# Overloading

Overloading refers to multiple meanings of the same name or symbol.

- Name overloading $\Longrightarrow$ overloaded function.
- Symbol overloading $\Longrightarrow$ overloaded operator.

# Operator Overloading Basic

## Operator

An operator is a symbol that tells the compiler to perform specific mathematical, logical manipulations, or some other special operation.

Example:

- arithmetic operator: + , −, ∗, /
- logical operator: && and ||
- pointer operator: & and ∗
- memory management operator: `new`, `delete[ ]`

A **binary operator** is an operator that takes two operands; a **unary operator** is one that takes one operands

## Operator overloading

Operator overloading refers to the multiple definitions of an operator.

Arithmetic operator such as + and / are already overloaded in C/C++ for different built-in types.

Example:

```
2 / 3      // integer division; result is 0

2.0 / 3.0  // floating-point division; result is 0.666667
```

For the same operator / , different algorithms are used to compute two types of divisions.

C++ allows most operators to be overloaded for user-defined types(classes).

The following operators can be overloaded:

```
new    new[]    delete   delete[]
+      -        *        /        %        ^        &
|      ~        !        =        <        >        +=
-=     *=       /=       %=       ^=       &=       |=
>>     <<       <<=      >>=      ==       !=       <=
>=     &&       ||       ++       --       ,        ->*
->     ()       []
```

The following can not be overloaded:

```
.    .*    ::    ?:    sizeof   typeid
```

## Why operator overloading?

Overloaded operators have appropriate meaning to user-defined types, so they can be used for these types.
e.g. to use operator + for adding two objects of a user-defined class.

An operator must be overloaded to be used on class objects.

However, there are two exceptions: operator = and operator &

Operator = and operator & are overloaded implicitly for every class, so they can be used for each class objects.

- operator = performs memberwise copy of the data members.
- operator & returns the address of the object in memory.

Example:

```
class C{
 public:
  C(): x(0), y(0) {}
  C(int xx, int yy): x(xx), y(yy) {}
 private:
  int x, y;
};
int main(){
  C c1, c2(5,6);
  C *ptr;
  c1 = c2;
  ptr = &c2;
}
```

## How to overload operators?

We can overload operators by writing special kinds of functions. These functions are called **operator functions**.

To overload operator `@`, the name of the operator function is `operator@`

These operator functions can be:

- class member functions, or
- stand-alone functions.

# Overload Operator as Class Member

Consider a binary operator @ ; `xobj` is an object of class `X` and `yobj` is of `Y`.

In order to use this @ as the following:

```
xobj @ yobj
```

we can have `operator@` as a member function in class `X`.

## Overloading operator +

To overload operator + for class `C` so that we can add two `C` objects with the result being another `C` object.

We declare a method named `operator+` in class `C`.

```
class C {
public:
   C operator+( const C& ) const;
   ...
};

C C::operator+( const C& c ) const {
   //  implementation of operator+
}
```

Now, we can invoke `operator+`, just like a regular class member function.

```
C  a, b, c;

a = b.operator+( c );
```

Since the keyword `operator`, this member function can, and normally would, be invoked as:

```
a = b + c;
```

Here, we add the `C` objects `b` and `c` to obtain another `C` object, which is then assigned to the `C` object `a`.

## Example:

```
#include <iostream>
using namespace std;

class C{
 public:
  void print();
  C operator+( const C& ) const;
  C() : x(0), y(0) {}
  C(int xx, int yy) : x(xx), y(yy) {}
 private:
  int x, y;
};

void C::print() const {
  cout << "x " << x << "y " << y << "\n";
}

C C::operator+( const C& c ) const{
  C tmp( x + c.x,  y + c.y );
  return tmp;
}

int main(){
  C c1( 2, 3 );
  C c2( 3, 4 );
  C result = c1 + c2;
  result.print();
}
```

## Example: A complex number class

A complex number is a number of the form

$$z = a + bi$$

where $i$ represents the square root of -1; $a$ is the real part of $z$ and $b$ is the imaginary part of $z$.

