Illustrations List *(Main Page)*

Operators that can be overloaded

```
+       -       *       /       %       ^       &       |
~       !       =       <       >       +=      -=      *=
/=      %=      ^=      &=      |=      <<      >>      >>=
<<=     ==      !=      <=      >=      &&      ||      ++
--      ->*     ,       ->      []      ()      new     delete
new[]   delete[]
```

Fig. 8.1 Operators that can be overloaded.

Operators that cannot be overloaded

```
.           .*          ::          ?:          sizeof
```

Fig. 8.2 Operators that cannot be overloaded.

```
1  // Fig. 8.3: fig08_03.cpp
2  // Overloading the stream-insertion and
3  // stream-extraction operators.
4  #include <iostream.h>
5  #include <iomanip.h>
6
7  class PhoneNumber {
8     friend ostream &operator<<( ostream&, const PhoneNumber & );
9     friend istream &operator>>( istream&, PhoneNumber & );
10
11 private:
12    char areaCode[ 4 ];  // 3-digit area code and null
13    char exchange[ 4 ];  // 3-digit exchange and null
14    char line[ 5 ];      // 4-digit line and null
15 };
```

Fig. 8.3 User-defined stream-insertion and stream-extraction operators (part 1 of 2).

```
16
17 // Overloaded stream-insertion operator (cannot be
18 // a member function if we would like to invoke it with
19 // cout << somePhoneNumber;).
20 ostream &operator<<( ostream &output, const PhoneNumber &num )
21 {
22    output << "(" << num.areaCode << ") "
23           << num.exchange << "-" << num.line;
24    return output;     // enables cout << a << b << c;
25 }
26
27 istream &operator>>( istream &input, PhoneNumber &num )
28 {
29    input.ignore();                    // skip (
30    input >> setw( 4 ) >> num.areaCode; // input area code
31    input.ignore( 2 );                 // skip ) and space
32    input >> setw( 4 ) >> num.exchange; // input exchange
33    input.ignore();                    // skip dash (-)
34    input >> setw( 5 ) >> num.line;    // input line
35    return input;      // enables cin >> a >> b >> c;
36 }
37
38 int main()
39 {
40    PhoneNumber phone; // create object phone
41
42    cout << "Enter phone number in the form (123) 456-7890:\n";
43
44    // cin >> phone invokes operator>> function by
45    // issuing the call operator>>( cin, phone ).
46    cin >> phone;
47
48    // cout << phone invokes operator<< function by
49    // issuing the call operator<<( cout, phone ).
50    cout << "The phone number entered was: " << phone << endl;
51    return 0;
52 }
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

Fig. 8.3 User-defined stream-insertion and stream-extraction operators (part 2 of 2).

```
1  // Fig. 8.4: array1.h
2  // Simple class Array (for integers)
3  #ifndef ARRAY1_H
4  #define ARRAY1_H
5
6  #include <iostream.h>
7
8  class Array {
9     friend ostream &operator<<( ostream &, const Array & );
10    friend istream &operator>>( istream &, Array & );
11 public:
12    Array( int = 10 );                  // default constructor
13    Array( const Array & );             // copy constructor
14    ~Array();                           // destructor
15    int getSize() const;                // return size
16    const Array &operator=( const Array & ); // assign arrays
17    bool operator==( const Array & ) const;  // compare equal
18
19    // Determine if two arrays are not equal and
20    // return true, otherwise return false (uses operator==).
21    bool operator!=( const Array &right ) const
22       { return ! ( *this == right ); }
23
24    int &operator[]( int );             // subscript operator
25    const int &operator[]( int ) const;  // subscript operator
26    static int getArrayCount();         // Return count of
27                                        // arrays instantiated.
28 private:
29    int size; // size of the array
30    int *ptr; // pointer to first element of array
31    static int arrayCount;  // # of Arrays instantiated
32 };
33
34 #endif
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 1 of 8).

```
35 // Fig 8.4: array1.cpp
36 // Member function definitions for class Array
37 #include <iostream.h>
38 #include <iomanip.h>
39 #include <stdlib.h>
40 #include <assert.h>
41 #include "array1.h"
42
43 // Initialize static data member at file scope
44 int Array::arrayCount = 0;   // no objects yet
45
46 // Default constructor for class Array (default size 10)
47 Array::Array( int arraySize )
48 {
49    size = ( arraySize > 0 ? arraySize : 10 );
50    ptr = new int[ size ]; // create space for array
51    assert( ptr != 0 );    // terminate if memory not allocated
52    ++arrayCount;          // count one more object
53
54    for ( int i = 0; i < size; i++ )
55       ptr[ i ] = 0;          // initialize array
56 }
57
58 // Copy constructor for class Array
59 // must receive a reference to prevent infinite recursion
60 Array::Array( const Array &init ) : size( init.size )
61 {
```

```
62     ptr = new int[ size ]; // create space for array
63     assert( ptr != 0 );    // terminate if memory not allocated
64     ++arrayCount;          // count one more object
65
66     for ( int i = 0; i < size; i++ )
67        ptr[ i ] = init.ptr[ i ];  // copy init into object
68  }
69
70  // Destructor for class Array
71  Array::~Array()
72  {
73     delete [] ptr;            // reclaim space for array
74     --arrayCount;             // one fewer objects
75  }
76
77  // Get the size of the array
78  int Array::getSize() const { return size; }
79
80  // Overloaded assignment operator
81  // const return avoids: ( a1 = a2 ) = a3
82  const Array &Array::operator=( const Array &right )
83  {
84     if ( &right != this ) {  // check for self-assignment
85
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 2 of 8).

