

CISC181 Introduction to Computer Science

Dr. McCoy

Lecture 25
December 1, 2009

1

8.1 Introduction

- Use operators with objects (operator overloading)
 - Clearer than function calls for certain classes
 - Operator sensitive to context
- Examples
 - <<
 - Stream insertion, bitwise left-shift
 - +
 - Performs arithmetic on multiple types (integers, floats, etc.)
- Will discuss when to use operator overloading

© 2001 Pearson Education, Inc. All rights reserved. [Navigation icons]

8.2 Fundamentals of Operator Overloading

- Types
 - Built in (`int`, `char`) or user-defined
 - Can use existing operators with user-defined types
 - Cannot create new operators
- Overloading operators
 - Create a function for the class
 - Name function `operator` followed by symbol
 - `operator+` for the addition operator `+`

© 2001 Pearson Education, Inc. All rights reserved. [Navigation icons]

8.2 Fundamentals of Operator Overloading

- Using operators on a class object
 - It must be overloaded for that class
 - Exceptions:
 - Assignment operator, `=`
 - Memberwise assignment between objects
 - Address operator, `&`
 - Returns address of object
 - Both can be overloaded
- Overloading provides concise notation
 - `object2 = object1.add(object2);`
 - `object2 = object2 + object1;`

© 2001 Pearson Education, Inc. All rights reserved. [Navigation icons]

8.3 Restrictions on Operator Overloading

- Cannot change
 - How operators act on built-in data types
 - I.e., cannot change integer addition
 - Precedence of operator (order of evaluation)
 - Use parentheses to force order-of-operations
 - Associativity (left-to-right or right-to-left)
 - Number of operands
 - `&` is unitary, only acts on one operand
- Cannot create new operators
- Operators must be overloaded explicitly
 - Overloading `+` does not overload `+=`

© 2001 Pearson Education, Inc. All rights reserved. [Navigation icons]

8.3 Restrictions on Operator Overloading

Operators that can be overloaded							
<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>++</code>	<code>--</code>	<code>!</code>
<code>-</code>	<code>!</code>	<code>-</code>	<code><</code>	<code>></code>	<code>++</code>	<code>--</code>	<code>!-</code>
<code>/=</code>	<code>!-</code>	<code>*=</code>	<code>=</code>	<code>!-</code>	<code><<</code>	<code>>></code>	<code><<=</code>
<code><<=</code>	<code>==</code>	<code>!-</code>	<code><=</code>	<code>>=</code>	<code>!!</code>	<code>!!</code>	<code>==</code>
<code>--</code>	<code>-></code>	<code>.</code>	<code>-></code>	<code>!!</code>	<code>!</code>	<code>new</code>	<code>delete</code>
<code>new[]</code>	<code>delete[]</code>						

Operators that cannot be overloaded				
<code>.</code>	<code>*</code>	<code>!</code>	<code>!</code>	<code>sizeof</code>

© 2001 Pearson Education, Inc. All rights reserved. [Navigation icons]

8.4 Operator Functions As Class Members Vs. As Friend Functions

- Operator functions
 - Member functions
 - Use `this` keyword to implicitly get argument
 - Gets left operand for binary operators (like +)
 - Leftmost object must be of same class as operator
 - Non member functions
 - Need parameters for both operands
 - Can have object of different class than operator
 - Must be a **friend** to access **private** or **protected** data
 - Called when
 - Left operand of binary operator of same class
 - Single operand of unitary operator of same class

© 2011 Pearson Education, Inc. All rights reserved. [Navigation icons]

8.4 Operator Functions As Class Members Vs. As Friend Functions

- Overloaded << operator
 - Left operand of type `ostream &`
 - Such as `cout` object in `cout << classObject`
 - Similarly, overloaded >> needs `istream &`
 - Thus, both must be non-member functions

© 2011 Pearson Education, Inc. All rights reserved. [Navigation icons]

8.4 Operator Functions As Class Members Vs. As Friend Functions

- Commutative operators
 - May want + to be commutative
 - So both "`a + b`" and "`b + a`" work
 - Suppose we have two different classes
 - Overloaded operator can only be member function when its class is on left
 - `HugeIntClass + long int`
 - Can be member function
 - When other way, need a non-member overload function
 - `long int + HugeIntClass`

© 2011 Pearson Education, Inc. All rights reserved. [Navigation icons]

8.5 Overloading Stream-Insertion and Stream-Extraction Operators

- << and >>
 - Already overloaded to process each built-in type
 - Can also process a user-defined class
- Example program
 - Class `PhoneNumber`
 - Holds a telephone number
 - Print out formatted number automatically
 - `(123) 456-7890`