Arithmetic operations on complex numbers are defined as follows:

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

$$(a + bi) * (c + di) = (ac - bd) + (ad + bc)i$$

$$(a + bi)/(c + di) = (ac + bd)/(c^2 + d^2) + ((bc - ad)/(c^2 + d^2))i$$

Implement a class that represents complex numbers, overloads +,−, ∗, / to support complex arithmetic and overloads equal (==) and not equal (!=) operator to support complex number comparison.

```
#include <iostream>
using namespace std;

class Complex{
public:
  Complex();
  Complex( double );
  Complex( double, double );
  void print() const;
  Complex operator+( const Complex& ) const;
  Complex operator-( const Complex& ) const;
  Complex operator*( const Complex& ) const;
  Complex operator/( const Complex& ) const;
  bool operator==( const Complex& ) const;
  bool operator!=( const Complex& ) const;
private:
  double real;
  double imag;
};
Complex::Complex() {
  real = imag = 0.0;
}
Complex::Complex( double re ) {
  real = re;
  imag = 0.0;
}
Complex::Complex( double re, double im ) {
  real = re;
  imag = im;
}
void Complex::print() const {
  cout << real << " + " << imag << "i\n";
}
```

```
Complex Complex::operator+( const Complex& u ) const{
  Complex v( real + u.real,
             imag + u.imag );
  return v;
}

Complex Complex::operator-( const Complex& u ) const{
  Complex v( real - u.real,
             imag - u.imag );
  return v;
}

Complex Complex::operator*( const Complex& u ) const{
  Complex v( real * u.real - imag * u.imag,
             imag * u.real + real * u.imag );
  return v;
}

Complex Complex::operator/( const Complex& u ) const{
  double abs_sq = real * u.real + imag * u.imag;
  Complex v( ( real * u.real + imag * u.imag ) / abs_sq,
             ( imag * u.real - real * u.imag ) / abs_sq );
  return v;
}

bool Complex::operator==( const Complex& u ) const{
  return  (real == u.real && imag == u.imag) ;
}

bool Complex::operator!=( const Complex& u ) const{
  return  !(real == u.real && imag == u.imag) ;
}
```

## A simple test client:

```
int main(){
  Complex c1( 8.8, 0 );
  Complex c2( 3.1, -4.3 );
  Complex c3 = c1 + c2;
  Complex c4 = c2 - c1;

  c3.print();
  c4.print();
  if ( c3 == c4 )
    cout << "No way.";
  else
    cout << "Sure they are not equal.";
}
```

## Overloading operator =

- Operator = is used to copy each data member from the source object to the corresponding data member in the target object.

- If user does not overload operator = for a class. The compiler provides a default overloaded version that does the memberwise copying.

- The compiler's version is dangerous for classes whose data members include a pointer to dynamically allocated memory.

Note: the situation is similar to a class's copy constructor.

## Example:

```
class Vector{
public:
  Vector():size(0), ptr(0){cout << "default constructor" << endl; }
  Vector(int);
  Vector(const Vector&);
  Vector& operator=( const Vector& );
  // ...
private:
  int size;
  int* ptr;
};
Vector::Vector(int n){
  size = n;
  ptr = new int[size];
  for ( int i =0; i<size; i++)
    ptr[i] = 0;
  cout << "constructor Vector(n)" << endl;
}
Vector::Vector(const Vector& rhs){
  if( rhs.ptr != 0 ){
    size = rhs.size;
    ptr = new int[size];
    for (int i=0; i<size; i++)
      ptr[i] = rhs.ptr[i];
  }
  else{
    ptr = 0;
    size = 0;
  }
  cout << "copy constructor" << endl;
}
```

## Example:

```
// overload = for class Vector
Vector& Vector::operator=( const Vector& rhs ){
  if (this != &rhs){
    if ( rhs.ptr != 0 ){
      size = rhs.size;
      delete [] ptr;
      ptr = new int[size];
      for ( int i=0; i<size; i++ )
        ptr[i] = rhs.ptr[i];
    }
    else{
      size = 0;
      delete [] ptr;
      ptr = 0;
    }
  }
  cout << "assignment =" << endl;
  return *this;
}

int main(){
  Vector v1(5);
  Vector v2;
  v2 = v1;
  Vector v3 = v2;
}
```

**Note:**

- If we use a class member function to overload a binary operator, the member function has only one parameter.

- Similarly, if we use a class member function to overload a unary operator, the member function has no parameters.

