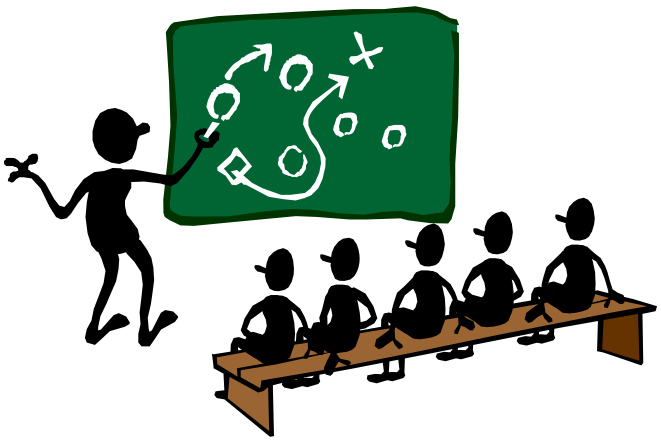
# C++ Programming Language
# Chapter 8
# Operator Overloading and Friends




*Juinn-Dar Huang*

*Associate Professor*

*jdhuang@mail.nctu.edu.tw*


*April 2011*

# Learning Objectives

- Why operator overloading?
- Operator overloading
  - binary operators
  - unary operators
- Friend functions and classes
- More about overloading
  - operators: << and >>
  - operators: ++ and --
  - operators: [] and ()

# Operators Are Functions!

- Operators ➔ +, -, *, %, ==, !=, [ ], and so on
  - they are actually functions!
- They are just invoked using different syntax

```
int x = 5;
int y = x + 7;
```

  - + is a binary operator with two operands (x & 7 in this case)
  - humans are used to this notation
- Think of + in another way ➔ Treat it as a function call!

```
int y = +(x, 7);
```

  - + is regarded as a function name
  - x & 7 are two arguments
  - function + returns sum of its two arguments

# Why Operator Overloading? (1/4)

- Operators for built-in types
  - e.g., +, -, *, /, =, %, ==, … for int
  - all operators are properly defined for C++ built-in types
- However, for user-defined types

```
class complex {
    double re, im;
public:
    complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }
};
void f() {
  complex a(1, 1), b(2, 2), c;
  c = a + b;    // error! compiler does not know how to + two complexes!
}
```

# Why Operator Overloading? (2/4)

- Define a member function add

```
class complex {
   double re, im;
public:
   complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }
   const complex add(const complex&) const;
};
const complex complex::add(const complex& rhs) const {
  complex result = rhs;    // using copy ctor, automatically generated by compiler
  result.re += re; result.im += im;  // using built-in += of type double
  return result;
}
void f() {
  complex a(1, 1), b(2, 2), c;
  c = a.add(b);    // ok! using a member function add for addition
}
```

using **assignment operator**, automatically generated by compiler

**Ugly here! Not elegant at all! Any other way to do this?**



Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

# Why Operator Overloading? (3/4)

- Define a member function operator+

```
class complex {
   double re, im;
public:
   complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }
   const complex operator+(const complex&) const;
};
const complex complex::operator+(const complex& rhs) const {
  complex result(rhs);     // using copy ctor, too
  result.re += re; result.im += im;
  return result; }
void f() {
  complex a(1, 1), b(2, 2), c;
  c = a.operator+(b);      // ok! explicit call, just ugly!
  c = a + b;               // ok! it is just a shorthand for operator+
}
```

**That's exactly what we use in math class!**

# Why Operator Overloading? (4/4)

- Allow you define operators for user-defined types!
- Provide a more conventional and convenient notation for manipulating user-defined objects



# Another Way for Operator Overloading (1/2)

- Overloaded operators are **NOT** necessarily member functions!

```
class complex {
    double re, im;
public:
    complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }
    double real() const { return re; }
    double image() const { return im; }
};
const complex operator+(const complex& lhs, const complex& rhs) {
  double real, image;
  real = lhs.real() + rhs.real(); image = lhs.image() + rhs.image();
  return complex(real, image);}
```

# Another Way for Operator Overloading (2/2)

```
void f() {
  complex a(1, 1), b(2, 2), c;
  c = operator+(a, b);      // ok! explicit call, just ugly!
  c = a + b;                // ok! it is just a shorthand for operator+
}
```

- Note that operator+ is
  - an overloaded nonmember function
    - overload the function (operator) “+”
  - not a member function of type complex

