Topic 9
Operator Overloading

Operator

- Concise notation for common operations
  - x + y * z is more intuitive than
  - Multiply y by z and add the result to x
- C++ supports a set of operators for your own classes. For example

```
Rectangle r1(10, 20), r2(30, 40) ;
Rectangle r3 ;
r3 = r1 + r2 ;
```

- Actually, two objects cin and cout support >> and << operators

```
cin >> value ;
cout << value ;
cout << r1 << r1 << r3 ;
```

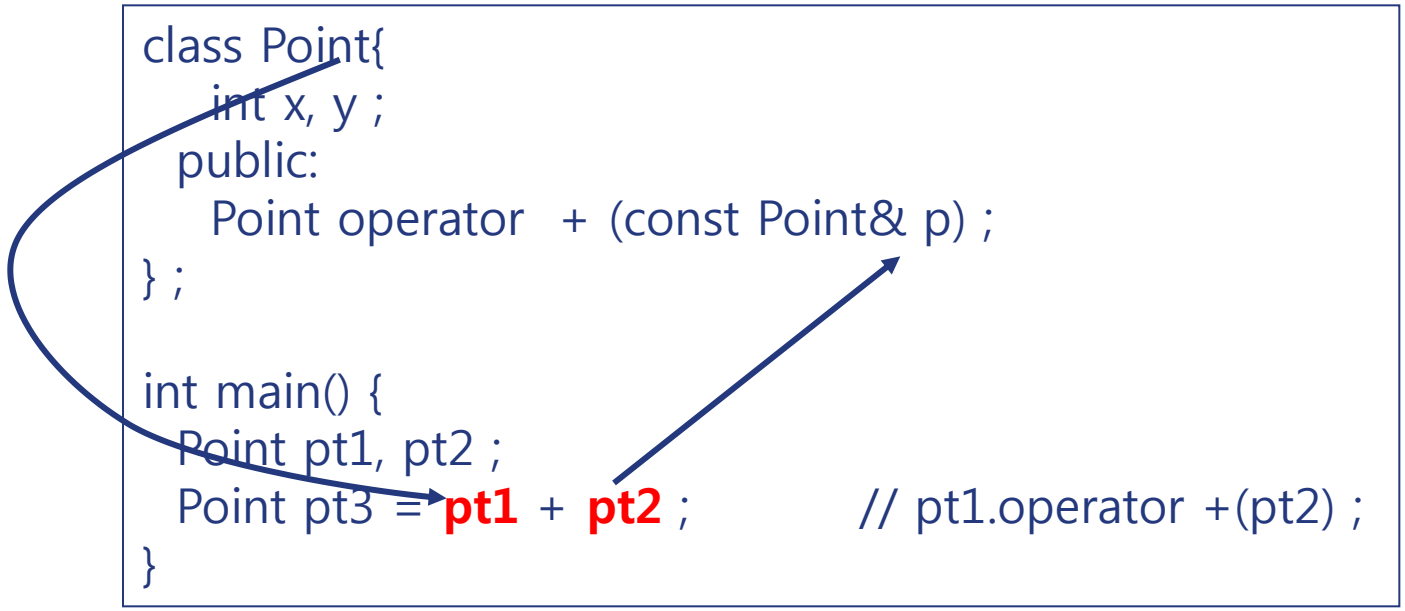
Motivation

- Operator overloading provides a more conventional and convenient notation.
- Want to add an "add" function to Complex class
  - Without operator overloading

```
class Complex {
private:
    double re, im ;
public:
   void add(const Complex& c)  {
      re += c.re ;
      im += c.im ;
   }
} ;
```

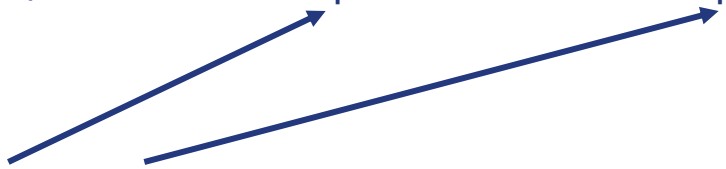
```
void f() {
  Complex c1, c2 ;
  c1.add( c2 ) ;
}
```

Member Operators



Non-Member Operators

```
class Point{
    int x, y ;
  public:
    ...
} ;
Point operator  + (const Point& p1, const Point& p2) {...}

int main() {
  Point pt1, pt2 ;
  Point pt3 = pt1 + pt2 ;        // operator +(pt1, pt2) ;
}
```



Operator Overloading Example

- A programmer can define a meaning for operators when applied to objects of a specific class

```
class Complex {
private:
   double re, im ;
public:
   void operator += (const Complex& c)  {
      re += c.re ;
      im += c.im ;
   }
} ;
```

```
void f() {
  Complex c1, c2 ;
  c1 += c2 ;
  // c1.operator+=(c2) ;
}
```

Overloadable Operators

- The following operators can be overloaded.

