

Operator Overloading

Professor Hugh C. Lauer

CS-2303, System Programming Concepts

(Slides include materials from *The C Programming Language*, 2nd edition, by Kernighan and Ritchie, *Absolute C++*, by Walter Savitch, *The C++ Programming Language*, Special Edition, by Bjarne Stroustrup, and from *C: How to Program*, 5th and 6th editions, by Deitel and Deitel)

Why Operator Overloading?

- **Readable code**
- **Extension of language to include user-defined types**
 - I.e., classes
- **Make operators sensitive to context**
- **Generalization of function overloading**

Simple Example

```
class complex {  
    double real, imag;  
public:  
    complex(double r, double i) :  
        real(r), imag(i) {}  
}
```

- Would like to write:–

```
complex a = complex(1, 3.0);  
complex b = complex(1.2, 2);  
complex c = b;
```

```
a = b + c;  
b = b + c*a;  
c = a*b + complex(1, 2);
```

I.e., ordinary arithmetic expressions for this user-defined class.

Operator Overloading

```
class complex {  
    double real, imag;  
public:  
    complex(double r, double i) :  
        real(r), imag(i) {}  
  
    complex operator +(complex a, complex b);  
    complex operator *(complex a, complex b);  
    complex& operator =(complex a, complex b);  
    ...  
}
```

General Format

returnType operator(parameters);*

↑ ↑ ↑
any type keyword operator symbol

- ***Return type*** may be whatever the operator returns
 - Including a reference to the object of the operand
- ***Operator symbol*** may be any ***overloadable operator*** — i.e., all except
 - `::` (scope resolution), `_` (member selection), `.*` (selection through pointer to member), `?:` (conditional expression)

Operators that Can and Cannot be Overloaded

Operators that can be overloaded

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

Deitel & Deitel, Figure 22.1

Operators that cannot be overloaded

.	.*	::	?:
---	----	----	----

Deitel & Deitel, Figure 22.2

C++ Philosophy

■ **All operators have context**

- Even the simple “built-in” operators of basic types
- E.g., '+', '-', '*', '/' for numerical types
- Compiler generates different code depending upon type of operands

■ **Operator overloading is a generalization of this feature to non-built-in types**

- E.g., '<<', '>>' for bit-shift operations and also for stream operations

**No counterpart
in Java!**

C++ Philosophy (continued)

- Operators *retain* their precedence and associativity, even when overloaded
- Operators *retain* their number of operands
- Cannot to define new operators
 - Only (a subset of) the built-in C++ operators can be overloaded
- Cannot redefine operators on built-in types

Outline

- **Fundamentals of Operator Overloading**
- **Restrictions on Operator Overloading**
- **Operator Functions as Class Members vs. Global Functions**
- **Overloading Stream Insertion and Stream Extraction Operators**

Operator Overload Function

■ Either

- a non-*static* member function definition

or

- a global function definition
 - Usually a *friend* of the class

■ Function “name” is keyword *operator* followed by the symbol for the operation being overloaded

- E.g., *operator+*, *operator=*, *operator->*, *operator()*

Operator Overload Function (continued)

- Operator overload function is a function just like any other

- Can be called like any other – e.g.,

a.operator+(b)

- C++ provides the following short-hand

a+b

Operator Overload Function (continued)

- If operator overload function is declared as a global or *friend*, then

operator+(a, b)

- also reduces to the following short-hand

a+b

Operator Overloading (continued)

- To use any operators on a class object, ...
 - The operator must be overloaded for that class.
- Three Exceptions: {overloading allowed but not required}
 - Assignment operator (=)
 - Memberwise assignment between objects
 - Dangerous for classes with pointer members!!
 - Address operator (&)
 - Returns address of the object in memory.
 - Comma operator (,)
 - Evaluates expression to its left then the expression to its right.
 - Returns the value of the expression to its right.

Questions?

Operator Functions as Class Members

- Leftmost operand must be of *same class* as operator function.
- Use *this* keyword to implicitly get left operand argument.
- Operators *()*, *[]*, *->* *or* any assignment operator *must* be overloaded as a class member function.
- Called when
 - Left operand of binary operator is of *this* class
 - Single operand of unary operator is of *this* class

Operator Functions as Global Members

- Need parameters for both operands.
- Can have object of different class than operator.
- Can be made a *friend* to access *private* or *protected* data.

© 2007 Pearson Ed -All rights reserved.

