type
=	Assignment	Binary
==	Equality	Binary
>	Greater than	Binary
>=	Greater than or equal to	Binary
>>	Right shift	Binary
>>=	Right shift assignment	Binary
[ ]	Array subscript	—
^	Exclusive OR	Binary
^=	Exclusive OR assignment	Binary
|	Bitwise inclusive OR	Binary
|=	Bitwise inclusive OR assignment	Binary
||	Logical OR	Binary
~	One's complement	Unary
delete	Delete	—
new	New	—
conversion operators	conversion operators	Unary

```
Vector operator+(const Vector& a, const Vector& b)
{
     Return Vector(s.getx()+b.getx(),a.gety()+b.gety();

}
```

General Rule of Operator Overloading

The following rules constrain how overloaded operators are implemented. However, they do not apply to the new and delete operators, which are covered separately.

Rules:

- cannot define undefinable operator
- cannot redefine the meaning of operators when applied to built-in data types.
- **Overloaded operators must either be:**
  - **non-static class member function**
  - **a global function**
- it is impossible to redefine a operator for a built-in type(like `int`)
- A global function that needs access to private or protected class members must be declared as a friend
- You can only overload operators for your own (user-defined) classes
- every operator function you define must **implicitly have at least one argument** of a user-defined class type
- **Operators obey the precedence**, grouping, and number of operands dictated by their typical use with built-in types
-

Member or Non-member Functions

Global Function

Example:

ostream operator <<

```
#include <iostream>      /* File: vector0-op-add-os.cpp */
#include "vector0.h"
using namespace std;

ostream& operator<<(ostream& os, const Vector& a)
    { return (os << '('  << a.getx() << ", " << a.gety() << ')'); }

Vector operator+(const Vector& a, const Vector& b)
    { return Vector(a.getx() + b.getx(), a.gety() + b.gety()); }

int main()
```

```
{
    Vector a(1.1, 2.2);
    Vector b(3.3, 4.4);
    cout << "vector + vector: a + b = " << a + b << endl;
    cout << "vector + scalar: b + 1.0 = " << b + 1.0 << endl;
    cout << "scalar + vector: 8.2 + a = " << 8.2 + a << endl;
    return 0;
}
```

`cout << " a = " << a << "\n";` is equivalent to:

`operator<<(operator<<(operator<<(cout," a = "),a),"\n");`

This can only work if operator<< returns the ostream object itself.

Member function

- **Unary operators** declared as member functions take no arguments; if declared as global functions, they take one argument.
- **Binary operators** declared as member functions take one argument; if declared as global functions, they take two arguments.
- If an operator can be used as either a unary or a binary operator (**&**, *****, **+**, and **-**), you can overload each use separately.
- Overloaded operators **cannot have default arguments**
- All overloaded operators except assignment (**operator=**) are inherited by derived classes.

Overload Operator For Assignment (=)

The assignment operator (=) is, strictly speaking, a binary operator. Its declaration is identical to any other binary operator, with the following exceptions:

- It must be a non-static member function. No **operator=** can be declared as a nonmember function.
- It is not inherited by derived classes.
- A default **operator=** function can be generated by the compiler for class types, if none exists.

```
class Vector
{
  public:
    Vector(double a = 0, double b = 0) : x(a), y(b) { }
    const Vector& operator=(const Vector& b);
    //Right side of copy assignment is the argument.
  private:
    double x, y;
};

const Vector& Vector::operator=(const Vector& b)
{
    if (this != &b) // Avoid self-assignment to save time
    {
        x = b.x;
        y = b.y;
    }
    return *this; // Why return const Vector& ?
    // Assignment operator returns left side of assignment.
};
```

1. supplied argument is the right side of the expression, let's say a=b, b is the supplied argument.
2. returned value is the left hand side value, which enable the chain equal.

   a=b=c;

copy constructor and copy assignment

The copy assignment operator is not to be confused with the copy constructor. The latter is called during the construction of a new object from an existing one:

```
// Copy constructor is called--not overloaded copy assignment operator!
Point pt3 = pt1;

// The previous initialization is similar to the following:
Point pt4(pt1); // Copy constructor call.
```