

Operator Overloading in C++



Systems Programming

Operator Overloading

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

Operator Overloading

- Overloading Unary Operators
- Overloading Binary Operators
- Case Study: Array Class
- Case Study: String Class N
- Case Study: A Date Class N
- Standard Library Class string N
- explicit Constructors N

Introduction

- Users can use operators with user-defined types (e.g., with objects {**operator overloading**}).
 - Clearer than function calls for certain classes.
 - C++ makes operators sensitive to context.
- Examples
 - <<
 - Stream insertion, bitwise left-shift
 - +
 - Performs arithmetic on multiple items (integers, floats, etc.)

© 2007 Pearson Ed -All rights reserved.

Operator Overloading

- An operator is overloaded by writing a non-**static** member function definition or a global function definition except that the function name becomes the keyword **operator** followed by the symbol for the operation being overloaded.

Operator Overloading

- Types for operator overloading
 - Built in (`int`, `char`) or user-defined (classes)
 - Can use existing operators with user-defined types.
 - **Cannot create new operators!**
- Overloading operators
 - Create a function for the class.
 - Name of operator function.
 - Keyword `operator` followed by the symbol

Example

function name `operator+` for the addition
operator `+`

© 2007 Pearson Ed -All rights reserved.

Operator Overloading

- Using operators on a class object:
 - The operator must be overloaded for that class.
- Three Exceptions: {overloading 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.

- Overloading provides concise notation

```
obj ect2 = obj ect1.add( obj ect2 );
```

vs.

```
obj ect2 = obj ect2 + obj ect1;
```

© 2007 Pearson Ed -All rights reserved.

Restrictions on Operator Overloading

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

© 2007 Pearson Ed -All rights reserved.

Fig. 22.1 Operators that can be overloaded.

Operators that can be overloaded

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

© 2007 Pearson Ed -All rights reserved.

Fig. 22.1 Operators that cannot be overloaded.

Operators that cannot be overloaded

.

.*

::

?:

© 2007 Pearson Ed -All rights reserved.

Software Engineering Observation 22.2

- At least one argument of an operator function must be an object or reference of a **user-defined type**.
- This prevents programmers from changing how operators work on fundamental types.

© 2007 Pearson Ed -All rights reserved.

22.4 Operator Functions as Class Members vs. Global Members

- Operator functions as member functions:
 - Leftmost object 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.

© 2007 Pearson Ed -All rights reserved.

22.4 Operator Functions as Class Members vs. Global Members

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

© 2007 Pearson Ed -All rights reserved.

Overloading Stream Insertion and Stream Extraction Operators

- Overloaded `<<` operator used where
 - Left operand of type `ostream &`
 - Such as `cout` object in `cout << classObject`
 - To use the operator in this manner where the right operand is an object of a user-defined class, it must be overloaded as a global function.
 - Similarly, overloaded `>>` has left operand of `istream &`
 - Thus, both must be global functions.

© 2007 Pearson Ed -All rights reserved.

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
 - For the other way, you need a global overloaded function.
 - `long int + HugeIntClass`

© 2007 Pearson Ed -All rights reserved.

22.5 Overloading Stream Insertion and Stream Extraction Operators

- `<<` and `>>` operators
 - Already overloaded to process each built-in type.
 - Can also process a user-defined class.
 - Overload using global, `friend` functions
- Example program
 - Class `PhoneNumber`
 - Holds a telephone number
 - Prints out formatted number automatically.
(123) 456-7890

© 2007 Pearson Ed -All rights reserved.

Overload Stream Insertion and Extraction Operators

```
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-digit area code
19     string exchange; // 3-digit exchange
20     string line; // 4-digit line
21 }; // end class PhoneNumber
22
23 #endif
```

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

© 2007 Pearson Ed -All rights reserved.

Overload Stream Insertion and Extraction Operators

```
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

© 2007 Pearson Ed -All rights reserved.

Overload Stream Insertion and Extraction Operators

```
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, PhoneNumber &number )
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

© 2007 Pearson Ed -All rights reserved.

Overload Stream Insertion and Extraction Operators

```
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 issuing
24     // the global function call operator<<(cout, phone)
25     cout << phone << endl;
26     return 0;
27 } // end main
```

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

© 2007 Pearson Ed -All rights reserved.

