Operator Overloading in C++
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
- Case Study: A Date Class
- Standard Library Class string
- explicit Constructors

N: Not applicable
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.)
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**

```cpp
function name operator + for the addition
operator +
```
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
  \[
  \text{object2} = \text{object1.add( object2 )};
  \]
  \[
  \text{vs.}
  \]
  \[
  \text{object2} = \text{object2} + \text{object1};
  \]

© 2007 Pearson Ed - All rights reserved.

Systems Programming: Operator Overloading
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.
### Operators that can be overloaded

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>-</td>
<td>*</td>
<td>/</td>
<td>%</td>
<td>^</td>
<td>&amp;</td>
</tr>
<tr>
<td>~</td>
<td>!</td>
<td>=</td>
<td>&lt;</td>
<td>&gt;</td>
<td>+=</td>
<td>-=</td>
</tr>
<tr>
<td>/=</td>
<td>%=</td>
<td>^=</td>
<td>&amp;=</td>
<td></td>
<td>=</td>
<td>&lt;&lt;=</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>!==</td>
<td>!==</td>
<td>!==</td>
<td>!==</td>
<td>!==</td>
<td>!==</td>
</tr>
<tr>
<td>-</td>
<td>-=</td>
<td>-&gt;</td>
<td>,</td>
<td>- &gt;</td>
<td>[ ]</td>
<td>( )</td>
</tr>
</tbody>
</table>
| new[ ]   | delete[ ]| new      | delete[ ]
Fig. 22.1 Operators that cannot be overloaded.

Operators that cannot be overloaded

.  .  .*  :::  ?:
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.
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.
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.
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.
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`
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`
Overload Stream Insertion and Extraction Operators

```
// Fig. 22.3: PhoneNumber.h
// PhoneNumber class definition
#ifndef PHONENUMBER_H
#define PHONENUMBER_H

#include <iostream>
using std::ostream;
using std::istream;

#include <string>
using std::string;

class PhoneNumber
{
    friend ostream &operator<<( ostream & const PhoneNumber & );
    friend istream &operator>>( istream &, PhoneNumber & );

private:
    string areaCode; // 3-digit area code
    string exchange; // 3-digit exchange
    string line; // 4-digit line
}; // end class PhoneNumber
#endif
```

Notice function prototypes for overloaded operators `>>` and `<<` (must be global, `friend` functions)
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
Overload Stream Insertion and Extraction Operators

```cpp
// overloaded stream extraction operator; cannot be a member function if we would like to invoke it with cin >> somePhoneNumber;
istream &operator>>( istream &input, PhoneNumber &number )
{
    input.ignore(); // skip (
    input >> setw( 3 ) >> number.areaCode; // input area code
    input.ignore( 2 ); // skip ) and space
    input >> setw( 3 ) >> number.exchange; // input exchange
    input.ignore(); // skip dash (-)
    input >> setw( 4 ) >> number.line; // input line
    return input; // enables cin >> a >> b >> c;
} // end function operator>>
```

- `ignore` skips specified number of characters from input (1 by default)
- Input each portion of phone number separately
Systems Programming: Operator Overloading

Overload Stream Insertion and Extraction Operators

// Fig. 22.5: fig22_05.cpp
// Demonstrating class PhoneNumber's overloaded stream insertion
// and stream extraction operators.
#include <iostream>
#include "PhoneNumber.h"

int main()
{
    PhoneNumber phone; // create object phone
    cout << "Enter phone number in the form (123) 456-7890: " << endl;

    cin >> phone; // cin >> phone invokes operator>> by implicitly issuing
    // the global function call operator>>( cin, phone )
    cout << "The phone number entered was: ";

    cout << phone << endl; // cout << phone << endl;
    return 0;
} // end main

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

© 2007 Pearson Ed - All rights reserved.
Enter phone number in the form (123) 456-7890:
(800) 555-1212

The phone number entered was: (800) 555-1212
22.6 Overloading Unary Operators

- Overloading unary operators of a class:
  - Can overload as a non-\texttt{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, \texttt{static} functions only access \texttt{static} data.
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)`
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.
22.7 Overloading Binary Operators

- If a non-\texttt{static} member function, it needs one argument.
  \begin{itemize}
  \item \texttt{class String}
    \begin{verbatim}
    public:
      const String & operator+=( const String & );
    ...
    
    \end{verbatim}
  \item \texttt{y += z} becomes \texttt{y.operator+= ( z )}
  \end{itemize}

- If a global function, it needs two arguments.
  \begin{itemize}
  \item \texttt{const String &operator+=( String & const String & );}
  \item \texttt{y += z} becomes \texttt{operator+= ( y, z )}
  \end{itemize}
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.}
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 `!=`
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.

```cpp
Array newArray( oldArray );
Array newArray = oldArray;  // (both are identical)
```

