
Creative Software Design

11 – Copy Constructor, Operator Overloading

Yoonsang Lee

Fall 2021

Final Exam Announcement

- In accordance with the "With Corona" policy, the university has decided to take the final exam in the second semester as a face-to-face exam in principle.
- In accordance with this policy, our final exam will be an offline exam.
- If you are unable to take the offline final exam (stay abroad, corona confirmed, etc.), please contact the TA in advance.
 - Jeongmin Lee (이정민 조교), j0064423@hanyang.ac.kr
- Time & Place: To be announced
- Scope: Lecture 8 ~ 13

Today's Topics

- Copy constructor
- friend, static
- Operator overloading

Copy constructor

- A copy constructor is a constructor that initializes an object using another object of the same class.
- The general form is:

```
ClassName (const ClassName& src_obj);
```

When is a copy constructor called?

- When an object is returned by value.
- When an object is passed by value (not by address value) as a function argument.
- When an object is constructed based on another object of the same class.

When is a copy constructor called?

```
class Point
{
public:
    double x, y;
    //...
};

Point getScaledPoint(double scale, Point p)
{
    Point p_new;
    p_new.x = p.x*scale; p_new.y = p.y*scale;
    return p_new;
}

int main(int argc, char* argv[])
{
    Point p1(0.1, 0.2);
    Point p2 = getScaledPoint(2.0, p1);

    Point p3 = p1;
    Point p4(p1);
    return 0;
}
```

- When an object is returned by value.
- When an object is passed by value (not by address value) as a function argument.
- When an object is constructed based on another object of the same class.

Default copy constructor

- A **default copy constructor** is implicitly created by compiler if there is no user-defined copy constructor.
- It does a **member-wise copy** between objects,
 - where each member is copied by its own copy constructor.
 - This works fine in general, but does not work for some cases. We should define our own copy constructor for these cases.

Default copy constructor: Example 1

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

class Point{
private:
    int x, y;
public:
    Point(int a=0): x(a), y(a) {}
    ~Point(){ cout << "bye " << x << " " << y << endl;}
    void Print(){ cout << x << " " << y << endl;}
};

int main()
{
    Point P1(3);
    Point P2 = P1; // by default copy constructor
    Point P3(P2); // by default copy constructor

    P1.Print();
    P2.Print();
    P3.Print();

    return 0;
}
```

- Default copy constructor copies each member of the object

Default copy constructor: Example 2-1

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

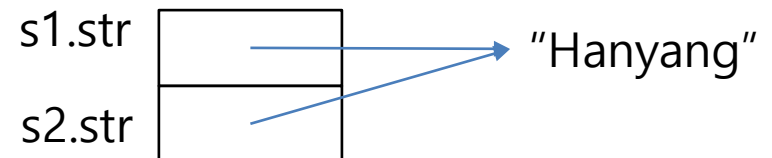
class MyString{
private:
    int len;
    char *str;
public:
    MyString(const char *s = ""){
        len = strlen(s);
        str = new char[len+1];
        strcpy(str, s);
    }
    ~MyString(){delete[] str;}
    void Print() { cout << str << endl;}
};

int main(){

    MyString s1 = "Hanyang";
    MyString s2 = s1; //copy constructor

    s1.Print();
    s2.Print();

    return 0;
}
```



Default copy constructor: Example 2-2

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

class MyString{
private:
    int len;
    char *str;
public:
    MyString(const char *s = ""){
        len = strlen(s);
        str = new char[len+1];
        strcpy(str, s);
    }
    ~MyString(){delete[] str;}
    void Print() { cout << str << endl;}
};

