

CS481 - Object-Oriented Programming



Operator Overloading

- ❖ Operator is just a function
 - Syntactic convenience (“sugar”)
- ❖ Can overload functions
 - By argument types (signature)
- ❖ So, can overload operators too
 - At least one user-defined type

Operator Overloading

- ❖ Can NOT use it to:
 - change definition of built-in operators
 - ◆ would be too confusing!
 - add new operators to the language:
 - ◆ Compiler wouldn't know how to parse!
- ❖ Use it to make user code more understandable!

What We Want

```
Complex c1;
Complex c2;
Complex c3;

c3 = c1 + c2; // binary op
c2 = -c1;     // unary op
c1 = c3;     // assignment
c2 += c1;
```

Operator Functions

- ❖ Special function name:
 - Keyword operator plus symbol
 - ◆ E.g., operator+, operator<<
- ❖ Operator function syntax:
 - Unary vs. binary operator
 - Global function vs. member function

Four Possibilities

Best Practices	Global	Member
Unary	-, ++, --	*, (), []
Binary	+, -, *, /, ==, !=, <, >, <<, >>, >=, <=, &&, &&&	+, +=, -=, *=, /=, %=, -> , !

CS481 - Object-Oriented Programming

Global Binary Ops

$$a = \underline{b} + \underline{c}i$$

- ❖ Two arguments to function
 - Left and right operands
 - Same or different types
 - ◆ At least one user-defined (friend)
- ❖ Return value is result
 - Usually a newly constructed object

7

Global Binary +

Complex "+" operator declaration

```
class Complex
{
    double re;
    double im;
public:
    ...
    friend const Complex operator+
        (const Complex& c1,
         const Complex& c2);
};
```

8

Global Binary +

Function definition

```
// inline
const Complex operator+
    (const Complex& c1,
     const Complex& c2)
{
    return Complex(c1.re+c2.re,
                  c1.im+c2.im);
}
```

9

Global Binary +

Constructor in return eliminates temporary variable

```
// inline
const Complex operator+
    (const Complex& c1,
     const Complex& c2)
{
    return Complex(c1.re+c2.re,
                  c1.im+c2.im);
}
```

10

Member Binary Ops

$$a = \boxed{b} + \underline{c}i$$

- ❖ Object of class is first operand
 - Left hand side of operator
 - Specified in function by `this` pointer
- ❖ Second operand is argument
 - May be same class, or another type
 - Often passed as `const` reference

11

Member Binary Ops

- ❖ Return value
 - Result value for normal operators
 - ◆ E.g., `c = a+b; // return a+b`

12

CS481 - Object-Oriented Programming

Member Binary Ops

❖ Return value

- Result value for normal operators
 - ◆ E.g., `c = a+b; // return a+b`
- Reference to left operand for assignment operators
 - ◆ E.g., `a+=b; // return a`
 - As in: `c = (a+=b);`

13

Member Binary +

Member binary function: "+"

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

14

Member Binary +

Return expression value

```
// inline
const Complex Complex::operator+
    (const Complex& c) const
{
    return Complex(re+c.re,
                  im+c.im);
}
```

15

Member Binary +=

Member binary function: "+="

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

16

Member Binary +=

Return ref so expression value can be used

```
// inline
const Complex& Complex::operator+=
    (const Complex& c)
{
    re += c.re;
    im += c.im;
    return *this; // return ref
}
```

17

Global Unary Ops

❖ Usually one argument!

- Operand value
 - ◆ Must be user-defined type (friend)
- ❖ Return value is result
 - Similar to binary operators

18

CS481 - Object-Oriented Programming

Global Unary -

```
class Complex
...
friend const Complex operator-
    (const Complex& c);
};

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

19

Member Unary Ops

- ❖ Object of class is operand
 - Specified in function by `this` pointer
- ❖ Usually no arguments!!
- ❖ Return value
 - Similar to binary operators

20

Member Unary -

```
class Complex
...
const Complex operator-()
    const;
};

const Complex Complex::operator-()
    const
{
    return Complex(-re, -im);
}
```

21

Special Case: ++, --

- ❖ Distinguish two cases
 - Prefix (`++c`, `--c`)
 - Postfix (`c++`, `c--`)
- ❖ Postfix has dummy `int`
 - Second arg for global function
 - Only arg for member function

22

++ and -- Operators

Global, unary operators

- ❖ Prefix signature:
 - friend const Complex
 - operator++(const Complex& c);
- ❖ Postfix signature:
 - friend const Complex
 - operator++(const Complex& c,
 - int);

23

Subscript Operator

