

# UEE1303

# Objective-Oriented Programming

---

C++\_Lecture 06:  
Operator Overloading

**C: How to Program 8<sup>th</sup> ed.**

# Agenda

- Fundamentals of Operator Overloading (chapter 18.3)
- Restrictions on Operator Overloading (chapter 18.3)
- Operator Functions as Class Members vs. Global Functions (chapter 18.4)
- Overloading Stream Insertion (<<) and Stream Extraction (>>) Operators (chapter 18.5)
- Overloading Unary Operators (chapter 18.6)
- Overloading Binary Operators (chapter 18.4)
- Case Study: Array Class (chapter 18.10)
- Converting between Types (chapter 18.11)
- Overloading ++ and -- (chapter 18.7)

# Fundamentals of Operator Overloading

- Operators are overloaded in C/C++
  - $+7$ ,  $2+5$ ,  $3.25+7.3$
- In addition to overloading, compilers often need to perform coercion or casting when the  $+$  symbol is used with mixed arithmetic
- To use arithmetic symbols with our own objects  $\Rightarrow$  must overload the symbols
  - **Polymorphism** allows the same operations to be carried out differently
  - Overload the  $+$  operator with a reasonable meaning

# A Starting Example

```
class CComplex {  
    double real, imag;  
public:  
    CComplex() { real = 0; imag = 0; }  
    CComplex(double r, double i) {  
        real = r; imag = i; }  
    CComplex cadd(CComplex & o2);  
    void display() { cout << "(" << real  
        << "," << imag << "i)" << endl }  
};  
CComplex CComplex::cadd(CComplex & o2) {  
    CComplex c; c.real = real + o2.real;  
    c.imag = imag + o2.imag; return c;  
}
```

## A Starting Example (cont.)

```
int main() {  
    CComplex c1(3,4), c2(2,-7), c3;  
    c3 = c1.cadd(c2);  
    cout << "c1 = "; c1.display();  
    cout << "c2 = "; c2.display();  
    cout << "c1+c2 = "; c3.display();  
    return 0;  
}
```

```
c1 = (3,4i)  
c2 = (2,-7i)  
c1+c2 = (5,-3i)
```

- Using member function is cumbersome
  - good to have  $c3 = c1 + c2$

# Restrictions on Operator Overloading

- If an operator is normally defined to be unary only, then you cannot overload it to be binary
  - cannot change associativity or precedence
- Operators cannot be overloaded for built-in datatypes
  - The meaning of how an operator works on fundamental types cannot be changed by operator overloading.
    - You cannot, for example, change the meaning of how + adds two integers.
- You also cannot overload operators that you invent

## Restrictions on Operator Overloading (cont.)

- Overloading an assignment operator and an addition operator to allow statements like
  - `object2 = object2 + object1;`
- does not imply that the `+=` operator is also overloaded to allow statements such as
  - `object2 += object1;`
- Such behavior can be achieved only by explicitly overloading operator `+=` for that class.

# List of Overloading Operators

- Arithmetic
  - `+`, `-`, `*`, `/`, `%`
- Bitwise
  - `^`, `&`, `|`, `~`, `>>`, `<<`
- Correlational
  - `<`, `<=`, `>`, `>=`, `!=`, `==`
- Logic
  - `!`, `&&`, `||`
- Assignment
  - `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`
- Other
  - `++`, `--`, `[]`, `()`, `->`, `new`, `new []`, `delete`, `delete []`



# List of Overloading Operators (cont.)

- Four operators cannot be overloaded

operator	usual use
. (dot operator)	member
. *	pointer to member
::	scope resolution
? :	conditional

# Operator Functions as Class Members vs. Global Functions

- Operator functions can be member functions or global functions.
- Member Function
  - use the `this` pointer implicitly to obtain one of their class object arguments (the `left` operand for binary operators).
- Global Function
  - Arguments for both operands of a binary operator must be explicitly listed in a global function call.
  - Global functions are often made `friends` for performance reasons.

# Operator Functions as Class Members

- When overloading `()`, `[]`, `->` or any of the `assignment operators`, the operator overloading function must be declared as a **class member**.
- For the other operators, the operator overloading functions can be class members or standalone functions.