```
86        // for arrays of different sizes, deallocate original
87        // left side array, then allocate new left side array.
88        if ( size != right.size ) {
89           delete [] ptr;         // reclaim space
90           size = right.size;     // resize this object
91           ptr = new int[ size ]; // create space for array copy
92           assert( ptr != 0 );    // terminate if not allocated
93        }
94
95        for ( int i = 0; i < size; i++ )
96           ptr[ i ] = right.ptr[ i ];  // copy array into object
97     }
98
99     return *this;   // enables x = y = z;
100 }
101
102 // Determine if two arrays are equal and
103 // return true, otherwise return false.
104 bool Array::operator==( const Array &right ) const
105 {
106    if ( size != right.size )
107       return false;    // arrays of different sizes
108
109    for ( int i = 0; i < size; i++ )
110       if ( ptr[ i ] != right.ptr[ i ] )
111          return false; // arrays are not equal
112
113    return true;        // arrays are equal
114 }
115
116 // Overloaded subscript operator for non-const Arrays
117 // reference return creates an lvalue
118 int &Array::operator[]( int subscript )
119 {
120    // check for subscript out of range error
121    assert( 0 <= subscript && subscript < size );
122
```

```
123    return ptr[ subscript ]; // reference return
124 }
125
126 // Overloaded subscript operator for const Arrays
127 // const reference return creates an rvalue
128 const int &Array::operator[]( int subscript ) const
129 {
130    // check for subscript out of range error
131    assert( 0 <= subscript && subscript < size );
132
133    return ptr[ subscript ]; // const reference return
134 }
135
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 3 of 8).

```
136 // Return the number of Array objects instantiated
137 // static functions cannot be const
138 int Array::getArrayCount() { return arrayCount; }
139
140 // Overloaded input operator for class Array;
141 // inputs values for entire array.
142 istream &operator>>( istream &input, Array &a )
143 {
144    for ( int i = 0; i < a.size; i++ )
145       input >> a.ptr[ i ];
146
147    return input;   // enables cin >> x >> y;
148 }
149
150 // Overloaded output operator for class Array
151 ostream &operator<<( ostream &output, const Array &a )
152 {
153    int i;
154
155    for ( i = 0; i < a.size; i++ ) {
156       output << setw( 12 ) << a.ptr[ i ];
157
158       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
159          output << endl;
160    }
161
162    if ( i % 4 != 0 )
163       output << endl;
164
165    return output;   // enables cout << x << y;
166 }
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 4 of 8).