```
class MyArray
{
    int aa[10];
public:
    int& operator[](unsigned int ss)
    { assert(ss < 10);
      return aa[ss]; }
};
```

MyArray x; ... ; x[3] = x[2];

24

CS481 - Object-Oriented Programming

Subscript Operator

```
class MyArray
{
  int aa[10];
public:
  int& operator[](unsigned int ss)
  { assert(ss < 10);
    return aa[ss]; }
};
```

Subscript could also be a non-integral type!

```
MyArray x; ... ; x[3] = x[2];
```

25

Subscript Operator

```
class MyArray
{
  int aa[10];
public:
  int& operator[](unsigned int ss)
  { assert(ss < 10);
    return aa[ss]; }
};
```

Returning a reference allows []'s on both sides of = sign.

```
MyArray x; ... ; x[3] = x[2];
```

26

More Complex Ops

For experts only?

❖ Comma operator

-operator,

❖ Function call operator

-operator()

❖ Smart pointer operator

-operator->

27

Non-Overloadables

❖ Member selection

- "." (dot) operator

❖ Pointer to member

- "*" operator

❖ FORTRAN exponentiation

- "**" operator; can't create it

28

operator =

❖ Must be a member function

❖ Called on assignment

- Not on initialization!

❖ Compiler generates default

- Memberwise assignment (bit copy?)

- May fail if pointer to heap memory²⁹

operator =

```
class MC {... // for MyClass
  char *str;
public:
  MC& operator=(const MC& mc);
  ...};
```

```
MC& MC::operator=(const MC& mc)
{
  free(str);
  str = malloc(strlen(mc.str)+1);
  strcpy(str, mc.str);
  return *this;
}
```

30

CS481 - Object-Oriented Programming

Example: B=A

```

MC& MC::operator=(const MC& mc)
{
  free(str);
  str = malloc(strlen(mc.str)+1);
  strcpy(str, mc.str);
  return *this; }
    
```

Example: B=A

```

MC& MC::operator=(const MC& mc)
{
  free(str);
  str = malloc(strlen(mc.str)+1);
  strcpy(str, mc.str);
  return *this; }
    
```

Example: B=A

```

MC& MC::operator=(const MC& mc)
{
  free(str);
  str = malloc(strlen(mc.str)+1);
  strcpy(str, mc.str);
  return *this; }
    
```

Example: B=A

```

MC& MC::operator=(const MC& mc)
{
  free(str);
  str = malloc(strlen(mc.str)+1);
  strcpy(str, mc.str);
  return *this; }
    
```

Example: B=A

```

MC& MC::operator=(const MC& mc)
{
  free(str);
  str = malloc(strlen(mc.str)+1);
  strcpy(str, mc.str);
  return *this; }
    
```

Canonical Form

For most (?) classes

- ❖ Ordinary constructors (alloc/init)
 - Including a default constructor (?)
- ❖ Copy constructor (alloc/copy)
- ❖ operator= (free/alloc/copy)
- ❖ Destructor (free/clean up)

CS481 - Object-Oriented Programming

Global vs Member

Operator	Method
All unary	Member recommended
= () [] ->	Must be member
+= -= etc.	Member recommended
Other binary	Global (friend) recommended

From Rob Murray

37

Type Conversion

- ❖ Automatic (as needed)
- ❖ Two methods (convert A to B)
 - Single-argument constructor
 - ◆ `B::B(const A& a_obj);`

38

Type Conversion

- ❖ Automatic (as needed)
- ❖ Two methods (convert A to B)
 - Single-argument constructor
 - ◆ `B::B(const A& a_obj);`

Warning: will accept any matching single-argument constructor!

39

Type Conversion

- ❖ Automatic (as needed)
- ❖ Two methods (convert A to B)
 - Single-argument constructor
 - ◆ `B::B(const A& a_obj);`
 - Operator conversion
 - ◆ `A::operator B() const;`

40

Operator Cautions

- ❖ Don't get carried away!
 - Redefine "+" to mean subtraction??
 - Overloading is supposed to simplify
- ❖ Ambiguous type conversion
 - Use automatic conversion only when it is really helpful

41