Chapter 18 - C++ Operator Overloading

Outline
18.1 Introduction
18.2 Fundamentals of Operator Overloading
18.3 Restrictions on Operator Overloading
18.4 Operator Functions as Class Members vs. as friend Functions
18.5 Overloading Stream-Insertion and Stream-Extraction Operators
18.6 Overloading Unary Operators
18.7 Overloading Binary Operators
18.8 Case Study: An Array Class
18.9 Converting between Types
18.10 Overloading ++ and --
Objectives

• In this chapter, you will learn:
  – To understand how to redefine (overload) operators to work with new types.
  – To understand how to convert objects from one class to another class.
  – To learn when to, and when not to, overload operators.
  – To study several interesting classes that use overloaded operators.
  – To create an Array class.
18.1 Introduction

• Chapter 16 and 17
  – ADT’s and classes
  – Function-call notation is cumbersome for certain kinds of classes, especially mathematical classes

• In this chapter
  – We use C++’s built-in operators to work with class objects
18.1 Introduction

• Operator overloading
  – Use traditional operators with user-defined objects
  – Straightforward and natural way to extend C++
  – Requires great care
    • When overloading is misused, programs become difficult to understand
18.2 Fundamentals of Operator Overloading

• Use operator overloading to improve readability
  – Avoid excessive or inconsistent usage

• Format
  – Write function definition as normal
  – Function name is keyword `operator` followed by the symbol for the operator being overloaded.
  – `operator+` would be used to overload the addition operator (+)
18.2 Fundamentals of Operator Overloading

• Assignment operator (\(=\))
  – may be used with every class without explicit overloading
  – *memberwise assignment*
  – Same is true for the address operator (\&)
18.3 Restrictions on Operator Overloading

Most of C++’s operators can be overloaded.
## 18.3 Restrictions on Operator Overloading

### Fig. 18.2 Operators that cannot be overloaded.

<table>
<thead>
<tr>
<th>Operators that cannot be overloaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
</tr>
</tbody>
</table>

18.3 Restrictions on Operator Overloading

• Arity (number of operands) cannot be changed
  – Unary operators remain unary, and binary operators remain binary
  – Operators &, *, +, and − each have unary and binary versions
    • Unary and binary versions can be overloaded separately
18.3 Restrictions on Operator Overloading

- No new operators can be created
  - Use only existing operators
- Built-in types
  - Cannot overload operators
  - You cannot change how two integers are added
18.4 Operator Functions as Class Members vs. as friend Functions

• Operator functions
  – Can be member or non-member functions

• Overloading the assignment operators
  – i.e: (), [], ->, =
  – Operator must be a member function
18.4 Operator Functions as Class Members vs. as friend Functions

• Operator functions as member functions
  – Leftmost operand must be an object (or reference to an object) of the class
  – If left operand of a different type, operator function must be a non-member function
  – A non-member operator function must be a friend if private or protected members of that class are accessed directly
Non-member overloaded operator functions
  – Enable the operator to be commutative

```cpp
HugeInteger bigInteger1;
long int number;
bigInteger1 = number + bigInteger1;
```

or

```cpp
bigInteger1 = biginteger1 + number;
```
18.5 Overloading Stream-Insertion and Stream-Extraction Operators

- Overloaded `<<` and `>>` operators
  - Must have left operand of types `ostream &`, `istream &` respectively
  - It must be a non-member function (left operand not an object of the class)
  - It must be a `friend` function if it accesses private data members
// Fig. 18.3: fig18_03.cpp
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;

#include <iomanip>

using std::setw;

class PhoneNumber {

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

private:
char areaCode[ 4 ]; // 3-digit area code and null
char exchange[ 4 ]; // 3-digit exchange and null
char line[ 5 ]; // 4-digit line and null
}; // end class PhoneNumber
// 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 &num )
{
    output << "(" << num.areaCode << "\n" << num.exchange << "-" << num.line;
    return output; // enables cout << a << b << c;
} // end operator<< function

istream &operator>>( istream &input, PhoneNumber &num )
{
    input.ignore(); // skip (
    input >> setw(4) >> num.areaCode; // input area code
    input.ignore(2); // skip ) and space
    input >> setw(4) >> num.exchange; // input exchange
    input.ignore(); // skip dash (-)
    input >> setw(5) >> num.line; // input line
    return input; // enables cin >> a >> b >> c;
} // end operator>> function