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

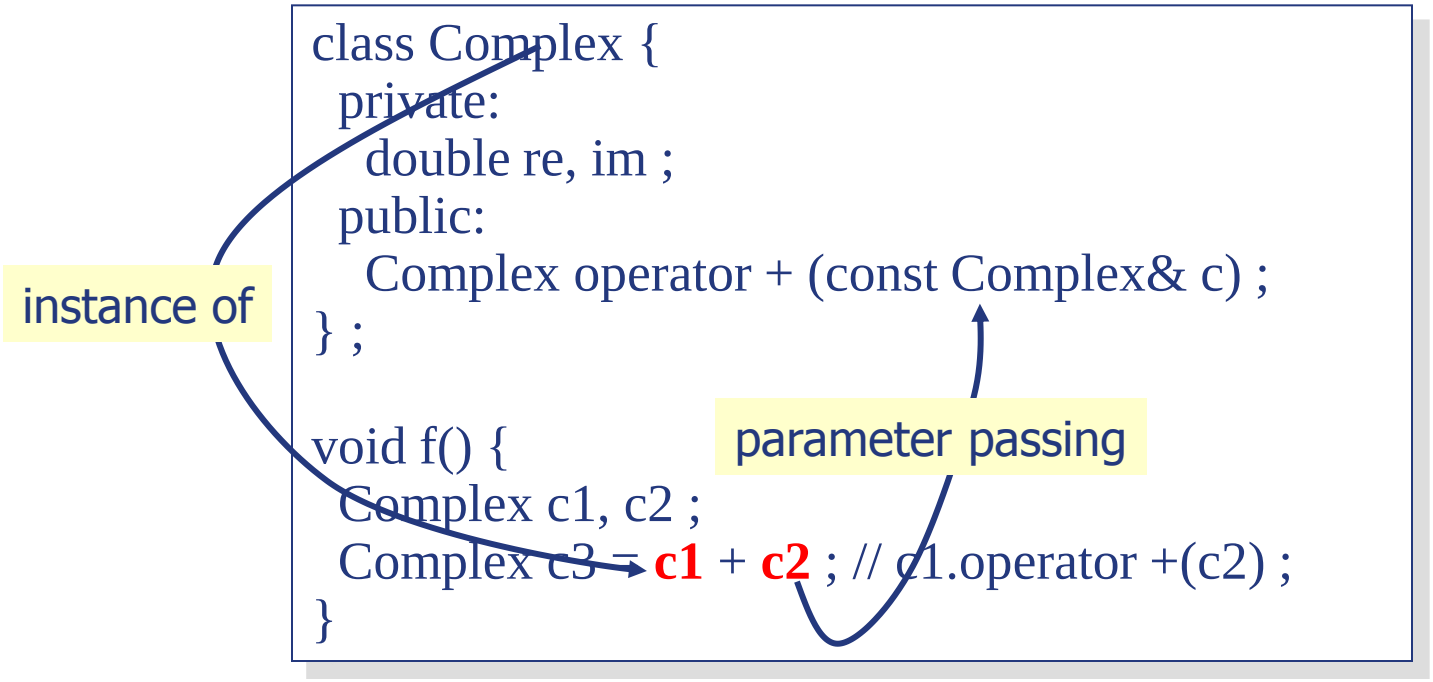
- It is not possible
  - to overload ::, ., .*, ?:, sizeof, or typeid
  - to change the precedence of the operators
  - to change the expression syntax, e.g.) unary vs. binary
  - to define a new operator

Operator Overloading

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

class Complex {
   double re, im ;
public:
   Complex(double re=0, double im=0) { this->re = re ; this->im = im ; }
   Complex operator+ (const Complex& c) { return Complex(re+c.re, im+c.im) ; }
   Complex operator- (const Complex& c) { return Complex(re-c.re, im-c.im) ; }
   friend ostream& operator << (ostream& os, const Complex& c) ;
} ;
ostream& operator << (ostream& os, const Complex& c) {
   os << c.re << " + " << c.im << "i" ;
   return os ;
}

int main() {
   Complex c1(1, 0), c2(2, 1) ;
   Complex c3 (c1 + c2 ) ;
   Complex c4 = c1 - c2 ;