MyString GetString(void){
    MyString str("HY");
    return str;
}

int main(){

    MyString s2 = GetString();
    s2.Print();

    return 0;
}
```

//the space for "HY" is deallocated

//the address to "HY" is copied

User-defined copy constructor: Example

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

class MyString{
private:
    int len;
    char *str;
public:
    MyString(const char *s = ""){
        len = strlen(s);
        str = new char[len+1];
        strcpy(str, s);
    }
    MyString(const MyString &s){ //redefine copy constructor
        len = s.len;
        str = new char[len+1];
        strcpy(str, s.str);
    }
    ~MyString(){delete[] str;}
    void Print() { cout << str << endl;}
};

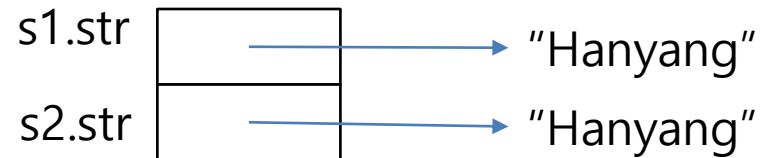
int main(){

    MyString s1 = "Hanyang";
    MyString s2 = s1; //copy constructor

    s1.Print();
    s2.Print();

    return 0;
}
```

- The problem of deallocation by delete operator was resolved



Default copy constructor & Default constructor

- Recall: A **default constructor** is implicitly created by compiler if there is no user-defined constructor.
- If you define a copy constructor, the compiler doesn't create the default constructor and default copy constructor.

Copy constructor: Example

```
class Point
{
public:
    double x, y;
    Point(double x_, double y_):x(x_), y(y_) {}

    // The most general form.
    Point(const Point& p) { x = p.x; y = p.y; }

    // This form compiles as well.
    // In general, however, copy constructors are not expected
to modify the object passed in, so this form is not recommended
    Point(Point& p) { x = p.x; y = p.y; }

    // Compile error. If it were compiled, it would result in
infinite calling of copy constructor.
    Point(Point p) { x = p.x; y = p.y; }
};
```

Quiz #1

- Go to <https://www.slido.com/>
- Join #csd-ys
- Click "Polls"

- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. **2017123456: 4)**

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

Friend Class and Function

- Functions or classes can be "friends" of another class (let's say ClassA).
 - If you declare them as "friends" in the definition of ClassA,
 - Then these "friends" can **access all members of ClassA including private members.**

```
class ClassA {
    private:
        int var_;
        friend class ClassB;
        friend void DoSomething(const ClassA& a);
};

class ClassB {
    // ...
    void Function(const ClassA& a) { cout << a.var_; } // OK.
};

void DoSomething(const ClassA& a) { cout << a.var_; } // OK.
```

Friend Class and Function

- "friend" should be used with caution.
 - Too many "friend" functions or classes may lessen the value of *encapsulation / data hiding*.
 - If "friends" are used properly, they can enhance *encapsulation / data hiding*.
 - For example, a LinkedList class may be allowed to access private members of Node class, but all other classes are not.
- Note that access specifiers have no effect on the meaning of friend declarations.
 - "friend" can appear in private, protected, or public sections, with no difference.
 - <https://en.cppreference.com/w/cpp/language/friend>

Static Members

- Static member variables in a class are **shared by all the objects of the class.**
 - Static member variables **should be defined outside the class** to allocate memory for them.
- Static member functions **can only access static members.**
 - Static member functions cannot be virtual.
- Static members can be accessed by class name or object name.

Static Members

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

class Point{
private:
    int x, y;
    static int count;
public:
    Point(int a=0, int b=0): x(a), y(b) {count++;}
    ~Point(){ cout << x << " " << y << endl;}
    static int GetCount() {return count;}
};
int Point::count = 0;

int main()
{
    cout << Point::GetCount() << endl;
    Point P1(1,2);
    cout << Point::GetCount() << endl;
    Point P2 = Point(3,4);
    cout << P2.GetCount() << endl;
    return 0;
}
```

Recall: Function Overloading

- Use multiple functions sharing the same name
 - A family of functions that do the same thing but using different argument lists

```
void print(const char * str, int width); // #1
void print(double d, int width);       // #2
void print(long l, int width);         // #3
void print(int i, int width);          // #4
void print(const char *str);           // #5
```