```
// Fig. 8.4: fig08_04.cpp
// Driver for simple class Array
#include <iostream.h>
#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";

```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 5 of 8).

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

   // print integers2 size and contents
   cout << "Size of array integers2 is "
        << integers2.getSize()
        << "\nArray after initialization:\n"
        << integers2 << '\n';

   // input and print integers1 and integers2
   cout << "Input 17 integers:\n";
   cin >> integers1 >> integers2;
   cout << "After input, the arrays contain:\n"
        << "integers1:\n" << integers1
        << "integers2:\n" << 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()
        << "\nArray after initialization:\n"
        << integers3 << '\n';

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

   // use overloaded equality (==) operator
   cout << "Evaluating: integers1 == integers2\n";
   if ( integers1 == integers2 )
      cout << "They are equal\n\n";

   // use overloaded subscript operator to create rvalue
```

```
228    cout << "integers1[5] is " << integers1[5] << '\n';
229
230    // use overloaded subscript operator to create lvalue
231    cout << "Assigning 1000 to integers1[5]\n";
232    integers1[5] = 1000;
233    cout << "integers1:\n" << integers1 << '\n';
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 6 of 8).

```
234
235    // attempt to use out of range subscript
236    cout << "Attempt to assign 1000 to integers1[15]" << endl;
237    integers1[15] = 1000;  // ERROR: out of range
238
239    return 0;
240 }
```

```
# 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
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 7 of 8).

```
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 87

abnormal program termination
```

Fig. 8.4 Demonstrating an **Array** class with overloaded operators (part 8 of 8).

```
1  // Fig. 8.5: string1.h
2  // Definition of a String class
3  #ifndef STRING1_H
4  #define STRING1_H
5
6  #include <iostream.h>
7
8  class String {
9     friend ostream &operator<<( ostream &, const String & );
10    friend istream &operator>>( istream &, String & );
11
12 public:
13    String( const char * = "" ); // conversion/default ctor
14    String( const String & );    // copy constructor
15    ~String();                   // destructor
16    const String &operator=( const String & );  // assignment
17    const String &operator+=( const String & ); // concatenation
18    bool operator!() const;                   // is String empty?
19    bool operator==( const String & ) const; // test s1 == s2
20    bool operator<( const String & ) const;  // test s1 < s2
21
22    // test s1 != s2
23    bool operator!=( const String & right ) const
24       { return !( *this == right ); }
25
26    // test s1 > s2
27    bool operator>( const String &right ) const
28       { return right < *this; }
29
```

Fig. 8.5 Definition of a basic **String** class (part 1 of 9).

```
30     // test s1 <= s2
31     bool operator<=( const String &right ) const
32        { return !( right < *this ); }
33
34     // test s1 >= s2
35     bool operator>=( const String &right ) const
36        { return !( *this < right ); }
37
38     char &operator[]( int );             // subscript operator
39     const char &operator[]( int ) const; // subscript operator
40     String &operator()( int, int ); // return a substring
41     int getLength() const;          // return string length
42
43  private:
44     int length;                   // string length
45     char *sPtr;                   // pointer to start of string
46
47     void setString( const char * );  // utility function
48  };
49
50  #endif
```

Fig. 8.5 Definition of a basic **String** class (part 2 of 9).

```
51  // Fig. 8.5: string1.cpp
52  // Member function definitions for class String
53  #include <iostream.h>
54  #include <iomanip.h>
55  #include <string.h>
56  #include <assert.h>
57  #include "string1.h"
58
59  // Conversion constructor: Convert char * to String
60  String::String( const char *s ) : length( strlen( s ) )
61  {
62     cout << "Conversion constructor: " << s << '\n';
63     setString( s );         // call utility function
64  }
65
66  // Copy constructor
67  String::String( const String &copy ) : length( copy.length )
68  {
69     cout << "Copy constructor: " << copy.sPtr << '\n';
70     setString( copy.sPtr ); // call utility function
71  }
72
73  // Destructor
74  String::~String()
75  {
76     cout << "Destructor: " << sPtr << '\n';
```