Stream Insertion/Extraction Operators

- Typically global or friend functions
- **Overload << operator used where**
 - Left operand of type *ostream* &
 - Such as *cout* object in *cout << classObject*
- **Overload >> has left operand of *istream* &**
 - Left operand of type *istream* &
 - Such as *cin* object in *cout >> classObject*
- **Reason:—**
 - These operators are associated with class of *right* operand

Commutative Operators

- **May need '+' (and others) to be commutative**
 - So both " $a + b$ " and " $b + a$ " work as expected.
- **Suppose we have two different classes**
 - Overloaded operator can only be member function when its class is on left.
 - *HugeIntClass + long int*
 - May be member function
 - For the other way, you need a global overloaded *friend* function
 - *long int + HugeIntClass*

Digression

friends and this

Ordinary Member Functions

- Function can access the private members of the class
- Function is in the scope of the class
- Function must be invoked on a specific object of the class – e.g.,
 - *ptr -> func()*
 - *obj.func()*

static Member Function

- Function can access the private members of the class
- Function is in the scope of the class
- Function must be invoked on a specific object of the class – e.g.,
 - *ptr -> func()*
 - *obj.func()*
- **Can access only the *static* members**
 - Members that exist independently of any objects

friend Function

- **Function can access the private members of the class**
- Function is in the scope of the class
- Function must be invoked on a specific object of the class – e.g.,
 - *ptr -> func()*
 - *obj.func()*

friend Function of a Class

- Defined outside of class's scope
- Not a member function of that class
- Has right to access *non-public* and public members of that class
- Often appropriate when a member function cannot be used for certain operations
- Can enhance performance

friend Functions and friend Classes

- **To declare a function as a *friend* of a class:–**
 - Provide the function prototype in the class definition preceded by keyword *friend*

- **To declare a class as a *friend* of another class:**
 - Place declaration of the form

friend class ClassTwo;

in definition of class *ClassOne*
 - All member functions of class *ClassTwo* become friends of class *ClassOne*

friend Functions and friend Classes (continued)

■ Friendship is granted, not taken

- For *class B* to be a friend of *class A*, *class A* must explicitly declare *class B* as a friend

■ Friendship relation is neither symmetric nor transitive

- If *class A* is a friend of *class B*, and *class B* is a friend of *class C*, cannot infer that
 - ! *class B* is a friend of *class A*
 - ! *class C* is a friend of *class B*
 - ! *class A* is a friend of *class C*

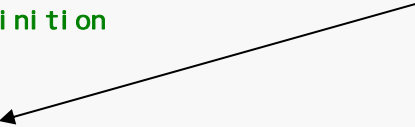
friend Functions and friend Classes (continued)

- ...
- It is possible to specify overloaded functions as *friends* of a class.
 - Each overloaded function intended to be a *friend* must be explicitly declared as a *friend* of the class.

friend Function Example

```
1 // Fig. 21.15: fig21_15.cpp
2 // Friends can access private members of a class.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // Count class definition
8 class Count
9 {
10     friend void setX( Count &, int ); // friend declaration
11 public:
12     // constructor
13     Count()
14         : x( 0 ) // initialize x to 0
15     {
16         // empty body
17     } // end constructor Count
18
19     // output x
20     void print() const
21     {
22         cout << x << endl;
23     } // end function print
24 private:
25     int x; // data member
26 }; // end class Count
```

friend function declaration (can appear anywhere in the class)



friend Function Example (continued)

```
27
28 // function setX can modify private data of Count
29 // because setX is declared as a friend of Count (line 10)
30 void setX( Count &c, int val )
31 {
32     c.x = val; // allowed because setX is a friend of Count
33 } // end function setX
34
35 int main()
36 {
37     Count counter; // create Count object
38
39     cout << "counter.x after instantiation: ";
40     counter.print();
41
42     setX( counter, 8 ); // set x using a friend function
43     cout << "counter.x after call to setX friend function: ";
44     counter.print();
45     return 0;
46 } // end main
```

friend function can modify Count's private data

Calling a friend function; note that we pass the Count object to the function

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

Questions about friends?