```
print("Pancakes", 15); // use #1
print("Syrup");        // use #5
print(1999.0, 10);     // use #2
print(1999, 12);       // use #4
print(1999L, 15);     // use #3
```

Operator Overloading

- An operator function is a special function form to overload an operator
- `operatorop` (arguments)
 - `op` is a valid C++ operator
 - `operator+()` overloads the `+` operator
- Note that C++ even allows redefining built-in operators such as `+`, `-`, `*`, ...
- There are two ways of operator overloading:
 - a) overloaded as a **class member function**
 - b) overloaded as a **non-member function**.

Operator overloading as member function

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

class Box {
private:
    int x, y, z;
public:
    Box(int a=0, int b=0, int c=0): x(a), y(b), z(c){}
    Box Sum(const Box box) {
        return Box(x+box.x, y+box.y, z+box.z);
    }
    void Print(){ cout << x << " " << y << " " << z << endl;
    }
};

int main(){
    Box B1(1,1,1);
    Box B2(2,2,2);
    Box B3 = B1.Sum(B2);
    B3.Print();

    return 0;
}
```

Operator overloading as member function

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

class Box {
private:
    int x, y, z;
public:
    Box(int a=0, int b=0, int c=0): x(a), y(b), z(c){}
    Box operator+(const Box box) {
        return Box(x+box.x, y+box.y, z+box.z);
    }
    void Print(){ cout << x << " " << y << " " << z << endl;
    }
};

int main(){
    Box B1(1,1,1);
    Box B2(2,2,2);
    Box B3 = B1.operator+(B2);
    B3.Print();
    Box B4 = B1 + B2;
    B4.Print();

    return 0;
}
```

P1 + P2
→ **P1.operator+(P2)**

Operator overloading as member function

- $P1 + P2$
- \rightarrow **$P1.operator+(P2)$**
- That means, the overloaded operator member function gets invoked on the **first operand**.

- What if the **first operand is not a class type**, like double?
 - For example, $2.0 + P2$?
- \rightarrow You should overload a **non-member** operator function!

Operator overloading as nonmember function

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

class Point{
    int x, y;
public:
    Point(int a, int b): x(a), y(b){}
    void Print(){ cout << "(" << x << "," << y << ")" << endl;}
    friend Point operator+(int a, Point &Po);
};

Point operator+(int a, Point &Po){
    return Point(a + Po.x, a + Po.y);
}

int main(){

    Point P1(2, 2);
    int a = 2;

    Point P3 = a + P1; // Point P3 = operator+(a, P1);
    P3.Print();

    return 0;
}
```

P1 + P2
→ **operator+(P1, P2)**

Operator overloading as nonmember function

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

class Box {
private:
    int x, y, z;
public:
    Box(int a=0, int b=0, int c=0): x(a), y(b), z(c){}

    friend Box operator+(const Box& box1, const Box& box2);
    void Print(){ cout << x << " " << y << " " << z << endl;}
};

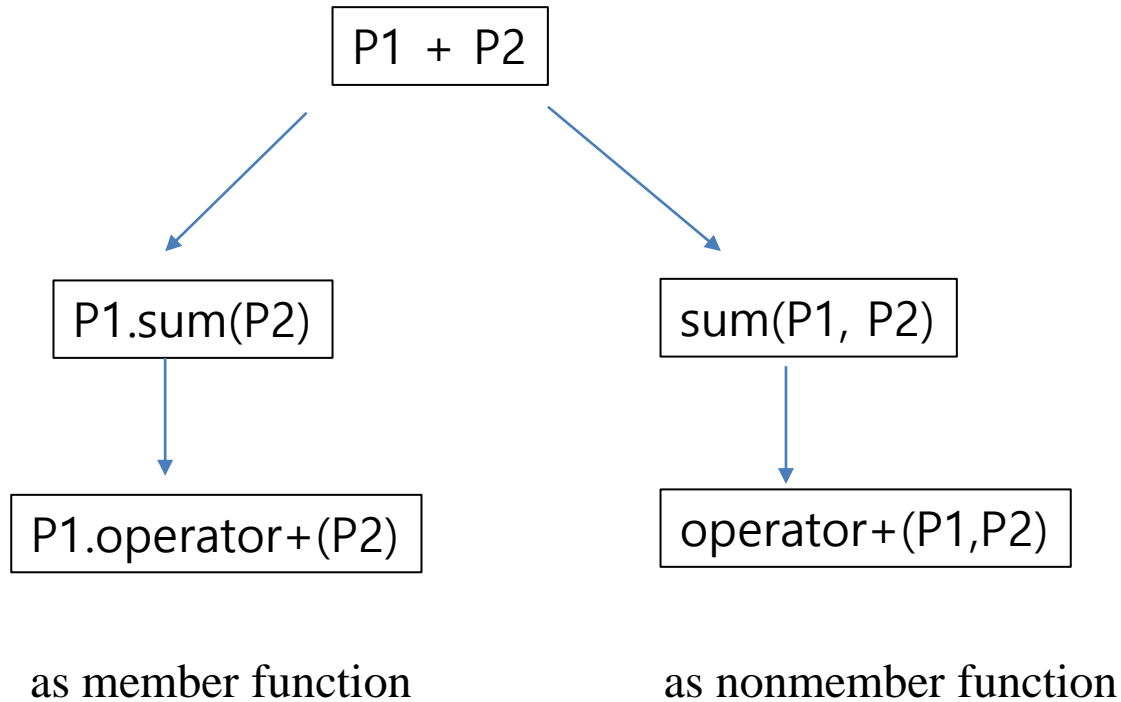
Box operator+(const Box& box1, const Box& box2) {
    return Box(box1.x+box2.x, box1.y+box2.y, box1.z+box2.z);
}