# Operators as Class Members and Global Functions

- When an operator function is implemented as a member function, the **leftmost** (or only) **operand** must be an object (or a reference to an object) of the operator's class.

```
Complex a, b, c;
```

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

- If the **left operand** must be **an object of a different class** or a **fundamental type**, this operator function must be implemented as a **global function** (as we'll do with `<<` and `>>`).
- A global operator function can be made a **friend** of a class if that function must access private or protected members of that class directly.

# Member vs. Friend Functions

- Implement the overloading operation as a member function
  - use `this` to visit the data member
  - The **left** operand must be a object of same class, ex: **c1**+**c2**
- What if `c3 = c1 + i` ?

```
CComplex CComplex::operator+(int& i) {  
    return CComplex(real+i,imag); }
```

- What if `c3 = i + c2` ?  $\Rightarrow$  use a friend function

```
CComplex operator+(int& i, CComplex& c) {  
    return CComplex(i+c.real,c.imag); }
```

# Overloading Stream Insertion (<<) and Stream Extraction (>>) Operators

- You can input and output fundamental-type data using the stream extraction operator >> and the stream insertion operator <<.
- The C++ class libraries overload these operators to process each fundamental type, including pointers and C-style `char *` strings.
- You can also overload these operators to perform input and output for your own types.
  - `cin >> myObject;`
  - `cout << myObject;`  
Instead of need for:  
`myObject.output();`

# Example

- The following program overloads >> and << operators to input and output PhoneNumber objects in format “(ooo) ooo-oooo”.

# PhoneNumber.h

```
// Fig. 18.3: PhoneNumber.h
// PhoneNumber class definition
#ifndef PHONENUMBER_H
#define PHONENUMBER_H
#include <iostream>
#include <string>
using namespace std;
class PhoneNumber
{
    friend ostream &operator<<(ostream &, const
PhoneNumber &);
    friend istream &operator>>(istream &, PhoneNumber &);
private:
    string areaCode; // 3-digit area code
    string exchange; // 3-digit exchange
    string line;      // 4-digit line
}; // end class SalesPerson
#endif
```



# PhoneNumber.cpp

```
// Fig. 18.4: PhoneNumber.cpp
#include <iomanip>
#include "PhoneNumber.h"
using namespace std;

// overloaded stream insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<(ostream &output, const PhoneNumber
&number)
{
    output << "(" << number.areaCode << ")"
        << number.exchange << "-" << number.line;
    return output; //enables cout << a << b << c;
}
```

# PhoneNumber.cpp (cont.)

```
// overloaded stream extraction operator; cannot be
// a member function if we would like to invoke it with
// cin >> somePhoneNumber;
istream &operator>>(istream &input, PhoneNumber &number)
{
    input.ignore(); // skip (
    input >> setw(3) >> number.areaCode;
    input.ignore(2); // skip ) and space
    input >> setw(3) >> number.exchange;
    input.ignore(); // skip (-)
    input >> setw(4) >> number.line;

    return input; //enables cin >> a >> b >> c;
}
```

# Fig19\_05.cpp

```
// Fig. 18.5: fig18_05.cpp
// Demonstrating class PhoneNumber's overloaded stream
// insertion and stream extraction operators.
#include <iostream>
#include "PhoneNumber.h"
using namespace std;
int main()
{
    PhoneNumber phone; // create object phone
    cout << "Enter phone number in the format (123) 456-
7890: " << endl;
    // cin >> phone invokes operator>> by implicitly
    // issuing the global function call
    // operator>>(cin, phone)
    cin >> phone;
    cout << "The phone number entered was: ";
    cout << phone << endl;
    return 0;
}
```

# Overloading >>

- Operands >>
  - `cin` object, of class type `istream`
  - our class type (`PhoneNumber`)
  - `cin >> phone;`
  - `operator>>(cin, phone)`
- Function `operator>>` returns `istream` reference input (i.e., `cin`).
  - Enables input operations on `PhoneNumber` objects to be cascaded with input operations on other `PhoneNumber` objects or on objects of other datatype.