Fig. 8.5 Definition of a basic **String** class (part 3 of 9).

```
77     delete [] sPtr;         // reclaim string
78  }
79
80  // Overloaded = operator; avoids self assignment
81  const String &String::operator=( const String &right )
82  {
83     cout << "operator= called\n";
84
85     if ( &right != this ) {         // avoid self assignment
86        delete [] sPtr;              // prevents memory leak
87        length = right.length;       // new String length
88        setString( right.sPtr );     // call utility function
89     }
90     else
91        cout << "Attempted assignment of a String to itself\n";
92
93     return *this;   // enables cascaded assignments
94  }
95
96  // Concatenate right operand to this object and
97  // store in this object.
98  const String &String::operator+=( const String &right )
99  {
100    char *tempPtr = sPtr;        // hold to be able to delete
101    length += right.length;      // new String length
102    sPtr = new char[ length + 1 ]; // create space
103    assert( sPtr != 0 );   // terminate if memory not allocated
104    strcpy( sPtr, tempPtr );     // left part of new String
105    strcat( sPtr, right.sPtr );  // right part of new String
106    delete [] tempPtr;           // reclaim old space
107    return *this;                // enables cascaded calls
108 }
109
110 // Is this String empty?
111 bool String::operator!() const { return length == 0; }
112
113 // Is this String equal to right String?
114 bool String::operator==( const String &right ) const
115    { return strcmp( sPtr, right.sPtr ) == 0; }
116
117 // Is this String less than right String?
118 bool String::operator<( const String &right ) const
119    { return strcmp( sPtr, right.sPtr ) < 0; }
120
121 // Return a reference to a character in a String as an lvalue.
122 char &String::operator[]( int subscript )
123 {
124    // First test for subscript out of range
125    assert( subscript >= 0 && subscript < length );
126
```

Fig. 8.5 Definition of a basic **`String`** class (part 4 of 9).

```
127    return sPtr[ subscript ];  // creates lvalue
128 }
129
130 // Return a reference to a character in a String as an rvalue.
131 const char &String::operator[]( int subscript ) const
132 {
133    // First test for subscript out of range
134    assert( subscript >= 0 && subscript < length );
135
136    return sPtr[ subscript ];  // creates rvalue
137 }
138
139 // Return a substring beginning at index and
140 // of length subLength as a reference to a String object.
141 String &String::operator()( int index, int subLength )
142 {
143    // ensure index is in range and substring length >= 0
144    assert( index >= 0 && index < length && subLength >= 0 );
145
146    String *subPtr = new String; // empty String
147    assert( subPtr != 0 ); // ensure new String allocated
148
149    // determine length of substring
150    if ( ( subLength == 0 ) || ( index + subLength > length ) )
151       subPtr->length = length - index + 1;
152    else
153       subPtr->length = subLength + 1;
154
155    // allocate memory for substring
156    delete subPtr->sPtr; // delete character array from object
157    subPtr->sPtr = new char[ subPtr->length ];
158    assert( subPtr->sPtr != 0 ); // ensure space allocated
159
160    // copy substring into new String
161    strncpy( subPtr->sPtr, &sPtr[ index ], subPtr->length );
162    subPtr->sPtr[ subPtr->length ] = '\0'; // terminate String
163
164    return *subPtr;          // return new String
165 }
166
167 // Return string length
168 int String::getLength() const { return length; }
169
170 // Utility function to be called by constructors and
171 // assignment operator.
172 void String::setString( const char *string2 )
173 {
174    sPtr = new char[ length + 1 ]; // allocate storage
175    assert( sPtr != 0 );  // terminate if memory not allocated
176    strcpy( sPtr, string2 );       // copy literal to object
177 }
```

Fig. 8.5 Definition of a basic **`String`** class (part 5 of 9).

```
178
179 // Overloaded output operator
180 ostream &operator<<( ostream &output, const String &s )
181 {
182    output << s.sPtr;
183    return output;   // enables cascading
184 }
185
186 // Overloaded input operator
187 istream &operator>>( istream &input, String &s )
188 {
189    char temp[ 100 ];   // buffer to store input
190
191    input >> setw( 100 ) >> temp;
192    s = temp;        // use String class assignment operator
193    return input;    // enables cascading
194 }
```

