Chapter 18 - Operator Overloading

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 (II)

- Operator overloading
  - Use traditional operators with user-defined objects
  - Straightforward and natural way to extend C++
  - Requires great care
    - When overloading misused, program 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 (II)

- 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

-

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

18.3 Restrictions on Operator Overloading (II)

- Arity (number of operands) cannot be changed
  - Urnary operators remain urnary, 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 (III)

- 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 (II)

- 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

18.4 Operator Functions as Class Members vs. as `friend` Functions (III)

- Non-member overloaded operator functions
  - Enable the operator to be commutative

```
HugeInteger bigInteger;
 int integer;
 bigInteger = integer + bigInteger;
```

   or

```
bigInteger = biginteger + integer;
```

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

```cpp
 1  // Fig. 18.3: fig18_03.cpp
 2  // Overloading the stream-insertion and
 3  // stream-extraction operators.
 4  #include <iostream>
 5
 6  using std::cout;
 7  using std::cin;
 8  using std::endl;
 9  using std::ostream;
10  using std::istream;
11
12  #include <iomanip>
13
14  using std::setw;
15
16  class PhoneNumber {
17     friend ostream &operator<<( ostream&, const PhoneNumber & );
18     friend istream &operator>>( istream&, PhoneNumber & );
19
20  private:
21     char areaCode[ 4 ];  // 3-digit area code and null
22     char exchange[ 4 ];  // 3-digit exchange and null
23     char line[ 5 ];      // 4-digit line and null
24  };
25
26  // Overloaded stream-insertion operator (cannot be
27  // a member function if we would like to invoke it with
28  // cout << somePhoneNumber;).
29  ostream &operator<<( ostream &output, const PhoneNumber &num )
30  {
```

```cpp
31      output << "(" << num.areaCode << ") "
32             << num.exchange << "-" << num.line;
33      return output;        // enables cout << a << b << c;
34  }
35
36  istream &operator>>( istream &input, PhoneNumber &num )
37  {
38      input.ignore();                       // skip (
39      input >> setw( 4 ) >> num.areaCode;  // input area code
40      input.ignore( 2 );                    // skip ) and space
41      input >> setw( 4 ) >> num.exchange;  // input exchange
42      input.ignore();                       // skip dash (-)
43      input >> setw( 5 ) >> num.line;      // input line
44      return input;        // enables cin >> a >> b >> c;
45  }
46
47  int main()
48  {
49      PhoneNumber phone; // create object phone
50
51      cout << "Enter phone number in the form (123) 456-7890:\n";
52
53      // cin >> phone invokes operator>> function by
54      // issuing the call operator>>( cin, phone ).
55      cin >> phone;
56
57      // cout << phone invokes operator<< function by
58      // issuing the call operator<<( cout, phone ).
59      cout << "The phone number entered was: " << phone << endl;
60      return 0;
61  }
```

```
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 **friend**s violates the encapsulation of a class.
  - As a member function:

```
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 & );
   ...
};
```

```
   y += z;
```
equivalent to
```
y.operator+=( z );
```

18.7 Overloading Binary Operators (II)

- Example