# Overloading unary operator !

```
#include <iostream>
using namespace std;

class C{
public:
  void print() const;
  C operator!();        // unary operator; takes no argument
  C() : x(0), y(0) {}
  C(int xx, int yy) : x(xx), y(yy) {}
private:
  int x;
  int y;
};

void C::print() const {
  cout << "x " << x << "y " << y << "\n";
}

C C::operator!(){
  C tmp( -x, -y );
  return tmp;
}

int main(){
  C c1, c2( 2, 3 );
  c1 = !c2;
  c1.print();
  c2.print();
}
```

## Overloading the Increment and Decrement operators

- The operator ++ and −− have two forms : *pre* and *post*

```
int x = 6;
++x;  // preincrement
x++;  // postincrement
--x;  // predecrement
x--;  // postdecrement
```

- To overload the preincrement and predecrement operator, we use the declaration:

```
operator++();
operator--();
```

- To overload the postincrement and postdecrement operator, we include a single `int` parameter in the declaration:

```
operator++( int );
operator--( int );
```

  The `int` is used to distinguish the *post* from the *pre* form.

# Example

```
#include <iostream>
using namespace std;

class C{
public:
  void print() const;
  C operator++( );
  C operator++(int);
  C() : x(0), y(0) {}
  C(int xx, int yy) : x(xx), y(yy) {}
private:
  int x;
  int y;
};

void C::print() const {
  cout << "x " << x << "y " << y << "\n";
}

C C::operator++(){       // preincrement
  x++;
  y++;
  return *this;
}

C C::operator++(int n){  // postincrement
  C tmp = *this;
  x++;
  y++;
  return tmp;
}
```

## A simple test client:

```
int main(){
  C a(1,1), b(1, 1);
  C c;

  c = a++;
  a.print();
  c.print();  // x 1 y 1

  c = ++b;
  b.print();
  c.print();  // x 2 y 2
}
```

# Overload Operator as Stand-alone Function

Consider a binary operator @ ; `x` is an object of class `X` and `y` is of `Y`.

To use @ as

```
x @ y
```

we can overload operator@ as a stand alone function which takes two parameters: one of type `X` and one of type `Y`.

```
operator@ ( X, Y )
```

An operator that is overloaded as a stand-alone function *must* include a class object among its parameter list. ( why? )

## Example:

To overload operator + using a stand-alone function, we define the following:

```
C operator+( const C& c1, const C& c2){
   // ...
};
```

This stand-alone function `operator+`, has two parameters - the two `C` objects, and returns one `C` object.

Following the usual syntax for invoking a function, the `operator+` can be invoked as:

```
C  a, b, c;

a = operator+( b , c );
```

Since the keyword `operator`, this function can, and normally would, be invoked as:

```
 a = b + c;
```

Consider the following implementation for overloading using stand-alone function. Is there a problem?

```
class C{
public:
  void print() const;
  C operator+( const C& ) const;
  C() : x(0), y(0) {}
  C(int xx, int yy) : x(xx), y(yy) {}
private:
  int x;
  int y;
};


// overload operator + as stand-alone function

C operator+( const C& c1, const C& c2 ){
  C tmp( c1.x + c2.x,
         c1.y + c2.y );
  return tmp;
}
```

The `operator+` can not access private data member of class `C` !

## Solution 1:

```
#include <iostream>
using namespace std;

class C{
public:
  int getX() const{ return x; }
  int getY() const{ return y; }
  void print() const;
  C() : x(0), y(0) {}
  C(int xx, int yy) : x(xx), y(yy) {}
private:
  int x;
  int y;
};
void C::print() const {
  cout << "x " << x << "y " << y << "\n";
}

C operator+( const C& c1, const C& c2 ){
  C tmp( c1.getX() + c2.getX(),
         c1.getY() + c2.getY() );
  return tmp;
}

int main(){
  C c1( 2, 3 );
  C c2( 3, 4 );
  C result;
  result = c1 + c2;
  result.print();
  return 0;
}
```

## Solution 2: Use friend functions

```
#include <iostream>
using namespace std;

class C{
public:
  C() : x(0), y(0) {}
  C(int xx, int yy) : x(xx), y(yy) {}
  void print() const;
  friend C operator+( const C&, const C& );
private:
  int x;
  int y;
};
void C::print() const {
  cout << "x " << x << "y " << y << "\n";
}

//  as stand-alone friend
C operator+( const C& c1, const C& c2 ){
  C tmp( c1.x + c2.x,
         c1.y + c2.y );
  return tmp;
}

int main(){
  C c1( 2, 3 );
  C c2( 3, 4 );
  C result;
  result = c1 + c2;
  result.print();
  return 0;
}
```