Example

■ << and >> operators

- Already overloaded by STL to process each built-in type (pointers and strings)
- Can also process a user-defined class
 - Overload using global, friend functions

■ Example program

- *class PhoneNumber* (on next slide)
 - Holds a telephone number
- Prints out formatted number automatically.
 - *(312) 456-7890*

Example (continued)

```

1 // Fig. 22.3: PhoneNumber.h
2 // PhoneNumber class definition
3 #ifndef PHONENUMBER_H
4 #define PHONENUMBER_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 #include <string>
11 using std::string;
12
13 class PhoneNumber
14 {
15     friend ostream &operator<<( ostream &, const PhoneNumber & );
16     friend istream &operator>>( istream &, PhoneNumber & );
17 private:
18     string areaCode; // 3-digits
19     string exchange; // 3-digits
20     string line; // 4-digits
21 }; // end class PhoneNumber
22
23 #endif

```

Note also: reference results!

Notice function prototypes for overloaded operators
>> and << (must be global *friend* functions)

Example (continued)

```

1 // Fig. 22.4: PhoneNumber.cpp
2 // Overloaded stream insertion and stream extraction operators
3 // for class PhoneNumber.
4 #include <iomanip>
5 using std::setw;
6
7 #include "PhoneNumber.h"
8
9 // overloaded stream insertion operator; cannot be
10 // a member function if we would like to invoke it with
11 // cout << somePhoneNumber;
12 ostream &operator<<( ostream &output, const PhoneNumber &number )
13 {
14     output << "(" << number.areaCode << ")" "
15         << number.exchange << "-" << number.line;
16     return output; // enables cout << a << b << c;
17 } // end function operator<<

```

Allows *cout << phone;* to be interpreted as:
operator<<(cout, phone);

Display formatted phone number

Example (continued)

```
18
19 // overloaded stream extraction operator; cannot be
20 // a member function if we would like to invoke it with
21 // cin >> somePhoneNumber;
22 istream &operator>>( istream &input, Phone
23 {
24     input.ignore(); // skip (
25     input >> setw( 3 ) >> number.areaCode; // input area code
26     input.ignore( 2 ); // skip ) and space
27     input >> setw( 3 ) >> number.exchange; // input exchange
28     input.ignore(); // skip dash (-)
29     input >> setw( 4 ) >> number.line; // input line
30     return input; // enables cin >> a >> b >> c;
31 } // end function operator>>
```

ignore skips specified number of characters from input (1 by default)

Input each portion of phone number separately

Example (concluded)

```

1 // Fig. 22.5: fig22_05.cpp
2 // Demonstrating class PhoneNumber's overloaded stream insertion
3 // and stream extraction operators.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "PhoneNumber.h"
10
11 int main()
12 {
13     PhoneNumber phone; // create object phone
14
15     cout << "Enter phone number in the form (123) 456-7890:" << endl;
16
17     // cin >> phone invokes operator>> by implicitly issuing
18     // the global function call operator>>( cin, phone )
19     cin >> phone;
20
21     cout << "The phone number entered was: ";
22
23     // cout << phone invokes operator<< by implicitly
24     // the global function call operator<<( cout, phone )
25     cout << phone << endl;
26     return 0;
27 } // end main

```

Invoke overloaded >> and << operators to input and output a *PhoneNumber* object

Questions?

Unary Operators

■ Can overload as

- Non-*static* member function with no arguments
- As a global function with one argument
 - Argument must be class object or reference to class object

■ Why non-*static*?

- *static* functions only access *static* data
- Not what is needed for operator functions

Another Example

- Overload '!' to test for empty string – e.g.,

while (!s) ...
if (!s) ...

compiler generates call to

s.operator!()

- Implemented as:–

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

Overloading Binary Operators

- Non-*static* member function with one argument.

or

- Global function with two arguments:

- One argument must be class object or reference to a class object.

← This is mechanism by which compiler prevents you from redefining built-in operations!

Overloading Binary Operators (continued)

- If a non-static member function, it needs one argument.

```
class String {  
    public:  
        String & operator+=( const String &);  
    ...  
};
```

- By shorthand rule

y += z becomes *y.operator+=(z)*

Overloading Binary Operators (continued)

- Global (*friend*) function needs two arguments

```
class String {  
    public:  
        String & operator+=( String &, const String & );  
    ...  
};
```

- By short-hand rule

y += z becomes *operator+=(y, z)*

Overloading Operators

- On the previous slide, **y** and **z** are assumed to be *String-class* objects or references to *String-class* objects.
- **Two ways to pass arguments to global function:–**
 - An object (requires a copy of object)
 - A reference to an object (function operates on object directly!)