- operator+ vs. operator+
  - which one is better?
  - discuss later (Page 17, 18)

# Limitations of Operator Overloading

- You cannot invent a whole new operator
  - e.g., defining a new operator ** is not allowed
- You cannot define a unary % or a ternary +
- The same precedence and associativity still hold
  - b = b + c * a; ➔ (b = ( b + (c * a)));
    even a,b,c are of type complex

# Operators Can Be Overloaded

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

42 (actually 48) operators can be overloaded in C++

# Operator Overloading (Comparisons)

- Equality operator, ==

```
bool operator==(const complex& lhs, const complex& rhs) {
  return ( lhs.real() == rhs.real() && lhs.image() == rhs.imagel() ) ;
}   // == : equality operator of type double
```

- Inequality operator, !=

```
bool operator!=(const complex& lhs, const complex& rhs) {
  return ! (lhs == rhs);
}   // == : equality operator of type complex
```

- Similarly, followings can be used (if applicable)
  - > vs. <=
  - < vs. >=



# Operator Overloading (Assignments) (1/3)

- Assignment operator, =

```
complex& complex::operator=(const complex& rhs) {
  re = rhs.re; im = rhs.im ; return *this;
}   // = : assignment operator of type double
```

- C++ standard specifies that an overloaded assignment operator **MUST** be a non-static member function
- Conventionally,
  - **return type** of assignment operator of class **X** ➔ **X&**
  - assignment operator returns *this **(referring to the calling object)**
  - why? ➔ make user-defined type behave like a built-in type

```
int i;
(i = 3) = 4;     // this line is legal
// i = 4
```

```
complex a(2,2), b(3,3), c(4,4);
(a = b) = c;        // legal, too
// a = ?
```

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

# Operator Overloading (Assignments) (2/3)

- Operator +=

```
complex& complex::operator+=(const complex& rhs) {
  re += rhs.re; im += rhs.im ; return *this;
}   // += of type double
```

- Conventionally,
  - op= is usually a non-static member function
  - return type of op= of class X ➔ X&
  - op= returns *this (referring to the calling object, in Chap 10)
  - why? ➔ again, make user-defined type behave like a built-in type

```
int i = 2;
(i += 3) = 4;   // this line is legal
// i = 4
```

```
complex a(2,2), b(3,3), c(4,4);
(a += b) = c;       // legal, too
// a = ?
```

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

# Operator Overloading (Assignments) (3/3)

- Typically, using op= to implement operator op
- Implement operator + using operator +=

```
const complex operator+(const complex& lhs, const complex& rhs) {
  complex result(lhs);     // copy ctor
  return ( result += rhs );
}
```

- Make op= right ➔ automatically make op right!
  - improve maintainability
  - no needs for access member functions ( real() and image() )
- For binary arithmetic operators +, - , *, / , %, ^, &, |, <<, >> of class X
  - conventionally, return type of them are usually **const X**
  - why? See next slide


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

# Returning Constant Value

- In the previous slide,

  **const complex** operator+(const complex& lhs , const complex& rhs)

- What if?

  **complex** operator+(const complex& lhs , const complex& rhs)

```
void f() {
    complex a(1,1), b(2,2), c(3,3);
    (a + b) = c;    // no error if using red one; error if using blue one
    if((a+b) = c)   // Oops, programmer actually wants ➔ if((a+b) == c)
      do_things     // again, no error if using red one; error if using blue one
}
```

- Hence, blue one is preferred

Juinn-Dar Huang

jdhuang@mail.nctu.edu.tw

# Overloading Unary Operators

- Unary minus operator, -

```
const complex complex::operator-() const          // NO parameter !
{ return complex(-re, -im); }
```

- For a binary operator a @ b
  - member function ➔ a.operator@(b) ; 1 parameter
  - nonmember function ➔ operator@(a, b) ; 2 parameters
- For a prefix unary operator @a
  - member function ➔ a.operator@() ; no parameter
  - nonmember function ➔ operator@(a) ; 1 parameter
- For a postfix unary operator a@
  - member function ➔ a.operator@(int) ; 1 parameter
  - nonmember function ➔ operator@(a, int) ; 2 parameters
  - yes, we are talking about ++, -- ; discuss later (Page 32 ~ 36)