   cout << c3 << c4  << endl ;
}
```

Member Operators

- An operator can be overloaded as a member function



Member Operators

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

class Complex {
    double re, im ;
public:
    Complex(double re=0, double im=0) { this->re = re ; this->im = im ; }

    Complex operator+ (const Complex& c) { return Complex(re+c.re, im+c.im) ; }
    bool operator == (const Complex& c) { return re == c.re && im == c.im ; }
    bool operator != (const Complex& c) { return ! operator == (c) ; }
} ;

int main() {
    Complex c1(1, 0), c2(2, 1) ;
    Complex c3 = c1 + c2 ;
    if ( c3 == Complex(3, 1) )
       cout << "They are same" << endl ;
    if ( c3 != Complex(3, 1) )
       cout << "They are not same" << endl ;
}
```

Non-Member Operators

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

class Complex {
    double re, im ;
public:
    Complex(double re=0, double im=0) { this->re
    Complex operator+ (const Complex& c) { return Complex(re+c.re, im+c.im) ; }

    friend bool operator == (const Complex& c1, const Complex& c2) ;
    friend bool operator != (const Complex& c1, const Complex& c2) ;
} ;

bool operator == (const Complex& c1, const Complex& c2) {
    return c1.re == c2.re && c1.im == c2.im ;
}
bool operator != (const Complex& c1, const Complex& c2) {
    return ! operator == (c1, c2) ;
}
```

```
int main() {
    Complex c1(1, 0), c2(2, 1) ;
    Complex c3 = c1 + c2 ;
    if ( c3 == Complex(3, 1) )
        cout << "They are same" << endl ;
    if ( c3 != Complex(3, 1) )
        cout << "They are not same" << endl ;
}
```

Non-Member Operators: Usage

- Used when the left operand cannot be accessed.

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

class Complex {
   double re, im ;
public:
   Complex(double re=0, double im=0) { this->re = re ; this->im = im ; }
   Complex operator+ (const Complex& c) { return Complex(re+c.re, im+c.im) ; }
   Complex operator- (const Complex& c) { return Complex(re-c.re, im-c.im) ; }
   friend ostream& operator << (ostream& os, const Complex& c) ;
} ;
ostream& operator << (ostream& os, const Complex& c) {
   os << c.re << " + " << c.im << "i" ;
   return os ;
}
int main() {
   Complex c1(1, 0), c2(2, 1) ;
   Complex c3 = c1 + c2 ;
   Complex c4 = c1 - c2 ;
   cout << c3 << endl << c4 ;
}
```

Non-Member Operators: Usage

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

enum Grade { FRESH=1, SOPHOMORE, JUNIOR, SENIOR } ;
Grade& operator ++ (Grade& grade) {
    return grade = ( grade != SENIOR ) ? Grade(grade+1) : SENIOR ;
}
class Student {
    Grade grade ;
    string name ;
public:
    Student(const string& _name, Grade grade=FRESH) : name(_name) { this->grade = grade ; }
    //void upGrade() { if ( grade != SENIOR ) grade = Grade(grade+1) ; }
    void upGrade() { ++ grade ; }

    //bool isEqual(const Student& st) const { return name == st.name && grade == st.grade ; }
    bool operator == (const Student& st) const { return name == st.name && grade == st.grade ; }
    friend ostream& operator << (ostream& os, const Student& st) ;
} ;
ostream& operator << (ostream& os, const Student& st) {
    os << st.name << " "  << st.grade ;
    return os ;
}
```

```
int main() {
    Student st1("Kim"), st2("Kim", SOPHOMORE) ;

    st1.upGrade() ;
    cout << st1 << endl ;
    if ( st1 == st2 ) cout << "They are same" << endl ;
}
```

Mixed-mode Arithmetic

```
int main() {
  Complex c1(1, 0) ;

  Complex c2 = c1 + c2 ;
  Complex c3 = c1 + 10 ;
  Complex c4 = 10 + c1 ;

  Complex c5 = c1 - c2 ;
  Complex c6 = c1 - 10 ;
  Complex c7 = 10 - c1 ;
}
```

```
class Complex {
  double re, im ;
public:
  Complex(double re=0, double im=0) { this->re = re ; this->im = im ; }
  // + operator
  Complex operator+ (const Complex& c) { return Complex(re+c.re, im+
  Complex operator+ (const double re) { return Complex(this->re+re, im
  friend Complex operator + (const double re, const Complex& c) ;
  // - operator
  Complex operator- (const Complex& c) { return Complex(re-c.re, im-c.im) ; }
  Complex operator- (const double re) { return Complex(this->re-re, im) ; }
  friend Complex operator - (const double re, const Complex& c) ;
} ;

