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
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:

\[
\ll
\]
- Stream insertion, bitwise left-shift

\[
+
\]
- Performs arithmetic on multiple items (integers, floats, pointers)
An operator is overloaded by writing:

- a **non-static** member function definition

or

- a global function definition *(non-member function definition in 7th edition of text)*

where

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

```c
function name operator +
```

to overload the addition operator `+`
To use an operator on a class object:
- The operator must be overloaded for that class.

Three Exceptions: {overloading not required}
- Assignment operator (=)
  • Performs “memberwise” assignment between objects
  • Dangerous for classes with pointer members!!
- Address operator (&)
  • Returns a pointer to the object.
- Comma operator (,)
  • Evaluates the expression to its left then the expression to its right.
  • Returns the value of the expression to its right.

Overloading provides concise notation
\[
\text{object2} = \text{object1}.\text{add}( \text{object2} );
\]
\[
\text{vs.}
\]
\[
\text{object2} = \text{object2} + \text{object1} ;
\]
Restrictions on Operator Overloading

- **Overloading cannot change:**
  - The precedence of the 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.**

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>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>-</td>
<td>*</td>
<td>/</td>
<td>%</td>
<td>^</td>
</tr>
<tr>
<td>~</td>
<td>!</td>
<td>=</td>
<td>&lt;</td>
<td>&gt;</td>
<td>+=</td>
</tr>
<tr>
<td>/=</td>
<td>%=</td>
<td>^=</td>
<td>&amp;=</td>
<td></td>
<td>=</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>==</td>
<td>!=</td>
<td>&lt;=</td>
<td>&gt;=</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>--</td>
<td>-*</td>
<td>,</td>
<td>-&gt;</td>
<td>[]</td>
<td>()</td>
</tr>
<tr>
<td>new[]</td>
<td>delete[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2007 Pearson Ed - All rights reserved.
Fig. 19.2 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 the same class as operator function.
  - Use **this** keyword to implicitly get left operand argument.
  - Operators ( ), [ ], -> or any of the assignment operators **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 an object of a different class than the 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 a member function.

- For the other way, you need a global overloaded function.
  
  long int + HugeIntClass
19.5 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
// 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;
// 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
// 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>>
// 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; // 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 invokes operator<< by implicitly issuing
    // the global function call operator<<( cout, phone )
    return 0;
} // end main
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
19.6 Overloading Unary Operators

- Overloading unary operators of a class:
  - Can overload as a non-static member function with no arguments.
  OR
  - 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 an 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;`  
  `}; // end class String`  

- If a global function, it needs one argument:
  - `bool operator!( const String & )`
  - `!s` becomes `operator!(s)`
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.
  - ```cpp
class String {
  public:
    const String & operator+=( const String & );
  ...  
};
```
  - `y += z` becomes ```cpp>y . operator+= ( z )```

- If a global function, it needs two arguments.
  - ```cpp>const String & operator+=( String &, const String & );
  - y += z becomes ```cpp>operator+= ( y, z )```
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 \textit{side-effect} this object that is called-by-reference!)
19.10 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 object of the same type.
    ```
    Array newArray( oldArray );
    Array newArray = oldArray;  // both are identical
    ```
    - `newArray` is a copy of `oldArray`. 
Case Study: Array Class

- Prototype for class `Array`

```
Array( const Array & );  //copy constructor
```

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

Operators for accessing specific elements of Array object

Operators for accessing specific elements of Array object

Operators for accessing specific elements of Array object

Operators for accessing specific elements of Array object

Note: An example of pointer data member
Case Study: Array Class

// 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

Note: Standard method for using pointer to access an array of objects.
We must declare a new integer array so the objects do not point to the same memory.
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 )
        {
            // release space
            delete [] ptr;
            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=
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
}

// 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)
Case Study: **Array Class**

100
101 // overloaded subscript operator for const Arrays
102 // const reference return creates an rvalue
103 int 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
// overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
    int i;

    // output private ptr-based array
    for ( i = 0; i < a.size; i++ )
    {
        output << setw( 12 ) << a.ptr[ i ];

        if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
            output << endl;
    } // end for

    if ( i % 4 != 0 ) // end last line of output
        output << endl;

    return output; // enables cout << x << y;
} // 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:\n" << 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

Case Study: Array Class

Systems Programming
cout << "\nAfter input, the Arrays contain:\n" << "integers1:\n" << integers1 << "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:" << endl;
integers1 = integers2; // note target Array is smaller
cout << "integers1:\n" << integers1 << "integers2:\n" << integers2;

// use overloaded equality (==) operator
cout << "\nEvaluating: integers1 == integers2" << endl;
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 << "integers1:\n" << integers1;

// attempt to use out-of-range subscript
cout << "\n\nAttempt to assign 1000 to integers1[15]" << 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  0  

Size of Array integers2 is 10 
Array after initialization:   
0  0  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
Size of Array integers3 is 7
Array after initialization:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assigning integers2 to integers1:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

integers2:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]

integers1:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>1000</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range
- 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.
  - Several good C++ concepts in this example including the copy constructor!!