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

# Member vs. Nonmember Operators (1/2)

- Member version

```
const complex complex::operator+(const complex& rhs) const {
    complex result(rhs);
    result.re += re; result.im += im;
    return result;
}
```

- Nonmember version

```
const complex operator+(const complex& lhs, const complex& rhs) {
    double real = lhs.real() + rhs.real();     // need real() to ger re
    double image = lhs.image() + rhs.image();  // need image() to get im
    return complex(real, image);
}
```

- It seems member version is better. However, …

# Member vs. Nonmember Operators (2/2)

- If mixed-mode arithmetic is allowed
  - e.g., allow adding a complex with a double

```
void f() { // operator+ is a member function here
    complex a(1,1), b;
    b = a + 1.0;   // ok! a.operator+( complex(1.0) )
    b = 1.0 + a;   // error! 1.0.operator+(a) ← no such function!
}
void f() { // operator+ is a nonmember function here
    complex a(1,1), b;
    b = a + 1.0;   // ok! operator+( a, complex(1.0) )
    b = 1.0 + a;   // ok! operator+( complex(1.0), a )
}
```

**implicit type conversion using ctor**

- In general, nonmember version is preferred
  - how about efficiency? Discuss later (Page 20)

# Never Overload "&&", "||", and ","

- &&, ||, and comma operator (,)
  - built-in versions work for bool type
  - recall: compiler uses short-circuit evaluation
  - when overloaded, no short-circuit anymore
    - use complete evaluation instead
  - contrary to users' expectations ➔ Avoid doing this!
- Comma operator (,)
  - when overloaded, left-to-right evaluation is NOT guaranteed
  - contrary to users' expectations ➔ Avoid doing this!
- You should never overload these operators!

# Friend Functions

- Nonmember functions
  - access private members through accessors and mutators
  - inefficient (overhead of calls to accessors and mutators)
  - e.g., operator is overloaded as nonmember function
- Friend functions can directly access private members
  - same access privilege as member functions
  - no calls to accessors and mutators ➔ more efficient
- You can make specific nonmember functions friends for better efficiency!
  - make friends judiciously!


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

# Friend Functions (1/2)

- Use keyword friend in front of function declaration
  - specified inside class definition
  - but it is NOT a member function!

```
class complex {
    double re, im;
public:
    complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }
    double real() const { return re; }
    double image() const { return im; }
    friend const complex operator+(const complex&, const complex&);
};
const complex operator-(const complex&, const complex&);
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

# Friend Functions (2/2)

```
// no need to add friend prefix in function definition
const complex operator+(const complex& lhs, const complex& rhs) {
    complex result(lhs);
    result.re += rhs.re;    result.im += rhs.im;
    return result;
}   // a friend function has same access privilege as member functions
```

**vs.**

```
const complex operator-(const complex& lhs, const complex& rhs)
    double real = lhs.real() + rhs.real();
    double image = lhs.image() + rhs.image();
    return complex(real, image);
}   // need accessors to get private data
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

# Friend Function Uses

- Most common use: nonmember operators
  - an operator is designed as a nonmember for a reason
    - e.g., mixed-mode arithmetic
  - a nonmember operator still has to access private members
  - so it is natural to make it a friend to improve efficiency
    - avoids calls to accessors/mutators
- Friend functions are not necessarily nonmember operators
  - they can be any nonmember functions



Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

# Friends vs. OOP

- Though friend functions are not member functions, they are still part of class design
- Yes, friend functions are inherently dangerous
  - so you have to make friends with extreme care!
- Encapsulation can still be achieved
  - friends are declared INSIDE class definition
- If you use friends properly, you can get all of
  - convenience (explicit type conversions)
  - efficiency
  - encapsulation


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

# More about Friends

- A friend declaration can be placed in either private or public part of class definition
  - does not matter; same effect
  - remember that friends are NOT member functions
- A member function of class A can be a friend of class B

```
class A {
public:
    void func(int, const A&) const;
    // …
};
class B {
    friend void A::func(int, const A&) const;
    // …
};
```


Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

# Friend Classes

- Make a class X a friend of class Y
  - make all member functions of class X friends of class Y

```
class X {
    void f1();
public:
    void f2();
};

class Y {
    friend class X;        // make both X::f1() and X::f2() friends of Y
    // …
};
```