Complex operator + (const double re, const Complex& c) { return Complex(re+c.re, c.im) ; }
Complex operator - (const double re, const Complex& c) { return Complex(re-c.re, c.im) ; }
```

Mixed-mode Arithmetic

```
class Complex {
    double re, im ;
public:
    Complex(double re=0, double im=0) { this->re = re ; this->im = im ; }

    friend Complex operator + (const Complex& c1, const Complex& c2) ;
    friend Complex operator - (const Complex& c1, const Complex& c2) ;
} ;
Complex operator + (const Complex& c1, const Complex& c2) {
    return Complex(c1.re+c2.re, c1.im+c2.im) ;
}
Complex operator - (const Complex& c1, const Complex& c2) {
    return Complex(c1.re-c2.re, c1.im-c2.im) ;
}
int main() {
    Complex c1(1, 0) ;

    Complex c2 = c1 + c2 ;       // operator + (Complex, Complex)
    Complex c3 = c1 + 10 ;       // operator + (Complex, Complex(10, 0))
    Complex c4 = 10 + c1 ;       // operator + (Complex(10, 0), Complex)

    Complex c5 = c1 - c2 ;       // operator - (Complex, Complex)
    Complex c6 = c1 - 10 ;       // operator - (Complex, Complex(10, 0))
    Complex c7 = 10 - c1 ;       // operator - (Complex(10, 0), Complex)
}
```

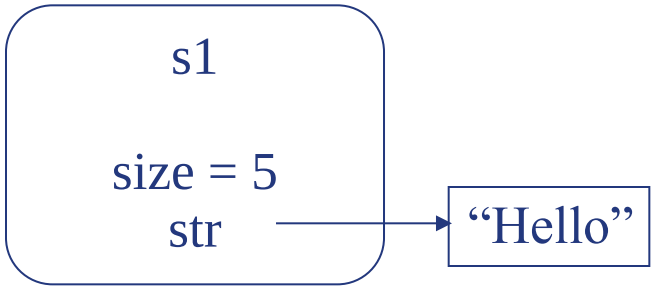
기본 대입 연산자의 문제점

```
#include <cstring>
using namespace std ;
const int STRING_DEFAULT_LENGTH = 100 ;
class MyString {
  char* str ;
  int size ;
public:
  MyString(const char* const s) { // 생성자
    size = strlen(s) ;
    str = new char[size+1] ;
    for ( int i = 0 ; i < size ; i ++ ) str[i] = s[i] ;
    str[size] ='₩0' ;
  }
  MyString(const MyString& another) { // 복사 생성자
    size = another.size ;
    str = new char[size+1] ;
    for ( int i = 0 ; i < size ; i ++ ) str[i] = another.str[i] ;
    str[size] ='₩0' ;
  }
  ~MyString() { delete [] str ; } // 소멸자
} ;
```

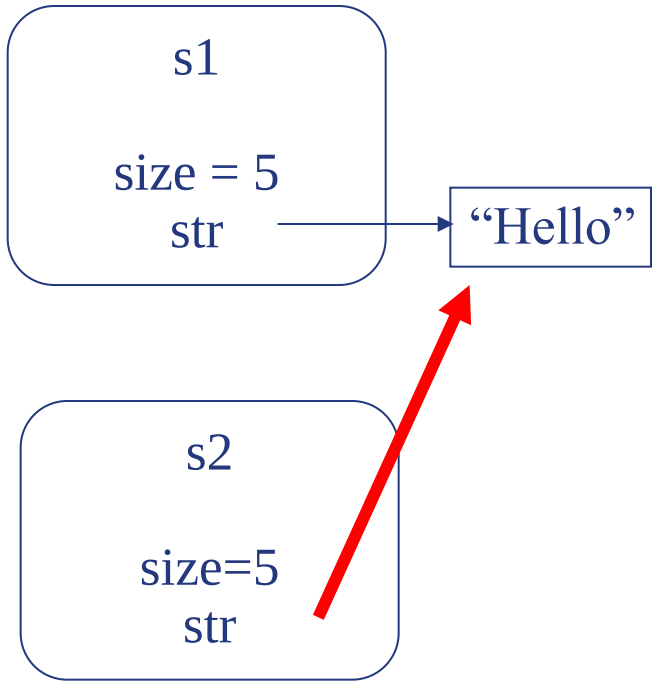
```
int main() {
    MyString s1("Hello") ;
    {
        MyString s2("") ;
        s2 = s1 ;
        cout << s1 << endl ;
    }
    cout << s1 ;
    cout << "End" ;
}
```

Shallow Copy

MyString s1("Hello") ;



s2 = s1 ;



=, [], << Operators