`cin >> phone1 >> phone2;`



`cin >> phone2;`

# Overloading Unary Operators

- We overload unary operator **!** To test whether an object of a `String` class we create is empty and return a `bool` result.
- A unary operator for a class can be overloaded
  - as a non-static member function with **no** arguments

```
class String {  
public:  
    bool operator!() const;  
};
```

- as global function with **one** argument that must be an object (or a reference to an object) of the class

```
bool operator!(const String&);
```

# Overloading Binary Operators

- We overload `<` to compare two `String` objects.
- A binary operator for a class can be overloaded
  - as a non-static member function with **one** arguments

• `y < z;`       $\Rightarrow$     `y.operator<(z);`

```
class String {  
public:  
    bool operator<(const String &) const;  
};
```

- as global function with **two** arguments – one of which must be an object (or a reference to an object) of the class

```
bool operator<(const String&, const String&);
```

# Case Study: Array Class

- Pointer-based arrays have many problems
  - A program can easily “walk off” either end of an array, because C++ does not check whether subscripts fall outside the range of an array.
  - Arrays of size  $n$  must number their elements  $0, \dots, n - 1$ ; alternate subscript ranges are not allowed.
  - An entire array cannot be input or output at once.
  - Two arrays cannot be meaningfully compared with equality or relational operators.
  - When an array is passed to a general-purpose function designed to handle arrays of any size, the array’s size must be passed as an additional argument.
  - One array cannot be assigned to another with the assignment operator.

## Case Study: Array Class (cont.)

- In this example, we create a powerful `Array` class:
  - Performs range checking.
  - Allows one array object to be assigned to another with the assignment operator.
  - Objects know their own size.
  - Input or output entire arrays with the stream extraction and stream insertion operators, respectively.
  - Can compare Arrays with the equality operators `==` and `!=`.



# Array.h

```
// Fig. 18.10: Array.h
// Array class definition with overloaded operators
#ifndef ARRAY_H
#define ARRAY_H
class Array
{
    friend ostream &operator<<(ostream &, const Array &);
    friend istream &operator>>(istream &, Array &);
public:
    Array( int = 10);    // default constructor
    Array (const Array &); // copy constructor
    ~Array();           // destructor
    int getSize() const; // return size
    const Array &operator=(const Array &); // assignment
    bool operator==(const Array &); const // equality
}
```

# Array.h (cont.)

```
bool operator!=( const Array &right) const
{
    // invokes Array::operator==
    return !(*this == right);
}
// subscript operator for non-const objects
// returns modifiable lvalue
int &operator[]( int );
// subscript operator for const objects returns rvalue
int operator[]( int ) const;
private:
    int size; // pointer-based array size
    // pointer to first element of pointer-based array
    int *ptr;
}; // end class Array
#endif
```

# Array.cpp

```
// Fig. 18.11: Array.cpp
#include <iostream>
#include <iomanip>
#include <cstdlib>    // exit function prototype
#include "Array.h"
using namespace std;

Array::Array(int arraySize) { // constructor
    size = (arraySize > 0 ? arraySize : 10);
    ptr = new int[size];
    for (int i = 0; i < size; i++)
        ptr[i] = 0;
}

// copy constructor
Array::Array(const Array &arrayToCopy)
    :size(arrayToCopy.size) {
    ptr = new int[size];
    for (int i = 0; i < size; i++)
        ptr[i] = arrayToCopy.ptr[i];
}
```

## Array.cpp (cont.)

```
Array::~~Array() { // destructor
    delete [] ptr;
}
// return number of elements of Array
int Array::getSize() const
{
    return size;
}
// determine if two Arrays are equal and return true
bool Array::operator==( const Array &right) const
{
    if (size != right.size)
        return false;
    for (int i = 0; i < size; i++)
        if ( ptr[i] != right.ptr[i] )
            return false;
    return true;
}
```