# I/O of User-Defined Types (<< , >>)

```
class complex {
public:
    void output() const {  cout << re << '+' << im << 'i' ;  }
    // …
};
void f() {
    complex a(3,4);
    a.output();     // ok, but ugly…
    int i = 3; char ch = 'a'; double d = 3.0;
    cout << i << ch << d << endl;
    cout << a;      // error!
}
```

- Is it possible to make "cout << a" work?
  - make class complex more like built-in types

# Behind the Scene

```
void f() {
    int i = 3; char ch = 'a'; double d = 3.0;
    cout << i << ch << d;
}
```

Step 0: cout is a predefined object of type ostream (output stream)

Step 1: a **binary** operator ostream& operator<<(ostream&, const int&) is called, which takes 2 arguments cout and i, and returns cout

Step 2: a **binary** operator ostream& operator<<(ostream&, const char&) is called, which takes 2 arguments cout and ch, and returns cout

Step 3: a **binary** operator ostream& operator<<(ostream&, const double&) is called, which takes 2 arguments cout and d, and returns cout



Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

# Overload <<

- Want to make "cout << complex(1,1)" work?

```
ostream& operator<<(ostream& os, const complex& rhs) {
    os << rhs.real() << '+' << rhs.image() << 'i' ;
    return os;
}
```

```
ostream& operator<<(ostream&, const double&)
ostream& operator<<(ostream&, const char&)
```

```
void f() {
    complex a(2,3), b(4,5);
    cout << a << endl << b << endl;  // more elegant!
}
```

- It is common to make operator<<  a friend

# Return Value of Operator <<

- Why return the 1st argument as return value?
  - that is why you can do things like

```
int i = 3; char ch = 'a'; double d = 3.0;
cout << i << ch << d;
```

- If you make operator>> return void …

```
void operator<<(ostream& os, const complex& rhs) {
    os << rhs.real() << '+' << rhs.image() << 'i' ;
}

void f() {
    complex a(2,3), b(4,5);
    cout << a << endl << b << endl;  // compilation error!
}      void
```

# Overload >>

- Similarly, forget about input()
- You can use “cin >>” for user-defined types
  - first, make istream& operator>>(istream&, complex&) a friend
  - then,

```
istream& operator>>(istream& is, complex& rhs) {
    is >> rhs.re >> rhs.im ;
    return is;
}
```

istream& operator>>(istream&, **double**&)

```
void f() {
    complex a, b;
    cin >> a >> b;  // more elegant!
}
```

cin is a predefined object of type **istream (input stream)**

# Increment/Decrement Operators (1/6)

- Both unary operators can be prefix or postfix
  - prefix: ++x, --x
  - postfix: x++, x--
- You might not know …

```
void f() {
    int i = 10;
    ++i;            // i = 11
    i++;            // i = 12
    ++++i;          // i = 14
    i++++;          // error! why? Is there a good reason to make this illegal?
}
```

# Increment/Decrement Operators (2/6)

- All prefix/postfix increment/decrement operators need lvalues
  - ++i ➔ ok ; i is a variable of int, which can appear on the left-hand side of assignment operator
  - ++5 ➔ error; 5 is not even a variable, which surely cannot appear on the left-hand side of assignment operator
  - so non-static member function is typically used for overloading here

Moreover,

- Prefix increment/decrement operators return lvalues
- Postfix increment/decrement operators don't
  - they return constant objects instead!

# Increment/Decrement Operators (3/6)

```
class LLint {        // class for long precision integer
public:
    LLint(); LLint(int);              // ctors
    LLint& operator++();              // prefix ++
    const LLint operator++(int);      // postfix ++, int is just a marker
    LLint& operator--();              // prefix --
    const LLint operator--(int);      // postfix --, int is just a marker
    // +, -, * , /, %, =, +=, …
};
LLint& LLint::operator++() {          // prefix ++
    *this += 1;
    return *this; }                   // no call to copy ctor!
const LLint LLint::operator++(int) {          // postfix ++
    LLint old(*this);                 // invoke copy ctor
    ++(*this);                        // invoke prefix ++
    return old; }                     // invoke copy ctor
```

**Prefix is more efficient than postfix !**

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

# Increment/Decrement Operators (4/6)

```
void f() {
    LLint i = 10, j;
    ++i;          // i = 11, ➔ i.operator++()
    i++;          // i = 12, ➔ i.operator++(0)
    ++++i;        // i = 14,➔ (i.operator++()).operator++()
    i++++;        // error!
}
```

