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
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, pointers)
An operator is overloaded by writing:
- a non-\texttt{static} member function definition
  or
- a global function definition
where
the function name becomes the keyword \texttt{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

```
fuction name operator+ for the addition
operator +
```
Operator Overloading

- To use an 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
  \[
  \text{object2} = \text{object1}.\text{add( object2 );}
  \]
  vs.

  \[
  \text{object2} = \text{object2} + \text{object1};
  \]
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.**
### Fig. 22.1 Operators that can be overloaded

<table>
<thead>
<tr>
<th>Operators that can be overloaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
</tr>
<tr>
<td>~</td>
</tr>
<tr>
<td>/=</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
</tr>
<tr>
<td>-=</td>
</tr>
</tbody>
</table>

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 made 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

© 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 (pointers and strings).
  - 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
// 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

```cpp
// Fig. 22.4: PhoneNumber.cpp
// Overloaded stream insertion and stream extraction operators
// for class PhoneNumber.
#include <iomanip>
using std::setw;

#include "PhoneNumber.h"

// overloaded stream insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<( ostream &output, const PhoneNumber &number )
{
    output << "(" << number.areaCode << " " << number.exchange << "-" << number.line;
    return output; // enables cout << a << b << c;
} // 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

© 2007 Pearson Ed - All rights reserved.
// Fig. 22.5: fig22_05.cpp
// Demonstrating class PhoneNumber's overloaded stream insertion
// and stream extraction operators.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "PhoneNumber.h"

int main()
{
    PhoneNumber phone; // create object phone
    cout << "Enter phone number in the form (123) 456-7890: " << endl;
    cin >> phone;
    cout << "The phone number entered was: ";
    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-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.
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-`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 )`
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.}
Case Study: Implement an **Array** class with:

1. Range checking
2. Array assignment ( = )
3. Arrays that know their own size.
4. Outputting/inputting entire arrays with << and >>
5. Array comparisons with == and !=
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( oldArray );
or
Array newArray = oldArray; (both are identical)
```

- `newArray` is a copy of `oldArray`. 
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 &const 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
!

// inequality operator; returns opposite of == operator
bool operator!=( const Array &right ) const
{
    return ! ( *this == right ); // invokes Array::operator ==
}
}; // end class Array
```

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**

Operators for accessing specific elements of Array object

Note: An example of pointer data member
// Fig 22.7: Array.cpp
// Member-function definitions for class Array

#include <iostream>

using std::cerr;
using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setw;

#include <cstdlib> // exit function prototype

using std::exit;

#include "Array.h" // Array class definition

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
    size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
    ptr = new int[ size ]; // create space for pointer-based array
    for ( int i = 0; i < size; i++ )
        ptr[ i ] = 0; // set pointer-based array element
} // end Array default constructor
Case Study: Array Class

```cpp
// 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()
{
    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.
systems programming: operator overloading

Case Study: Array Class

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

```cpp
// 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
} // end function operator==

// overloaded subscript operator for non-const Arrays;
// reference return creates a modifiable lvalue
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[]
```

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

© 2007 Pearson Ed - All rights reserved.
// overloaded subscript operator for const Arrays
// const reference return creates an rvalue
int Array::operator[]( int subscript ) const
{
    // check for subscript out-of-range error
    if ( subscript < 0 || subscript >= size )
    {
        cerr << "Error: Subscript " << subscript
             << " out of range" << endl;
        exit( 1 ); // terminate program; subscript out of range
    } // end if

    return ptr[ subscript ]; // returns copy of this element
} // end function operator[]

// overloaded input operator for class Array;
// inputs values for entire Array
istream &operator>>( istream &input, Array &a )
{
    for ( int i = 0; i < a.size; i++ )
        input >> a.ptr[ i ];

    return input; // enables cin >> x >> y;
} // end function
Case Study: Array Class

```cpp
125 // overloaded output operator for class Array
126 ostream &operator<<( ostream &output, const Array &a )
127 {
128     int i;
129
130     // output private ptr-based array
131     for ( i = 0; i < a.size; i++ )
132     {
133         output << setw( 12 ) << a.ptr[ i ];
134     }
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<=
// 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 << "Size of Array integers2 is 
" << integers2.getSize() << "Array after initialization:
" << integers2;

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

Retrieve number of elements in Array
Use overloaded >> operator to input

© 2007 Pearson Ed - All rights reserved.
cout << "\nAfter input, the Arrays contain: \n" << integers1: \n" << integers2: \n" << integers2;

// use overloaded inequality (!=) operator
cout << "\nEvaluating: 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 << "\nSize of Array integers3 is "
  << integers3.getSize()
  << "\nArray after initialization: \n" << integers3;

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

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

// use overloaded equality (==) operator
cout << "\nEvaluating: integers1 == integers2" << endl;
Use overloaded == operator to test for equality

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

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

// use overloaded subscript operator to create rvalue
cout << "\n\nintegers1[5] is " << integers1[5];

// use overloaded subscript operator to create lvalue
cout << "\n\nAssigning 1000 to integers1[5]" << endl;
integers1[5] = 1000;
cout << "\n\nAssigning 1000 to integers1[5]" << endl;
integers1 = 1000;
cout << "\n\nAssigning 1000 to integers1[5]" << endl;
integers1[15] = 1000; // ERROR: out of range
return 0;
} // end main
```
Case Study: Array Class

Size of Array integers1 is 7
Array after initialization:

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
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.
Summary

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