```
class String {
    friend const String &operator+=( String &,
                                      const 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 **!=**

```
1  // Fig. 18.4: array1.h
2  // Simple class Array (for integers)
3  #ifndef ARRAY1_H
4  #define ARRAY1_H
5
6  #include <iostream>
7
8  using std::ostream;
9  using std::istream;
10
11 class Array {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14 public:
15    Array( int = 10 );                    // default constructor
16    Array( const Array & );               // copy constructor
17    ~Array();                             // destructor
18    int getSize() const;                  // return size
19    const Array &operator=( const Array & ); // assign arrays
20    bool operator==( const Array & ) const;  // compare equal
21
22    // Determine if two arrays are not equal and
23    // return true, otherwise return false (uses operator==).
24    bool operator!=( const Array &right ) const
25       { return ! ( *this == right ); }
26
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 private:
32    int size; // size of the array
33    int *ptr; // pointer to first element of array
34    static int arrayCount;  // # of Arrays instantiated
```

```
35  };
36
37  #endif
38  // Fig 18.4: array1.cpp
39  // Member function definitions for class Array
40  #include <iostream>
41
42  using std::cout;
43  using std::cin;
44  using std::endl;
45
46  #include <iomanip>
47
48  using std::setw;
49
50  #include <cstdlib>
51  #include <cassert>
52  #include "array1.h"
53
54  // Initialize static data member at file scope
55  int Array::arrayCount = 0;   // no objects yet
56
57  // Default constructor for class Array (default size 10)
58  Array::Array( int arraySize )
59  {
60      size = ( arraySize > 0 ? arraySize : 10 );
61      ptr = new int[ size ]; // create space for array
62      assert( ptr != 0 );    // terminate if memory not allocated
63      ++arrayCount;          // count one more object
64
65      for ( int i = 0; i < size; i++ )
66          ptr[ i ] = 0;              // initialize array
```

```
67  }
68
69  // Copy constructor for class Array
70  // must receive a reference to prevent infinite recursion
71  Array::Array( const Array &init ) : size( init.size )
72  {
73     ptr = new int[ size ]; // create space for array
74     assert( ptr != 0 );    // terminate if memory not allocated
75     ++arrayCount;          // count one more object
76
77     for ( int i = 0; i < size; i++ )
78        ptr[ i ] = init.ptr[ i ];  // copy init into object
79  }
80
81  // Destructor for class Array
82  Array::~Array()
83  {
84     delete [] ptr;         // reclaim space for array
85     --arrayCount;          // one fewer object
86  }
87
88  // Get the size of the array
89  int Array::getSize() const { return size; }
90
91  // Overloaded assignment operator
92  // const return avoids: ( a1 = a2 ) = a3
93  const Array &Array::operator=( const Array &right )
94  {
95     if ( &right != this ) {  // check for self-assignment
96
97        // for arrays of different sizes, deallocate original
98        // left side array, then allocate new left side array.
99        if ( size != right.size ) {
100           delete [] ptr;        // reclaim space
```
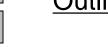
```
101            size = right.size;      // resize this object
102            ptr = new int[ size ]; // create space for array copy
103            assert( ptr != 0 );     // terminate if not allocated
104        }
105
106        for ( int i = 0; i < size; i++ )
107            ptr[ i ] = right.ptr[ i ];   // copy array into object
108    }
109
110    return *this;    // enables x = y = z;
111 }
112
113 // Determine if two arrays are equal and
114 // return true, otherwise return false.
115 bool Array::operator==( const Array &right ) const
116 {
117    if ( size != right.size )
118        return false;     // arrays of different sizes
119
120    for ( int i = 0; i < size; i++ )
121        if ( ptr[ i ] != right.ptr[ i ] )
122            return false; // arrays are not equal
123
124    return true;          // arrays are equal
125 }
126
127 // Overloaded subscript operator for non-const Arrays
128 // reference return creates an lvalue
129 int &Array::operator[]( int subscript )
130 {
131    // check for subscript out of range error
132    assert( 0 <= subscript && subscript < size );
```
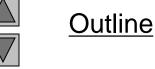
```
133
134     return ptr[ subscript ]; // reference return
135 }
136
137 // Overloaded subscript operator for const Arrays
138 // const reference return creates an rvalue
139 const int &Array::operator[]( int subscript ) const
140 {
141    // check for subscript out of range error
142    assert( 0 <= subscript && subscript < size );
143
144    return ptr[ subscript ]; // const reference return
145 }
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 }
160
161 // Overloaded output operator for class Array
162 ostream &operator<<( ostream &output, const Array &a )
163 {
```

```
164    int i;
165
166    for ( i = 0; i < a.size; i++ ) {
167       output << setw( 12 ) << a.ptr[ i ];
168
169       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
170          output << endl;
171    }
172
173    if ( i % 4 != 0 )
174       output << endl;
175
176    return output;    // enables cout << x << y;
177 }
178 // Fig. 18.4: fig18_04.cpp
179 // Driver for simple class Array
180 #include <iostream>
181
182 using std::cout;
183 using std::cin;
184 using std::endl;
185
186 #include "array1.h"
187
188 int main()
189 {
190    // no objects yet
191    cout << "# of arrays instantiated = "
192         << Array::getArrayCount() << '\n';
193
```

```
194    // create two arrays and print Array count
195    Array integers1( 7 ), integers2;
196    cout << "# of arrays instantiated = "
197         << Array::getArrayCount() << "\n\n";
198
199    // print integers1 size and contents
200    cout << "Size of array integers1 is "
201         << integers1.getSize()
202         << "\nArray after initialization:\n"
203         << integers1 << '\n';
204
205    // print integers2 size and contents
206    cout << "Size of array integers2 is "
207         << integers2.getSize()
208         << "\nArray after initialization:\n"
209         << integers2 << '\n';
210
211    // input and print integers1 and integers2
212    cout << "Input 17 integers:\n";
213    cin >> integers1 >> integers2;
214    cout << "After input, the arrays contain:\n"
215         << "integers1:\n" << integers1
216         << "integers2:\n" << integers2 << '\n';
217
218    // use overloaded inequality (!=) operator
219    cout << "Evaluating: integers1 != integers2\n";
220    if ( integers1 != integers2 )
221       cout << "They are not equal\n";
222
223    // create array integers3 using integers1 as an
224    // initializer; print size and contents
225    Array integers3( integers1 );
226
```

```
227    cout << "\nSize of array integers3 is "
228        << integers3.getSize()
229        << "\nArray after initialization:\n"
230        << integers3 << '\n';
231
232    // use overloaded assignment (=) operator
233    cout << "Assigning integers2 to integers1:\n";
234    integers1 = integers2;
235    cout << "integers1:\n" << integers1
236        << "integers2:\n" << integers2 << '\n';
237
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 }
```

Program Output

```
# of arrays instantiated = 0
# of arrays instantiated = 2

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

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:
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 to char
  A::operator int() const;        //A to int
  A::operator otherClass() const; //A to otherClass
  ```

  - When compiler sees **(char *) s** it calls

    ```
    s.operator char*()
    ```

18.9 Converting between Types (II)

- The compiler can call these functions to create temporary objects.
  - If **s** is not of type **char ***

  Calls **A::operator char *() 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);