int main(){
    Box B1(1,1,1);
    Box B2(2,2,2);

    Box B4 = operator+(B1,B2);    // Box B4 = B1 + B2;
    B4.Print();

    return 0;
}
```

Operator function



Quiz #2

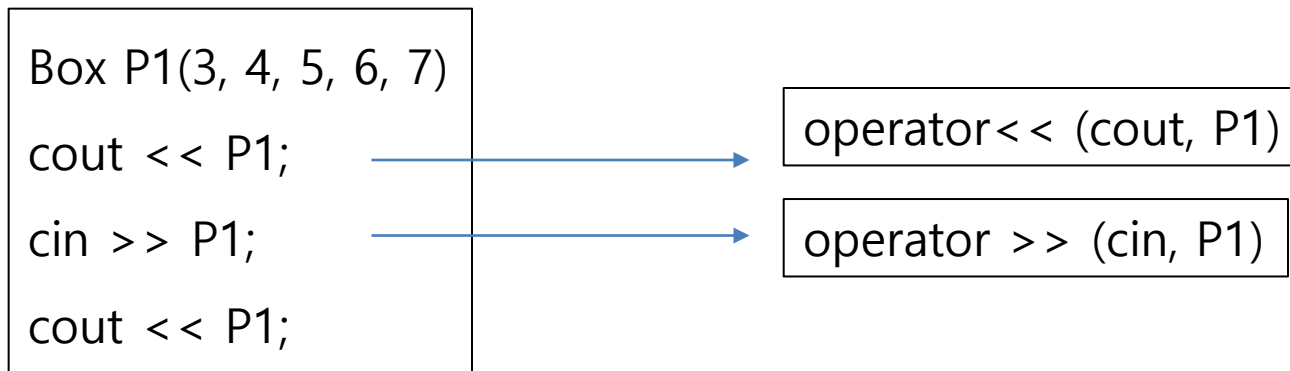
- Go to <https://www.slido.com/>
- Join #csd-ys
- Click "Polls"

- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. 2017123456: 4)

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

Operator Overloading: <<, >> operator

- You should overload stream operators as non-member functions
 - because you cannot modify the first operand's class.



Operator Overloading: <<, >> operator

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

class Point{
private:
    int x, y;
public:
    Point(int a, int b): x(a), y(b){}
    void Print(){ cout << x << " " << y << endl;}
    friend ostream& operator<< (ostream& os, const Point& pt);
    friend istream& operator>> (istream& is, Point& pt);
};

ostream& operator<<(ostream& os, const Point& pt)
{
    os << pt.x << " " << pt.y << endl;
    return os;
}

istream& operator>>(istream& is, Point& pt)
{
    is >> pt.x >> pt.y;
    return is;
}

int main(){
    Point P1(2,2);
    P1.Print();
    cout << P1;
    cin >> P1;
    cout << P1;

    return 0;
}
```

Assignment Operator(= operator) Overloading

- A **default assignment operator** is implicitly created by compiler if there is no user-defined assignment operator.
- It does a **member-wise copy** between objects.
 - where each member is copied by its own assignment operator.
 - Like default copy constructor, this works fine in general, but does not work for some cases.