## Array.cpp (cont.)

```
// overloaded assignment operator;
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right)
{
    if ( &right != this ) // avoid self-assignment
    {
        // for Arrays of different sizes, deallocate
        // original left-side array, then allocate new
        // left-side array
        if ( size != right.size)
        {
            delete [] ptr;      // release space
            size = right.size; // resize this object
            ptr = new int[size];
        }
        for (int i = 0; i < size; i++)
            ptr[i] = right.ptr[i];
        return *this; // enable x = y = z;
    }
}
```

## Array.cpp (cont.)

```
// overloaded subscript operator for non-const Arrays;  
// reference return creates a modifiable lvalue  
int &Array::operator[]( int subscript)  
{  
    // check for subscript out-of-range error  
    if ( subscript < 0 || subscript >= size)  
    {  
        cerr << "\nError: Subscript " << subscript  
              << " out of range" << endl;  
        exit(1); // terminate program;  
    }  
    return ptr[subscript]; // reference return  
}
```

## Array.cpp (cont.)

```
// overloaded subscript operator for const Arrays;  
// const reference return creates an rvalue  
int Array::operator[](int subscript) const  
{  
    // check for subscript out-of-range error  
    if ( subscript < 0 || subscript >= size)  
    {  
        cerr << "\nError: Subscript " << subscript  
              << " out of range" << endl;  
        exit(1); // terminate program;  
    }  
    return ptr[subscript]; // returns copy of this element  
}
```

## Array.cpp (cont.)

```
// overloaded input operator for class Arrays;
// inputs values for entire Array
istream &operator>>( istream &input, Array & a) {
    for ( int i = 0; i < a.size; i++)
        input >> a.ptr[i];
    return input; // enables cin >> x >> y >> z;
}

// overloaded output operator for class Arrays;
ostream &operator<<( ostream &output, const Array & a) {
    int i;
    for ( i = 0; i < a.size; i++) {
        output << setw(12) << a.ptr[i];
        if ((i+1)%4 == 0) // 4 numbers per row for output
            output << endl;
    }
    if ( i % 4 != 0) // end last line of output
        output << endl;
    return output; // enables cin >> x >> y >> z;
}
```



# Fig19\_o8.cpp

```
// Fig. 18.9: fig18_09.cpp
// Array class test program.
#include <iostream>
#include "Array.h"
using namespace std;
int main()
{
    Array array1(7);    // 7-element Array
    Array array2;        // 10-element Array
    cout << "Size of Array array1 is "
         << array1.getSize()
         << "\n Array after initialization: \n" << array1;

    cout << "\nEnter 17 integers:" << endl;
    cin >> array1 >> array2;

    cout << "array1: \n" << array1
         << "array2: \n" << array2;
```

## Fig19\_o8.cpp (cont.)

```
// use overloaded inequality (!=) operator
cout << "\nEvaluating: array1 != array2" << endl;
if ( array1 != array2)
    cout << "array1 and array2 are not equal\n";

Array array3(array1); // invoke copy constructor

// use overloaded assignment (=) operator
cout << "\nAssigning array2 to array1:" << endl;
array1 = array2;
cout << "array1: \n" << array1
    << "array2: \n" << array2;
// use overloaded equality (==) operator
cout << "\nEvaluating: array1 == array2" << endl;
if ( array1 == array2)
    cout << "array1 and array2 are equal\n";
```

## Fig19\_o8.cpp (cont.)

```
// use overloaded subscript operator to create rvalue
// array1[5] => array.operator[](5)
cout << "\narray1[5] is " << array1[5];

// use overloaded subscript operator to create lvalue
cout << "\nAssigning 1000 to array1[5]" << endl;
array1[5] = 1000;
cout << "array1: \n" << array1;

// attempt to use out-of-range subscript
cout << "\nAttempt to assign 1000 to array1[15]" <<
endl;
array1[15] = 1000; // ERROR: output of range

return 0;
}
```

# Overload Array Operator [ ]

- Can overload [ ] for your class
  - used with objects of your class
  - typically,  $x[i] \Leftrightarrow *(x+i)$
  - a binary operator: the left operand is a reference object + the right one is an integer
- Format
  - operator must return a reference
  - operator [ ] must be a member function

```
{<cname>& <cname>::operator[](int i)  
{ //functional body; }
```