- `newArray` is a copy of `oldArray`
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!
Case Study: Array Class

```cpp
// Fig. 22.6: Array.h
// Array class for storing arrays of integers.
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
using std::ostream;
using std::istream;

class Array
{
    friend ostream &operator<<( ostream &, const Array & );
    friend istream &operator>>( istream &, Array & );

public:
    Array( int = 10 ); // default constructor
    Array( const Array & ); // copy constructor
    ~Array(); // destructor
    int getSize() const; // return size
    const Array &operator=( const Array & ); // assignment operator
    bool operator==( const Array & ) const; // equality operator
    bool operator!=( const Array & right ) const
    {
        return ! ( *this == right ); // invokes Array::operator==
    } // end function operator!=

    const Array &operator=( const Array & ); // assignment operator
    bool operator==( const Array & ) const; // equality operator

    // inequality operator; returns opposite of == operator
    bool operator!=( const Array &right ) const
    {
        return ! ( *this == right ); // invokes Array::operator==
    } // 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
// subscript operator for non-const objects returns modifiable lvalue
int &operator[]( int );

// subscript operator for const objects returns rvalue
int operator[]( int ) const;

private:
int size; // pointer-based array size
int *ptr; // pointer to first element of pointer-based array
}; // end class Array

Note: An example of pointer data member

Operators for accessing specific elements of Array object
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.
// copy constructor for class Array;
// must receive a reference to prevent infinite recursion
Array::Array( const Array &arrayToCopy )
    : size( arrayToCopy.size )
{
    ptr = new int[size]; // create space for pointer-based array

    for ( int i = 0; i < size; i++ )
        ptr[i] = arrayToCopy.ptr[i]; // copy into object
} // end Array copy constructor

// destructor for class Array
Array::~Array()
{
    delete[] ptr; // release pointer-based array space
} // end destructor

// return number of elements of Array
int Array::getSize() const
{
    return size; // number of elements in Array
} // end function getSize

We must declare a new integer array so the objects do not point to the same memory
```cpp
// overloaded assignment operator;
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
    if ( &right != this ) // avoid self-assignment
    {
        // for Arrays of different sizes, deallocate original
        // left-side array, then allocate new left-side array
        if ( size != right.size )
        {
            delete [] ptr; // release space
            size = right.size; // resize this object
            ptr = new int[size]; // create space for array copy
        } // end inner if

        for ( int i = 0; i < size; i++ )
        {
            ptr[i] = right.ptr[i]; // copy array into object
        } // end outer if
    }

    return *this; // enables x = y = z, for example
} // end function operator=
```

Want to avoid self assignment

This would be dangerous if `this` is the same `Array` as `right`
// determine if two Arrays are equal and return true, otherwise return false
bool Array::operator==( const Array &right ) const
{
    if ( size != right.size )
        return false; // arrays of different number of elements
    for ( int i = 0; i < size; i++ )
        if ( ptr[ i ] != right.ptr[ i ] )
            return false; // Array contents are not equal
    return true; // Arrays are equal
}

int &Array::operator[]( int subscript )
{
    // check for subscript out-of-range error
    if ( subscript < 0 || subscript >= size )
    {
        cerr << "\nError: Subscript " << subscript << " out of range" << endl;
        exit( 1 ); // terminate program: subscript out of range
    } // end if
    return ptr[ subscript ]; // reference return
} // end function operator[]
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
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 <<
```

Systems Programming: Operator Overloading
```cpp
// Fig. 22.8: fig22_08.cpp
// Array class test program
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "Array.h"

int main()
{
    Array integers1(7); // seven-element Array
    Array integers2; // 10-element Array by default

    // print integers1 size and contents
    cout << "Size of Array integers1 is " << integers1.getSize() << \
         "Array after initialization:
    " << integers1;

    // print integers2 size and contents
    cout << "\nSize of Array integers2 is " << integers2.getSize() << \
         "Array after initialization:\n    " << integers2;

    // input and print integers1 and integers2
    cout << "\nEnter 17 integers: " << endl;
    cin >> integers1 >> integers2;
```

Retrieve number of elements in `Array`

Use overloaded `>>` operator to input
cout << "After input, the Arrays contain:
" << integers1 << " integers2:"
// use overloaded inequality (!=) operator
cout << "Evaluating: integers1 != integers2" << endl;
if ( integers1 != integers2 )
    cout << "integers1 and integers2 are not equal" << endl;

// create Array integers3 using integers1 as an
// initializer; print size and contents
Array integers3( integers1 ); // invokes copy constructor

cout << "Size of Array integers3 is " << integers3.getSize()
    << " Array after initialization:
" << integers3;

// use overloaded assignment (=) operator
cout << "Assigning integers2 to integers1: " << endl;
integers1 = integers2; // note target Array is smaller

cout << "integers1:\n" << integers1 << " integers2:"
    << integers2;

// use overloaded equality (==) operator
cout << "Evaluating: integers1 == integers2" << endl;
Case Study: Array Class

```cpp
if ( integers1 == integers2 )
    cout << "integers1 and integers2 are equal" << endl;

// use overloaded subscript operator to create rvalue
cout << "\ni\ntegers1[5] is " << integers1[5];

// use overloaded subscript operator to create lvalue
cout << "\nAssigning 1000 to integers1[5]" << endl;
integers1[5] = 1000;
cout << "\nAssigning 1000 to integers1[5]" << endl;
integers1[5] = 1000; // ERROR: out of range

return 0;
```
**Case Study: Array Class**

<table>
<thead>
<tr>
<th>Size of Array</th>
<th>integers1 is 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array after initialization:</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>Array after initialization:</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size of Array</th>
<th>integers2 is 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array after initialization:</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>Array after initialization:</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>Array after initialization:</td>
<td>0 0</td>
</tr>
</tbody>
</table>

**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:**

<table>
<thead>
<tr>
<th>integers1:</th>
<th>1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 6 7</td>
<td></td>
</tr>
<tr>
<td>integers2:</td>
<td>8 9 10 11</td>
</tr>
<tr>
<td>12 13 14 15</td>
<td></td>
</tr>
<tr>
<td>16 17</td>
<td></td>
</tr>
</tbody>
</table>

**Evaluating: integers1 != integers2**
integers1 and integers2 are not equal
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
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.
Went through overloading unary and binary operators.

Looked at operator overloading in an elaborate case study involving an Array class.