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
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-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

```
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: {can be overloaded - not just for one class}
  - 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

```
object2 = object1.add( object2 );
```

vs.

```
object2 = object2 + object1;
```
Restrictions on Operator Overloading

- Overloading cannot change:
  - The precedence of the operator (order of evaluation)
    - Use parentheses to force order of operators.
  - Associativity of the operator (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>&lt;=</td>
<td>=&gt;</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>--</td>
<td>-*</td>
<td>,</td>
<td>-&gt;</td>
<td>[]</td>
<td>()</td>
<td>new</td>
</tr>
<tr>
<td>delete[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 22.1 Operators that can be overloaded.
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** (i.e., non-member function):

  - 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 Binary 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;

#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)
// 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<<

// Fig. 22.4: PhoneNumber.cpp
// Overloaded stream insertion and stream extraction operators
// for class PhoneNumber.
1 // include <iomanip>
2 using std::setw;
3
4 // include "PhoneNumber.h"
5
6 // overloaded stream insertion operator; cannot be
7 // a member function if we would like to invoke it with
8 // cout << somePhoneNumber;
9 ostream &operator<<( ostream &output, const PhoneNumber &number )
10 {
11     output << "(" << number.areaCode << ") "
12         << number.exchange << ":" << number.line;
13     return output; // enables cout << a << b << c;
14 } // end function operator<<

Displays formatted phone number

 Allows cout << phone; to be interpreted as: operator<< (cout, phone);
// 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 invokes operator>> by implicitly issuing
    // the global function call operator>>( cin, phone )
    cin >> phone;

    cout << "The phone number entered was: ";

    // cout << phone invokes operator<< by implicitly issuing
    // the global function call operator<<( cout, phone )
    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
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

\[ s.\text{operator!()} \]

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

- class String

\[
\begin{array}{l}
\text{public:} \\
\quad \text{bool operator!()} \text{ const;} \\
\end{array}
\]

};  // end class String

- If a global function, it needs one argument:

\[
\begin{array}{l}
\text{bool operator!}(\text{const String} \&) \\
\text{!s becomes operator!}(s) \\
\end{array}
\]
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 )`
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!)
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: Array Class

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.
  ```
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!**
// 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

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

```cpp
// 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

© 2007 Pearson Ed - All rights reserved.
// 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.
// 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.
// 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=
// 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[]
// 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
// 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<<
Case Study: **Array Class**

```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()
         << \n         << "Array after initialization:\n" << integers1;

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

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

- Retrieve number of elements in `Array`
- Use overloaded `>>` operator to input
cout << "\nAfter input, the Arrays contain:\n" << integers1 << 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() << integers3;

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

cout << integers1 << integers2;

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

Use overloaded `==` operator to test for equality

Use overloaded `[]` operator to access individual integers, with range-checking
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
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
Review of Operator Overloading

- 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.
Review of Operator 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!!