**LLint acts just like int !**

**That is what we try to achieve ➔**
**Make user-defined types work like built-in types !**

# Increment/Decrement Operators (5/6)

- What if the return type of postfix ++ changes from const LLint to plain LLint ?
  - then, i++++ becomes legal now !
  - however, the outcome may surprise you !
  - very **BAD** idea to do this !

```
void f() {
    LLint i = 10, j;
    ++i;           // i = 11
    i++;           // i = 12
    ++++i;         //  i = 14
    i++++;         // ok now! ➔ (i.operator++(0)).operator++(0)
    j = i;         // i = ? , j = ?
}
```

**We use ++ for discussions**
**Similar discussions for --**

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

# Increment/Decrement Operators (6/6)

Summary

- Both unary ++/-- operators can be prefix or postfix
- Conventions
  - they are overlodaed by non-static member functions
    - they all need lvalues
  - prefix ++/-- return *this
  - postfix ++/-- return const object
- Typically, prefix is more efficient than postfix
  - prefer prefix ++/-- to postfix ones whenever possible
- Implement postfix ++/-- in terms of prefix ++/--
  - improve maintainability

    ( similar to ➔ implementing + in terms of += )

# Overload Subscript Operator [ ] (1/2)

- It must be a non-static member function
- It usually returns a reference

Example:

```
void f() {
    int ir[100], x = 8;
    ir[x] = 12;
    x = 108;
    ir[x] = 16;       // no compilation error, likely to be a runtime disaster!
}
// Is it possible to perform runtime range checking?
```

# Overload Subscript Operator [ ] (2/2)

```
class Iarr100 {
    int arr[100];
public:
    int& operator[](int);   };

int& Iarr100::operator[](int index) {
    if ( (index < 0) || (index > 100) ) { cerr << "out of range!\n"; exit(1); }
    return arr[index] ; }

void f() {
    Iarr100 ir;   int x = 8;
    ir[x] = 12;      // ok, ➔ ir.operator[](x) = 12
    x = 108;
    ir[x] = 16;      // no compilation error either; but issue a runtime error !
}
```

# Overload Function Call Operator () (1/3)

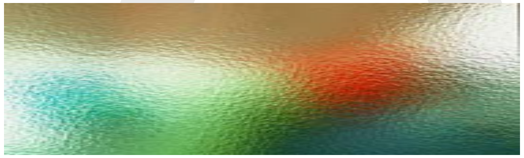
- It must be a non-static member function
- Common uses
  - make objects behave like functions
    ➔ function object or **functor** (see next slide)
  - implement subscript operator for multidimensional arrays
    - e.g.,

```
Int3Darr arr(7, 8, 9);      // arr is a 3-dimensional int array object
arr(2, 3, 5) = 100;
```

# Overload Function Call Operator () (2/3)

- It must be a non-static member function
- Example:

```
void f(int i, int j) {
```



```
    if ( lessthan(i, j) )        // Is lessthan a function returning bool ?
        cout << i << " is less than " << j << endl;
}
```

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

# Overload Function Call Operator () (3/3)

```
class LessThan {
public:
    bool operator()(int lhs, int rhs) const { return lhs < rhs ; }
};  // LessThan does not even have a data member!

void f(int i, int j) {

LessThan lessthan;          // lessthan is an object of class LessThan

    if ( lessthan(i, j) )     // ➔ lessthan.operator()(i, j)
        cout << i << " is less than " << j << endl;
}   // here, lessthan is an example of functor !
```

- Notice that, unlike other overloaded operators, operator() can have arbitrary number (0, 1, 2, …) of parameters

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

# Summary (1/2)

- Operators are actually just functions
- C++ built-in operators can be overloaded
  - ways to define proper operators for user-defined types
- Operator overloading
  - cannot invent new operators
  - follow same grammar (#operands, precedence, associativity)
- An overloaded operator can be
  - non-static member function
  - nonmember function
  - which one is better? ➔ case by case
- Understand the differences between op and op=
- Never overload && || ,

# Summary (2/2)

- Friend functions (Friend classes)
  - nonmember functions but can access private members
- Make friends judiciously
  - efficiency and encapsulation
- I/O of user-defined types: <<, >>
- Prefix/Postfix increment/decrement operators: ++ --
- Subscript operator [ ]
- Function call operator ()
  - functor
- While overloading operators ➔ follow the conventions!



Juinn-Dar Huang jdhuang@mail.nctu.edu.tw