© 2011 Pearson Education, Inc. All rights reserved. [Navigation icons]

```

1 // fig. 8.4: fig08_04.cpp
2 // overloading the stream-insertion and
3 // stream-extraction operators.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8 using std::endl;
9 using std::endl;
10 using std::endl;
11
12 #include <string>
13
14 using std::endl;
15
16 // PhoneNumber class definition
17 class PhoneNumber {
18     friend ostream& operator<<(ostream&, const PhoneNumber&);
19     friend istream& operator>>(istream&, PhoneNumber&);
20 };
21 private:
22     char areaCode[ 3 ]; // 123
23     char exchange[ 3 ]; // 456
24     char line[ 8 ]; // 78901234 and null
25 }; // end class PhoneNumber
    
```

Outline
fig08_04.cpp (3 of 3)

Notice function prototypes for overloaded operators << and >>. They must be non-member **friend** functions, since the object of class `PhoneNumber` appears on the right of the operator:
`cin << object`
`cout >> object`

© 2011 Pearson Education, Inc. All rights reserved.

```

27
28 // overloaded stream-insertion operator: cannot be
29 // a member function if we would like to cascade it with
30 // cin as described below.
31 ostream& operator<<(ostream& out, const PhoneNumber& num) {
32     {
33         num.out << " as non-overloaded on " << num;
34         num.out << " as non-overloaded on " << num;
35     }
36     return out; // enables use as a as b as c
37 } // end function operator<<
38
39 // overloaded stream-extraction operator: cannot be
40 // a member function if we
41 // cin as described below.
42 istream& operator>>(istream& in, PhoneNumber& num) {
43     {
44         num.ignore(); // skip specified
45         num.ignore( 3 ); // skip 3
46         num.ignore( 3 ); // skip 3
47         num.ignore( 3 ); // skip 3
48         num.ignore( 3 ); // skip 3
49         num.ignore( 3 ); // skip 3
50         num.ignore( 3 ); // skip 3
51         num.ignore( 3 ); // skip 3
52     }
53     return num; // enables use
    
```

Outline
fig08_05.cpp (3 of 3)

The expression: `cout << phone;` is interpreted as the function call: `operator<<(cout, phone);` output is as also for `cout`.

This allows objects to be cascaded: `<< phone1 << phone2;` will be interpreted as `operator<<(cout, phone1);` and `operator<<(cout, phone2);` Next `cout << phone2` executes.

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

Stream manipulator `setw` restricts number of characters read. `setw(4)` allows 3 characters to be read, leaving room for the null character.

© 2011 Pearson Education, Inc. All rights reserved.

```

85 | // end function operators
86 |
87 | int main()
88 | {
89 |     //overload phone: // access object phone
90 |     auto a = "over phone number in the form (123) 555-1234";
91 |
92 |     // use as phone function operators by explicitly loading
93 |     // the non-member function call operators: a(a, phone)
94 |     a(a, phone);
95 |
96 |     auto b = "the phone number entered was: ";
97 |
98 |     // use as phone function operators by explicitly loading
99 |     // the non-member function call operators: a(a, phone)
100 |    a(a, phone, a, a);
101 |
102 |    return 0;
103 | }
104 | // end main

```

over phone number in the form (123) 555-1234
(123) 555-1234
the phone number entered was: (123) 555-1234

8.6 Overloading Unary Operators

- Overloading unary operators
 - Non-**static** member function, no arguments
 - Non-member function, one argument
 - Argument must be class object or reference to class object
 - Remember, **static** functions only access **static** data

8.6 Overloading Unary Operators

- Upcoming example (8.10)
 - Overload ! to test for empty string
 - If non-**static** member function, needs no arguments
 - !a becomes a.operator!()

```

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

```
 - If non-member function, needs one argument
 - a! becomes operator!(a)

```

class String {
    friend bool operator!( const String & )
    ...
};

```

8.7 Overloading Binary Operators

- Overloading binary operators
 - Non-**static** member function, one argument
 - Non-member function, two arguments
 - One argument must be class object or reference
- Upcoming example
 - If non-**static** member function, needs one argument

```

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

```
 - **y += z** equivalent to **y.operator+=(z)**

8.7 Overloading Binary Operators

- Upcoming example
 - If non-member function, needs two arguments
 - Example:

```

class String {
    friend const String &operator+=(
        String &, const String & );
    ...
};

```
 - **y += z** equivalent to **operator+=(y, z)**