## Operator functions: As class member v.s. As stand-alone

- Using class member functions, the overloaded operator is invoked as a member function on an object.

```
a = b + c;
a = b.operator+( c );
```

- Using stand-alone functions, the overloaded operator is invoked as a function that treats the two operands equally.

```
a = operator+( b , c );
```

- An operator intended to accept a basic type as its first operand can only be overloaded as stand alone function.

## Overloading the Input and Output operators

- Bitwise operator >> ( right shift ) and << ( left shift ) are built-in operators in C/C++.

- These two operators are overloaded in system library for formatted input and output of built-in types.

```
class ostream{
   //...
   ostream& operator<<( const char* );
   ostream& operator<<( const int );
   //...
};
```

- Since `cout` is an object of `ostream`, the following code

```
int i;
char* s;
//...
cout << i;
cout << s;
```

  can be interpreted as

```
cout.operator<<( i );
cout.operator<<( s );
```

- Again, << and >> can be further overloaded for user-defined types.

- Question: Do we overload << and >> as stand-alone function or class member function?

Example:

To overload >> to read into a `C` object as the following:

```
C c;
cin >> c;
```

we write a stand-alone function `operator>>` as

```
istream& operator>>( istream& in, C& c) {
      return in >> c.x >> c.y;
}
// as friend
```

Thus, the statement

```
cin >> c;
```

is now equivalent to

```
operator>>( cin, c );
```

which is evaluated as

```
cin >> c.x >> c.y;
```

# A modified complex number class

```
#include <iostream>
using namespace std;

class Complex{
public:
  Complex();
  Complex( double );
  Complex( double, double );
  friend Complex operator+( const Complex&, const Complex& );
  friend Complex operator-( const Complex&, const Complex& );
  friend Complex operator*( const Complex&, const Complex& );
  friend Complex operator/( const Complex&, const Complex& );
  friend bool operator==( const Complex&, const Complex& );
  friend bool operator!=( const Complex&, const Complex& );
  friend istream& operator>>( istream&, Complex& );
  friend ostream& operator<<( ostream&, const Complex& );
private:
  double real;
  double imag;
};

Complex::Complex() {
  real = imag = 0.0;
}
Complex::Complex( double re ) {
  real = re;
  imag = 0.0;
}
Complex::Complex( double re, double im ) {
  real = re;
  imag = im;
}
```

```
Complex operator+( const Complex& t, const Complex& u ){
  return Complex( t.real + u.real,
                  t.imag + u.imag );
}
Complex operator-( const Complex& t, const Complex& u ){
  return Complex( t.real - u.real,
                  t.imag - u.imag );
}
Complex operator*( const Complex& t, const Complex& u ){
  return Complex( t.real * u.real - t.imag * u.imag,
                  t.imag * u.real + t.real * u.imag );
}
Complex operator/( const Complex& t, const Complex& u ){
  double abs_sq = t.real * u.real + t.imag * u.imag;
  return Complex( ( t.real * u.real + t.imag * u.imag ) / abs_sq,
              ( t.imag * u.real - t.real * u.imag ) / abs_sq );
}
bool operator==( const Complex& t, const Complex& u ){
  return  ( t.real == u.real && t.imag == u.imag) ;
}
bool operator!=( const Complex& t, const Complex& u ){
  return  !( t.real == u.real && t.imag == u.imag) ;
}
istream& operator>>( istream& in, Complex& c ){
  return in >> c.real >> c.imag ;
}
ostream& operator<<( ostream& out, const Complex& c ){
  return out << c.real << " + " << c.imag << "i\n";
}
```

## A simple test client:

```
int main(){
  Complex c1, c2;

  cin >> c1 >> c2;
  cout << c1 << c2;
  cout << c1 + c2;

  return 0;
}
```

## Note

- The precedence of any operator can not be changed.
- The number of operands required by the operator can not be changed.

  Example:

```
class C{
  C operator+(); // error! + is a binary operator
  // ...
};
```