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

Copy Constructor vs. Assignment Operator

```
class Point
{
private:
    double x, y;
public:
    Point(double x_, double y_):x(x_), y(y_) {}

    Point(const Point& p)
    { x = p.x; y = p.y; cout << "copy constructor" << endl; }

    Point& operator=(const Point& p)
    { x = p.x; y = p.y; cout << "assignment operator" << endl; return *this;
    }
};

int main()
{
    Point p1(1,2);

    Point p2(p1);    // "copy constructor"
    Point p3 = p1; // "copy constructor"

    Point p4(2,3);
    p4 = p1;      // "assignment operator"

    return 0;
}
```

Return Type of Assignment Operator

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

class Point
{
private:
    double x, y;
public:
    Point():x(0.0), y(0.0) {}
    Point(double x_, double y_):x(x_), y(y_) {}

    // inconsistent behavior with default assignment operator & assignments for primitive types
    Point operator=(const Point& p)
    { x = p.x; y = p.y; return Point(*this); }

    // same behavior as default assignment operator & assignments for primitive types-> use this!
    Point& operator=(const Point& p)
    { x = p.x; y = p.y; return *this; }

    friend ostream& operator<<(ostream& os, const Point& p);
};

ostream& operator<<(ostream& os, const Point& p)
{
    os << "(" << p.x << ", " << p.y << ")";
    return os;
}

int main()
{
    Point p1(1,2);
    Point p2, p3;
    (p3 = p2) = p1;

    cout << p1 << p2 << p3 << endl;
    return 0;
}
```


Default assignment operator: Example

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

class MyString{
private:
    int len;
    char *str;
public:
    MyString(const char *s = ""){
        len = strlen(s);
        str = new char[len+1];
        strcpy(str, s);
    }
    ~MyString(){delete[] str;}
    void Print() { cout << str << endl;}
};

int main(){

    MyString s1("Hanyang");
    MyString s2("University");

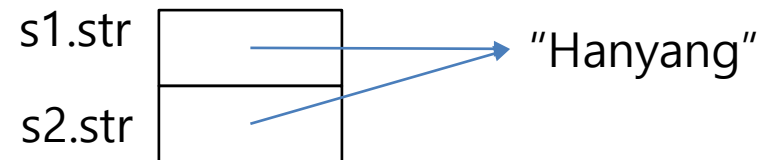
    s2 = s1;

    s1.Print();
    s2.Print();

    return 0;
}
```

Is it OK?

= operator copies the address



User-defined assignment operator: Example

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

class MyString{
private:
    int len;
    char *str;
public:
    MyString(const char *s = ""){
        len = strlen(s);
        str = new char[len+1];
        strcpy(str, s);
    }
    MyString &operator=(const MyString &string){
        delete[] str;
        len = string.len;
        str = new char[len+1];
        strcpy(str, string.str);
        return(*this);
    }
    ~MyString(){delete[] str;}
    void Print() { cout << str << endl;}
};

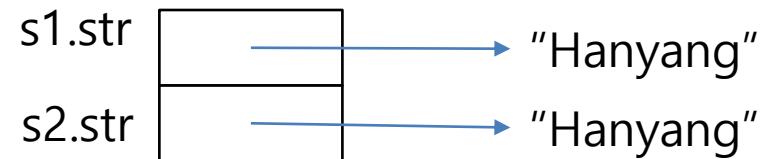
int main(){

    MyString s1("Hanyang");
    MyString s2("University");

    s2 = s1;

    s1.Print();
    s2.Print();

    return 0;
}
```



Operator Overloading: unary operator

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

class Point{
private:
    int x, y;
public:
    Point(int a, int b): x(a), y(b){}
    Point operator-() { return Point(-x, -y); }
    Point& operator-() { x=-x; y=-y; return *this; }
    void Print(){ cout << x << " " << y << endl;}
};

int main(){
    Point P1(2,2);
    P1.Print();
    Point P2 = -P1;
    P1.Print();
    P2.Print();

    return 0;
}
```

1)

2)

1) is consistent with primitive types.