Overload Stream Insertion and Extraction Operators

```
Enter phone number in the form (123) 456-7890:  
(800) 555-1212  
The phone number entered was: (800) 555-1212
```

© 2007 Pearson Ed -All rights reserved.

22.6 Overloading Unary Operators

- Overloading unary operators of a class:
 - Can overload as a non-**static** member function with no arguments.
 - Can overload as a global function with one argument.
 - Argument must be class object or reference to class object.
 - Remember, **static** functions only access **static** data.

© 2007 Pearson Ed -All rights reserved.

22.6 Overloading Unary Operators

Example

Overload `!` to test for empty string

- Consider the expression `!s` in which `s` is an object of class `String`. For `!s` the compiler generates the call `s.operator!()`

Namely, since it is a non-`static` member function, it needs no arguments:

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

- If a global function, it needs one argument:
 - `bool operator! (const String &)`
 - `s!` becomes `operator!(s)`

© 2007 Pearson Ed -All rights reserved.

22.7 Overloading Binary Operators

- **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.

© 2007 Pearson Ed -All rights reserved.

22.7 Overloading Binary Operators

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

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

- If a global function, it needs two arguments.

- `const String &operator+=(String &, const String &);`
- `y += z` becomes `operator+=(y, z)`

© 2007 Pearson Ed -All rights reserved.

Overloading Operators

- On the previous slide, *y* and *z* are assumed to be String-class objects or references to String-class objects.
- There are two ways to pass arguments to the global function, either with an argument that is an object (this requires a copy of the object) or with an argument that is a reference to an object (this means the side effects of the function called to implement the overloaded operator can **side-effect** this object that is called-by-reference!)

22.8 Case Study: Array Class

- Problems with pointer-based arrays in C++:
 - No range checking.
 - Cannot be compared meaningfully with `==`
 - No array assignment (array names are `const` pointers).
 - If array passed to a function, size must be passed as a separate argument.

{Basic point of this chapter - by using C++ classes and operator overloading, one can significantly change the capabilities of the built in array type.}

© 2007 Pearson Ed -All rights reserved.

22.8 Case Study: Array Class

Case Study: Implement an **Array** class with:

- Range checking
- Array assignment
- Arrays that know their own size.
- Outputting/inputting entire arrays with **<<** and **>>**
- Array comparisons with **==** and **!=**

© 2007 Pearson Ed -All rights reserved.

22.8 Case Study: Array Class

- **Copy constructor**

- Used whenever copy of object is needed:

- Passing by value (return value or parameter).
- Initializing an object with a copy of another of same type.

`Array newArray(ol dArray);` or

`Array newArray = ol dArray;` (both are identical)

- `newArray` is a copy of `ol dArray`

© 2007 Pearson Ed -All rights reserved.

22.8 Case Study: Array Class

- Prototype for class Array

```
Array( const Array & );
```

- Must take reference

- Otherwise, the argument will be passed by value...
- Which tries to make copy by calling copy constructor...
 - This yields an infinite loop!

© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

© 2007 Pearson Ed -All rights reserved.

```
1 // Fig. 22.6: Array.h
2 // Array class for storing arrays of integers.
3 #ifndef ARRAY_H
4 #define ARRAY_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 class Array
11 {
12     friend ostream &operator<<( ostream &, const Array & );
13     friend istream &operator>>( istream &, Array & );
14 public:
15     Array( int = 10 ); // default constructor
16     Array( const Array & ); // copy constructor
17     ~Array(); // destructor
18     int getSize() const; // return size
19
20     const Array &operator=( const Array & ); // assignment operator
21     bool operator==( const Array & ) const; // equality operator
22
23     // inequality operator; returns opposite of == operator
24     bool operator!=( const Array &right ) const
25     {
26         return ! ( *this == right ); // invokes Array::operator
27     } // end function operator!=
```

Most operators overloaded as member functions (except << and >>, which must be global functions)

Prototype for copy constructor

!= operator simply returns opposite of == operator – only need to define the == operator

Case Study: Array Class

```
28
29 // subscript operator for non-const objects returns modifiable value
30 int &operator[]( int );
31
32 // subscript operator for const objects returns rvalue
33 int operator[]( int ) const;
34 private:
35     int size; // pointer-based array size
36     int *ptr; // pointer to first element of pointer-based array
37 }; // end class Array
38
39 #endif
```