# Example of Overloading [ ]

- New class `CData` with a short integer array

```
class CData {  
    int len; short *sary;  
public:  
    CData(unsigned long n = 0) {  
        unsigned long temp = n; len = 1;  
        while (temp > 10)  
            { temp = temp/10; len++; }  
        sary = new short[len];  
        for (int i=0; i<len; ++i)  
            { sary[i] = n%10; n /=10; }  
    }  
};
```

## Example of Overloading [ ] (cont.)

```
class CData {  
    ~CData() {  
        if (sary) { delete [] sary;  
                    sary = NULL; }  
    }  
  
    void display() {  
        for (int i=len-1; i>=0; i--)  
            { cout << *(sary+i); }  
        cout << endl;  
    }  
  
    short& operator[](int i) {  
        if (i>=len) { cerr << "Error"; }  
        return *(sary+i);  
    }  
};
```

# Example of Overloading [ ] (cont.)

```
//in main()  
CData x(9316);  
x.display();  
cout << "x[0] = " << x[0]  
      << ", x[3] = " << x[3] << endl;  
x[0] = 2;  
x[3] = 7;  
x.display();
```

```
9316  
x[0] = 6, x[3] = 9  
7312
```

# Converting between Types

- C++ provides explicit type conversion
  - `<datatype>(<data>)`, ex: `int(82.7)`
  - `(<datatype>)<data>`, ex: `(double)49`
- Conversion constructor casts the data of one type into an object of another class, ex

```
class CComplex {  
    CComplex(double r) {  
        real = r; imag = 0;    }  
};
```

```
CComplex o1(4.2);  
CComplex o2 = o1 + CComplex(2.5);
```



# Type Casting for Class

- What is converting a CComplex into a double?
  - need a type conversion function

- Format: 

```
<cname>::operator<datatype> ()  
    { //functional body; }
```

- Example

```
class CComplex {  
    operator double() { return real; }  
};
```

- cannot assign the return datatype
- cannot have any parameter

```
CComplex o1(4.2); double d2 = 12;  
double d3 = d2 + o1;
```

# Overloading ++ and --

- ++/-- are unary operators
  - **prefix** operation: ++obj, --obj
  - **postfix** operation: obj++, obj--
- Declaration of member functions

```
<CNAME> &<CNAME>::operator++(); //prefix  
<CNAME> <CNAME>::operator++(int); //postfix
```

- Declaration of friend functions

```
//prefix friend function  
friend <CNAME> & operator++(<CNAME>&);  
//postfix friend function  
friend <CNAME> operator++(<CNAME>&, int);
```

# Example of Overloading ++

```
class CCount {  
    unsigned int unCnt;  
public:  
    CCount(int n = 0) { unCnt = 0; }  
    void display() { cout << unCnt; }  
    //prefix increment with member function  
    CCount& operator++();  
    //postfix increment with global function  
    friend CCount operator++(CCount&, int);  
};  
CCount& CCount::operator++() {  
    unCnt++; return *this; }  
CCount operator++(CCount& x, int y) {  
    CCount tmp = x; x.unCnt++; return tmp; }
```

# Example of Overloading ++ (cont.)

```
//in main()  
    CCount d1(12), d2;  
    d2=d1++; //call postfix increment  
    d1.display();d2.display();cout << endl;  
    d2=++d1; //call prefix increment  
    d1.display();d2.display();cout << endl;  
    ++++d1;  
    d1.display();d2.display();cout << endl;
```

```
13 12  
14 14  
16 14
```

# Summary

- C++ built-in operators can be overloaded
  - to work with objects of your class
- Operators are really just functions
- Operators can be overloaded as **member functions** where
  - the **first operand is the calling object**
- Overloading operators can be classified into
  - overloading member functions
  - overloading friend functions

# References

- Paul Deitel and Harvey Deitel, “C How to Program” Sixth Edition
  - Chapter 18
- Paul Deitel and Harvey Deitel, “C++ How to Program (late objects version)” Seventh Edition
  - Chapter 11: Operator Overloading
- W. Savitch, “Absolute C++,” Fourth Edition
  - Chapter 8