1)

2	2
2	2
-2	-2

2)

2	2
-2	-2
-2	-2

Operator Overloading: increment operator

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

class Point{
private:
    int x, y;
public:
    Point(int a, int b): x(a), y(b){}
    Point &operator++(){x++; y++; return *this;}
    void Print(){ cout << x << " " << y << endl;}
};

int main(){
    Point P1(2,2);
    P1.Print();
    Point P2 = ++P1;
    P1.Print();
    (++P1).Print();

    return 0;
}
```

(++P1) → P1.operator++()

Operator Overloading: increment operator

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

class Point{
private:
    int x, y;
public:
    Point(int a, int b): x(a), y(b){}
    //Point &operator++(int a){Point temp = (*this); x++; y++; return temp;}
    Point operator++(int a){Point temp = (*this); x++; y++; return temp;}
    void Print(){ cout << x << " " << y << endl;}
};

int main(){
    Point P1(2,2);
    P1.Print();
    Point P2 = P1++;
    P1.Print();
    P2.Print();

    return 0;
}
```

(++P1) → P1.operator++()

(P1++) → P1.operator++(0)

Operator Overloading: []

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

class Point{
private:
    int x,y,z;
public:
    Point(int a = 0, int b = 0, int c = 0): x(a), y(b), z(c){}
    int& operator[](int index){
        if (index == 0) return x;
        else if (index == 1) return y;
        else if (index == 2) return z;
    }
    void Print(){cout << x << " " << y << " " << z << endl;}
};

int main(){
    Point P1(1,1,1);
    P1[0] = 2;
    P1[1] = 3;
    P1[2] = 4;
    P1.Print();
    return 0;
}
```

Quiz #3

- Go to <https://www.slido.com/>
- Join #csd-ys
- Click "Polls"

- Submit your answer in the following format:
 - **Student ID: Your answer**
 - e.g. 2017123456: 4)

- Note that you must submit all quiz answers **in this format** to be counted as attendance.

Operator Overloading: Summary

```
class A {                                     // A a0, a1;
    A& operator =(const A& a);               // a0 = a1;
    A operator +(const A& a) const;         // a0 + a1
    A operator +() const;                   // +a0
    A& operator +=(const A& a);             // a0 += a1;
    A& operator ++();                       // ++a0
    A operator ++(int);                     // a0++
};

A operator +(const A& a0, const A& a1);    // a0 + a1
A operator +(const A& a0);                 // +a0
A& operator +=(A& a0, const A& a1);        // a0 += a1;
A& operator ++(A& a0);                     // ++a0
A operator ++(A& a0, int);                 // a0++

std::ostream& operator <<(std::ostream& out, const A& a); // cout << a0;
```


Operator Overloading: Summary

- In general, an operator whose result is ...
- New value: Returns the new value by value
 - e.g. +, -, ...
- Existing value, but modified: Returns a reference to the modified value.
 - e.g. =, +=, ...

Operator Overloading: Summary

- The C++ language rarely puts constraints on operator overloading such as
 - what the overloaded operators do
 - what should be the return type
- But in general, overloaded operators *are expected to behave as similar as possible to the **built-in operators***:
 - operator+ is expected to add, rather than multiply its arguments,
 - operator= is expected to assign
- The return types are limited by the expressions in which the operator is expected to be used:
 - for example, assignment operators return by reference to make it possible to write `a = b = c = d`, because the built-in operators allow that.

Operator Overloading: Summary

- Most commonly overloaded operators are
 - Arithmetic operators : +, -, *, / ...
 - Assignment operators : =, +=, -=, *= ...
 - Comparison operators : <, >, <=, >=, ==, != ...
 - For array or containers : [], () ...
 - Rarely : ->, new, delete, ...
- Operator overloading must be used very carefully, since it can hamper the readability seriously.

Operator that can be overloaded

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

Example 1

```
class Time
{
private:
    int hours;
    int minutes;
public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time operator+(const Time & t) const;
    void Show() const;
};
```

```
void Time::AddMin(int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}
```