Operators for accessing specific elements of **Array** object

Note: An example of pointer data member

© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

```
1 // Fig 22.7: Array.cpp
2 // Member-function definitions for class Array
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // exit function prototype
13 using std::exit;
14
15 #include "Array.h" // Array class definition
16
17 // default constructor for class Array (default size 10)
18 Array::Array( int arraySize )
19 {
20     size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
21     ptr = new int[ size ]; // create space for pointer-based array
22
23     for ( int i = 0; i < size; i++ )
24         ptr[ i ] = 0; // set pointer-based array element
25 } // end Array default constructor
```

© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

```
26
27 // copy constructor for class Array;
28 // must receive a reference to prevent infinite recursion
29 Array::Array( const Array &arrayToCopy )
30     : size( arrayToCopy.size )
31 {
32     ptr = new int[ size ]; // create space for pointer-based array
33
34     for ( int i = 0; i < size; i++ )
35         ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
36 } // end Array copy constructor
37
38 // destructor for class Array
39 Array::~Array()
40 {
41     delete [] ptr; // release pointer-based array space
42 } // end destructor
43
44 // return number of elements of Array
45 int Array::getSize() const
46 {
47     return size; // number of elements in Array
48 } // end function getSize
```

We must declare a new integer array so the objects do not point to the same memory

© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

```
49
50 // overloaded assignment operator;
51 // const return avoids: ( a1 = a2 ) = a3
52 const Array &Array::operator=( const Array &right )
53 {
54     if ( &right != this ) // avoid self-assignment
55     {
56         // for Arrays of different sizes, deallocate original
57         // left-side array, then allocate new left-side array
58         if ( size != right.size )
59         {
60             delete [] ptr; // release space
61             size = right.size; // resize this object
62             ptr = new int[ size ]; // create space for array copy
63         } // end inner if
64
65         for ( int i = 0; i < size; i++ )
66             ptr[ i ] = right.ptr[ i ]; // copy array into object
67     } // end outer if
68
69     return *this; // enables x = y = z, for example
70 } // end function operator=
```

Want to avoid self assignment

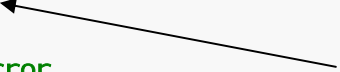
This would be dangerous if **this** is the same **Array** as **right**

© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

```
71
72 // determine if two Arrays are equal and
73 // return true, otherwise return false
74 bool Array::operator==( const Array &right ) const
75 {
76     if ( size != right.size )
77         return false; // arrays of different number of elements
78
79     for ( int i = 0; i < size; i++ )
80         if ( ptr[ i ] != right.ptr[ i ] )
81             return false; // Array contents are not equal
82
83     return true; // Arrays are equal
84 } // end function operator==
85
86 // overloaded subscript operator for non-const Arrays;
87 // reference return creates a modifiable lvalue
88 int &Array::operator[]( int subscript )
89 {
90     // check for subscript out-of-range error
91     if ( subscript < 0 || subscript >= size )
92     {
93         cerr << "\nError: Subscript " << subscript
94             << " out of range" << endl;
95         exit( 1 ); // terminate program; subscript out of range
96     } // end if
97
98     return ptr[ subscript ]; // reference return
99 } // end function operator[]
```

integers1[5] calls
integers1.operator[](5)



© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

```
100
101// overloaded subscript operator for const Arrays
102// const reference return creates an rvalue
103int Array::operator[]( int subscript ) const
104{
105    // check for subscript out-of-range error
106    if ( subscript < 0 || subscript >= size )
107    {
108        cerr << "\nError: Subscript " << subscript
109            << " out of range" << endl ;
110        exit( 1 ); // terminate program; subscript out of range
111    } // end if
112
113    return ptr[ subscript ]; // returns copy of this element
114} // end function operator[]
115
116// overloaded input operator for class Array;
117// inputs values for entire Array
118istream &operator>>( istream &input, Array &a )
119{
120    for ( int i = 0; i < a.size; i++ )
121        input >> a.ptr[ i ];
122
123    return input; // enables cin >> x >> y;
124} // end function
```

© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

```
125
126// overloaded output operator for class Array
127ostream &operator<<( ostream &output, const Array &a )
128{
129    int i;
130
131    // output private ptr-based array
132    for ( i = 0; i < a.size; i++ )
133    {
134        output << setw( 12 ) << a.ptr[ i ];
135
136        if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
137            output << endl;
138    } // end for
139
140    if ( i % 4 != 0 ) // end last line of output
141        output << endl;
142
143    return output; // enables cout << x << y;
144} // end function operator<<
```

© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

```
1 // Fig. 22.8: fig22_08.cpp
2 // Array class test program.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include "Array.h"
9
10 int main()
11 {
12     Array integers1( 7 ); // seven-element Array
13     Array integers2; // 10-element Array by default
14
15     // print integers1 size and contents
16     cout << "Size of Array integers1 is "
17         << integers1.getSize()
18         << "\nArray after initialization:\n" << integers1;
19
20     // print integers2 size and contents
21     cout << "\nSize of Array integers2 is "
22         << integers2.getSize()
23         << "\nArray after initialization:\n" << integers2;
24
25     // input and print integers1 and integers2
26     cout << "\nEnter 17 integers: " << endl;
27     cin >> integers1 >> integers2;
```

Retrieve number of elements in **Array**

Use overloaded >> operator to input

© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

```
28
29 cout << "\nAfter input, the Arrays contain:\n"
30     << "Integers1:\n" << integers1
31     << "Integers2:\n" << integers2;
32
33 // use overloaded inequality (!=) operator
34 cout << "\nEvaluating: integers1 != integers2" << endl;
35
36 if ( integers1 != integers2 )
37     cout << "Integers1 and integers2 are not equal" << endl;
38
39 // create Array integers3 using integers1 as an
40 // initializer; print size and contents
41 Array integers3( integers1 ); // invokes copy constructor
42
43 cout << "\nSize of Array integers3 is "
44     << integers3.GetSize()
45     << "\nArray after initialization:\n" << integers3;
46
47 // use overloaded assignment (=) operator
48 cout << "\nAssigning integers2 to integers1:" << endl;
49 integers1 = integers2; // note target Array is smaller
50
51 cout << "Integers1:\n" << integers1
52     << "Integers2:\n" << integers2;
53
54 // use overloaded equality (==) operator
55 cout << "\nEvaluating: integers1 == integers2" << endl;
```

Use overloaded << operator to output

Use overloaded != operator
to test for inequality

Use copy constructor

Use overloaded = operator to assign

© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

```
56
57 if ( integers1 == integers2 )
58     cout << "integers1 and integers2 are equal" << endl;
59
60 // use overloaded subscript operator to create rvalue
61 cout << "\nintegers1[5] is " << integers1[ 5 ];
62
63 // use overloaded subscript operator to create lvalue
64 cout << "\n\nAssigning 1000 to integers1[5]" << endl;
65 integers1[ 5 ] = 1000;
66 cout << "integers1: \n" << integers1;
67
68 // attempt to use out-of-range subscript
69 cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
70 integers1[ 15 ] = 1000; // ERROR: out of range
71 return 0;
72 } // end main
```

Use overloaded == operator to test for equality

Use overloaded [] operator to access individual integers, with range-checking

© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

Size of Array integers1 is 7

Array after initialization:

0	0	0	0
0	0	0	0

Size of Array integers2 is 10

Array after initialization:

0	0	0	0
0	0	0	0
0	0		

Enter 17 integers:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the Arrays contain:

integers1:

1	2	3	4
5	6	7	

integers2:

8	9	10	11
12	13	14	15
16	17		

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

© 2007 Pearson Ed -All rights reserved.

Case Study: Array Class

Size of Array integers3 is 7

Array after initialization:

1	2	3	4
5	6	7	

Assigning integers2 to integers1:

integers1:

8	9	10	11
12	13	14	15
16	17		

integers2:

8	9	10	11
12	13	14	15
16	17		

Evaluating: integers1 == integers2

integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]

integers1:

8	9	10	11
12	1000	14	15
16	17		

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range

© 2007 Pearson Ed -All rights reserved.

Summary

- Covered operator overloading basics.
- Reviewed operator overloading restrictions.
- Explained when to use class member functions and when to use global functions to implement operator overloading.
- Discussed overloading stream insertion and stream extraction operators and did one simple example of overloading.

Summary

- Went through overloading unary and binary operators.
- Looked at operator overloading in an elaborate case study involving an Array class.