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

// cin >> phone invokes operator>> function by
// issuing the call operator>>( cin, phone ).
cin >> phone;

// cout << phone invokes operator<< function by
// issuing the call operator<<( cout, phone ).
cout << "The phone number entered was: " << phone << endl;
return 0;

} // end function main

Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
18.6 Overloading Unary Operators

- Overloading unary operators
  - Avoid friend functions and friend classes unless absolutely necessary.
  - Use of friends violates the encapsulation of a class.
  - As a member function:

```cpp
class String {
  public:
    bool operator!() const;
    ...  
};
```
18.7 Overloading Binary Operators

• Overloaded binary operators
  – Non-static member function, one argument
  – Non-member function, two arguments

    class String {
    public:
      const String &operator+=( const String & );
      ...
    } // end class String

    y += z;
    equivalent to
    y.operator+=( z );
18.7 Overloading Binary Operators

• Example

```c++
class String {
    friend const String &operator+=( String &, const String & );
    ...
}; // end class String

y += z;
equivalent to
operator+=( y, z );
```
18.8 Case Study: An Array class

- Implement an Array class with
  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with << and >>
  - Array comparisons with == and !=
// Fig. 18.4: array1.h
// Simple class Array (for integers)

#ifndef ARRAY1_H
#define ARRAY1_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 & ); // assign arrays
    bool operator==( const Array & ) const; // compare equal

    // Determine if two arrays are not equal and
    // return true, otherwise return false (uses operator==).
    bool operator!=( const Array &right ) const
    {
        return ! ( *this == right );
    }

};

#endif
27  int &operator[]( int );              // subscript operator
28  const int &operator[]( int ) const;  // subscript operator
29  static int getArrayCount();        // Return count of
30                                                      // arrays instantiated.
31  
32  private:
33      int size;  // size of the array
34      int *ptr;  // pointer to first element of array
35      static int arrayCount;  // # of Arrays instantiated
36  
37     #endif
38  
39     // Fig 18.4: array1.cpp
40     // Member function definitions for class Array
41 #include <iostream>
42
43    using std::cout;
44    using std::cin;
45    using std::endl;
46
47     #include <iomanip>
48
49    using std::setw;
50
51     #include <cstdlib>
52     #include <cassert>
53     #include "array1.h"
int Array::arrayCount = 0; // no objects yet

// Default constructor for class Array (default size 10)
Array::Array(int arraySize)
{
    size = (arraySize > 0 ? arraySize : 10);
    ptr = new int[size]; // create space for array
    assert(ptr != 0); // terminate if memory not allocated
    ++arrayCount; // count one more object

    for (int i = 0; i < size; i++)
        ptr[i] = 0; // initialize array
}
 // end Array constructor

// Copy constructor for class Array
// must receive a reference to prevent infinite recursion
Array::Array(const Array &init) : size(init.size)
{
    ptr = new int[size]; // create space for array
    assert(ptr != 0); // terminate if memory not allocated
    ++arrayCount; // count one more object

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

// Destructor for class Array
Array::~Array()
{
    delete [] ptr; // reclaim space for array
    --arrayCount; // one fewer object
} // end Array destructor

// Get the size of the array
int Array::getSize() const { return size; }

// Overloaded assignment operator
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
    if ( &right != this ) { // check for self-assignment
        // for arrays of different sizes, deallocate original
        // left side array, then allocate new left side array.
        if ( size != right.size ) {
            delete [] ptr; // reclaim space
            size = right.size; // resize this object
            ptr = new int[ size ]; // create space for array copy
            assert( ptr != 0 ); // terminate if not allocated
        } // end if

        for ( int i = 0; i < size; i++ )
            ptr[ i ] = right.ptr[ i ]; // copy array into object
    } // end if
return *this;   // enables x = y = z; 
} // end operator= function 