```
void Time::AddHr(int h)
{
    hours += h;
}
```

```
void Time::Reset(int h, int m)
{
    hours = h;
    minutes = m;
}
```

```
Time Time::operator+(const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}
```

Converting Constructor & Operator Overloading

- Basically, constructors can convert some type (the parameter type) to another type (the class belonging the constructor).
- This can affect the behavior of overloaded operators.
- See the following Example 2.

Example 2

```
class Complex {
public:
    Complex() : real(0.0), imag(0.0) {}
    Complex(double r, double i) : real(r), imag(i) {}
    Complex(const Complex& c) : real(c.real), imag(c.imag) {}

    Complex operator+(const Complex& c) const {
        return Complex(real + c.real, imag + c.imag);
    }
private:
    double real, imag;
};

void Test() {
    Complex a(1.0, 2.0), b(2.0, 5.0);
    Complex c(a + b);
    c = c + a;
}
```

Example 2

```
class Complex {
public:
    Complex() : real(0.0), imag(0.0) {}
    Complex(double r, double i) : real(r), imag(i) {}
    Complex(const Complex& c) : real(c.real), imag(c.imag) {}

    Complex operator+(const Complex& c) const;
private:
    double real, imag;
};

void Test() {
    Complex a(1.0, 2.0), b(2.0, 5.0), c;
    c = a + b;      // OK.
    c = a + 3.0;   // Error.
    c = 2.0 + b;   // Error.
}
```


Example 2

```
class Complex {
public:
    Complex() : real(0.0), imag(0.0) {}
    Complex(double v) : real(v), imag(0.0) {} //Constructor for a
single v.
    Complex(double r, double i) : real(r), imag(i) {}
    Complex(const Complex& c) : real(c.real), imag(c.imag) {}

    Complex operator+(const Complex& c) const;
private:
    double real, imag;
};

void Test() {
    Complex a(1.0, 2.0), b(2.0, 5.0), c;
    c = a + b; // OK.
    c = a + 3.0; // OK.
    c = 2.0 + b; // Error.
}
```

Example 2

```
class Complex {
public:
    Complex() : real(0.0), imag(0.0) {}
    Complex(double v) : real(v), imag(0.0) {} // Constructor for a single v.
    Complex(double r, double i) : real(r), imag(i) {}
    Complex(const Complex& c) : real(c.real), imag(c.imag) {}

    Complex& operator=(const Complex& c);
private:
    double real, imag;
    friend Complex operator+(const Complex& lhs, const Complex& rhs);
};
Complex operator+(const Complex& lhs, const Complex& rhs) {
    return Complex(lhs.real + rhs.real, lhs.imag + rhs.imag);
}
void Test() {
    Complex a(1.0, 2.0), b(2.0, 5.0), c;
    c = a + b;    // OK.
    c = a + 3.0; // OK.
    c = 2.0 + b; // OK.}
```

```

class Complex {
public:
    Complex() : real(0.0), imag(0.0) {}
    Complex(double v) : real(v), imag(0.0) {}
    Complex(double r, double i) : real(r), imag(i) {}
    Complex(const Complex& c) : real(c.real), imag(c.imag) {}

    Complex& operator=(const Complex& c) {                // Complex a(1.0, 0.0), c;
        real = c.real, imag = c.imag;                    // c = a;
        return *this;
    }

    Complex operator+() const { return *this; }           // c = +a;
    Complex operator-() const { return Complex(-real, -imag); } // c = -a;

    double& operator[](int i) { return i == 0 ? real : imag; } // i = c[0];
    const double& operator[](int i) const { return i == 0 ? real : imag; }

private:
    double real, imag;

    friend Complex operator+(const Complex& lhs, const Complex& rhs);
    friend bool operator<(const Complex& lhs, const Complex& rhs);
};

Complex operator+(const Complex& lhs, const Complex& rhs) const { // c + a
    return Complex(lhs.real + rhs.real, lhs.imag + rhs.imag);
}

bool operator<(const Complex& lhs, const Complex& rhs) { // if (c < a)
    return lhs.real < rhs.real && lhs.imag < rhs.imag;
}

```

Next Time

- Labs in this week:
 - Lab1: Assignment 11-1
 - Lab2: Assignment 11-2

- Next lecture:
 - 12 - Template