```
int main() {
    MyString str1("C++"), str2("C--") ;

    // << 연산자
    cout << str1 << endl << str2 << endl ;
    if ( str1 == str2 )
        cout << "They are same" << endl ;
    else
        cout << "They are different" << endl ;

    // = 연산자테스트
    MyString str3(str1) ;
    {
        MyString str4 ;
        str4 = str1 ;
    }
    cout << str1 << endl ;

    // [] 연산자테스트
    for ( int i = 0 ; i < str2.length() ; i ++ )
        cout << str2[i] ;

    str2[1] = str2[2] = '+' ;
    cout << endl << str2 << endl ;
}
```

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

const int STRING_DEFAULT_LENGTH = 100 ;
class MyString {
    char* str ;
    int size ;
public:
    MyString(int size=STRING_DEFAULT_LENGTH) {
        this->size = size ;
        str = new char[size+1] ;
        str[0] = '\0' ;
    }
    MyString(const MyString& another) {
        size = another.size ;
        str = new char[size+1] ;
        for ( int i = 0 ; i < size ; i ++ ) str[i] = another.str[i] ;
        str[size] ='\0' ;
    }
```

```
    MyString(const char* const s) {
        assert (s != 0) ;
        size = strlen(s) ;
        str = new char[size+1] ;
        for ( int i = 0 ; i < size ; i ++ ) str[i] = s[i] ;
        str[size] ='\0' ;
    }
    ~MyString() { delete [] str ; }
    MyString& operator = (const MyString& another) {
        delete [] str ;
        size = another.size ;
        str = new char[size+1] ;
        for ( int i = 0 ; i < size ; i ++ ) str[i] = another.str[i] ;
        str[size] ='\0' ;
        return *this ;
    }
    int length() const { return size ; }
    // char at(int i) const { assert ( i >= 0 && i < size) ; return str[i]; }
    char& operator [] (int i) { assert ( i >= 0 && i < size) ; return str[i]; }
    //bool isEqual(const MyString& str) {
    bool operator == (const MyString& str) { return stricmp(this->str, str.str) == 0 ; }
    // void print() const { cout << str << endl ; }
    friend ostream& operator << (ostream& os, const MyString& s) ;
} ;
ostream& operator << (ostream& os, const MyString& s) {
    os << s.str ;
    return os ;
}
```

Increment & Decrement Operator

```
#include <iostream>
using namespace std ;
class Complex {
   double re, im ;
public:
   Complex(double re=0, double im=0) { this->re = re ; this->im = im ; }
   // prefix ++
   Complex& operator ++ () { re++ ; return *this ; }
   // postfix ++
   Complex operator++(int) { return Complex(re++, im) ; }
   friend ostream& operator << (ostream& os, const Complex& c) ;
} ;
ostream& operator << (ostream& os, const Complex& c) {
   os << c.re << " + " << c.im << "i" ;
   return os ;
}
```

```
int main() {
  Complex c1(1, 1) ;

  Complex c2 =  c1 ++ ;
  cout << c1 << " " << c2 << endl ;

  Complex c3 = ++ c1 ;
  cout << c1 << " " << c3 << endl ;
}
```

Conversion Operator

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

class Complex {
    double re, im ;
public:
    Complex(double re=0, double im=0) { this->re = re ; this->im = im ; }
    operator double() { return re ; }
    friend ostream& operator << (ostream& os, const Complex& c) ;
} ;
ostream& operator << (ostream& os, const Complex& c) {
    os << c.re << " + " << c.im << "i" ;
    return os ;
}
int main() {
    Complex c1(1, 0), c2(2, 1) ;

    double re1 = c1 ;
    double re2 = 10 + c2 ;
    cout <<re1 << endl << re2 << endl ;
}
```

Exercise #1

- Implement class Complex to support the following code

```
int main()
{
  Complex c1(10), c2(20), c3 ;

  c3 = c1 + c2 ;
  c3 -= c1 ;

  double r = c1 ;

  cout << c1 << c2 << c3 ;
}
```

Exercise #2

- Implement class Matrix for main() to run like this.

```
class Matrix {
    int** values ;
    int row, column ;
public:
    …
} ;
```

```
int main() {
    Matrix m1(2, 2), m2(2, 2) ;
    cin >> m1 ;
    cin >> m2 ;

    Matrix m3 = m1 + m2 ;
    Matrix m4 = m1 * 10 ;

    cout << m3 << endl ;
    cout << m4 << endl ;
}
```

```
1 2 3 4
5 6 7 8
6       8
10      12

60      80
100     120
```