// 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 sizes

    for ( int i = 0; i < size; i++ )
        if ( ptr[ i ] != right.ptr[ i ] )
            return false;       // arrays are not equal

    return true;      // arrays are equal
} // end operator== function 

// Overloaded subscript operator for non-const Arrays 
// reference return creates an lvalue 
int &Array::operator[]( int subscript ) 
{
    // check for subscript out of range error
    assert( 0 <= subscript && subscript < size );

    return ptr[ subscript ]; // reference return
} // end operator[] function 

137 // Overloaded subscript operator for const Arrays
138 // const reference return creates an rvalue
139 const int &Array::operator[]( int subscript ) const
140 {
141   // check for subscript out of range error
142   assert( 0 <= subscript && subscript < size );
143
144   return ptr[ subscript ]; // const reference return
145 } // end operator[] function
146
147 // Return the number of Array objects instantiated
148 // static functions cannot be const
149 int Array::getArrayCount() { return arrayCount; }
150
151 // Overloaded input operator for class Array;
152 // inputs values for entire array.
153 istream &operator>>( istream &input, Array &a )
154 {
155   for ( int i = 0; i < a.size; i++ )
156     input >> a.ptr[ i ];
157
158   return input; // enables cin >> x >> y;
159 } // end operator>> function
160
// Overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
    int i;

    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 )
        output << endl;

    return output; // enables cout << x << y;
} // end operator<< function

// Fig. 18.4: fig18_04.cpp
// Driver for simple class Array
#include <iostream>

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

#include "array1.h"
`int main() {
    // no objects yet
    cout << "# of arrays instantiated = " << Array::getArrayCount() << '\n';

    // create two arrays and print Array count
    Array integers1(7), integers2;
    cout << "# of arrays instantiated = " << Array::getArrayCount() << '\n' << '\n';

    // print integers1 size and contents
    cout << "Size of array integers1 is " << integers1.getSize() << '\n' << integers1 << '\n';

    // print integers2 size and contents
    cout << "Size of array integers2 is " << integers2.getSize() << '\n' << integers2 << '\n';
}`
```cpp
// input and print integers1 and integers2
cout << "Input 17 integers:\n";
cin >> integers1 >> integers2;
cout << "After input, the arrays contain:\n" << integers1 << integers2 << '\n';

// use overloaded inequality (!=) operator
cout << "Evaluating: integers1 != integers2\n";
if (integers1 != integers2 )
    cout << "They are not equal\n";

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

cout << "\nSize of array integers3 is " << integers3.getSize() << '\n';

// use overloaded assignment (=) operator
cout << "Assigning integers2 to integers1:\n";
integers1 = integers2;
cout << integers1 << integers2 << '\n';
```

238     // use overloaded equality (==) operator
239     cout << "Evaluating: integers1 == integers2\n";
240     if ( integers1 == integers2 )
241         cout << "They are equal\n\n";
242
243     // use overloaded subscript operator to create rvalue
244     cout << "integers1[5] is " << integers1[ 5 ] << '\n';
245
246     // use overloaded subscript operator to create lvalue
247     cout << "Assigning 1000 to integers1[5]\n";
248         integers1[ 5 ] = 1000;
249     cout << "integers1:\n" << integers1 << '\n';
250
251     // attempt to use out of range subscript
252     cout << "Attempt to assign 1000 to integers1[15]" << endl;
253         integers1[ 15 ] = 1000; // ERROR: out of range
254
255     return 0;
256 } // end function main
# of arrays instantiated = 0
# of arrays instantiated = 2

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
```

Input 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
They are not equal

Size of array integers3 is 7
Array after initialization:

```
  1   2   3   4
  5   6   7
```
Assigning `integers2` to `integers1`:

```
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`
They 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]`
Assertion failed: 0 <= subscript && subscript < size, file Array1.cpp, line 95 abnormal program termination
18.9 Converting between Types

• Cast operator
  – Convert objects into built-in types or other objects
  – Conversion operator must be a non-static member function.
  – Cannot be a friend function
  – Do not specify return type

For user-defined class A
  A::operator char *() const;
  A::operator int() const;
  A::operator otherClass() const;

  – When compiler sees (char *) s it calls
    s.operator char*()
18.9 Converting between Types

- The compiler can call these functions to create temporary objects.
  - If \( s \) is not of type \( \text{char \ast} \)

Calls \( \text{A::operator char \ast() const; for cout \<\< s; } \)
18.10 Overloading ++ and --

- **Pre/post-incrementing/decrementing operators**
  - Can be overloaded
  - How does the compiler distinguish between the two?
  - Prefix versions overloaded same as any other prefix unary operator would be. i.e. `d1.operator++();` for `++d1;`

- **Postfix versions**
  - When compiler sees postincrementing expression, such as `d1++;`
    - Generates the member-function call
      ```
      d1.operator++( 0 );
      ```
  - Prototype:
    ```
    Date::operator++( int );
    ```