Fig. 8.5 Member function definitions for class **String** (part 6 of 9).

```
195 // Fig. 8.5: fig08_05.cpp
196 // Driver for class String
197 #include <iostream.h>
198 #include "string1.h"
199
200 int main()
201 {
202    String s1( "happy" ), s2( " birthday" ), s3;
203
204    // test overloaded equality and relational operators
205    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
206         << "\"; s3 is \"" << s3 << '\"'
207         << "\nThe results of comparing s2 and s1:"
208         << "\ns2 == s1 yields "
209         << ( s2 == s1 ? "true" : "false" )
210         << "\ns2 != s1 yields "
211         << ( s2 != s1 ? "true" : "false" )
212         << "\ns2 >  s1 yields "
213         << ( s2 > s1 ? "true" : "false" )
214         << "\ns2 <  s1 yields "
215         << ( s2 < s1 ? "true" : "false" )
216         << "\ns2 >= s1 yields "
217         << ( s2 >= s1 ? "true" : "false" )
218         << "\ns2 <= s1 yields "
219         << ( s2 <= s1 ? "true" : "false" );
220
221    // test overloaded String empty (!) operator
222    cout << "\n\nTesting !s3:\n";
223    if ( !s3 ) {
224       cout << "s3 is empty; assigning s1 to s3;\n";
```

Fig. 8.5 Definition of a basic **String** class (part 7 of 9).

```
225       s3 = s1;              // test overloaded assignment
226       cout << "s3 is \"" << s3 << "\"";
227    }
228
229    // test overloaded String concatenation operator
230    cout << "\n\ns1 += s2 yields s1 = ";
231    s1 += s2;                // test overloaded concatenation
232    cout << s1;
233
234    // test conversion constructor
235    cout << "\n\ns1 += \" to you\" yields\n";
236    s1 += " to you";         // test conversion constructor
237    cout << "s1 = " << s1 << "\n\n";
238
239    // test overloaded function call operator () for substring
240    cout << "The substring of s1 starting at\n"
241         << "location 0 for 14 characters, s1(0, 14), is:\n"
242         << s1( 0, 14 ) << "\n\n";
243
244    // test substring "to-end-of-String" option
245    cout << "The substring of s1 starting at\n"
246         << "location 15, s1(15, 0), is: "
247         << s1( 15, 0 ) << "\n\n";  // 0 is "to end of string"
248
249    // test copy constructor
250    String *s4Ptr = new String(s1);
251    cout << "*s4Ptr = " << *s4Ptr << "\n\n";
252
253    // test assignment (=) operator with self-assignment
254    cout << "assigning *s4Ptr to *s4Ptr\n";
255    *s4Ptr = *s4Ptr;          // test overloaded assignment
256    cout << "*s4Ptr = " << *s4Ptr << '\n';
257
258    // test destructor
259    delete s4Ptr;
260
261    // test using subscript operator to create lvalue
262    s1[ 0 ] = 'H';
263    s1[ 6 ] = 'B';
264    cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
265         << s1 << "\n\n";
266
267    // test subscript out of range
268    cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
269    s1[ 30 ] = 'd';     // ERROR: subscript out of range
270
271    return 0;
272 }
```

Fig. 8.5 Definition of a basic **`String`** class (part 8 of 9).

```
Conversion constructor: happy
Conversion constructor:  birthday
Conversion constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""
The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion constructor:  to you
Destructor:  to you
s1 = happy birthday to you

Conversion constructor:
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Conversion constructor:
The substring of s1 starting at
location 15, s1(15, 0), is: to you

Copy constructor: happy birthday to you
*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday
to you

Attempt to assign 'd' to s1[30] yields:
Assertion failed: subscript >= 0 && subscript < length,
file String1.cpp, line 76

abnormal program termination
```

Fig. 8.5 Definition of a basic **`String`** class (part 9 of 9).

```
1  // Fig. 8.6: date1.h
2  // Definition of class Date
3  #ifndef DATE1_H
4  #define DATE1_H
5  #include <iostream.h>
6
7  class Date {
8     friend ostream &operator<<( ostream &, const Date & );
9
10 public:
11    Date( int m = 1, int d = 1, int y = 1900 ); // constructor
12    void setDate( int, int, int ); // set the date
13    Date &operator++();            // preincrement operator
14    Date operator++( int );        // postincrement operator
15    const Date &operator+=( int ); // add days, modify object
16    bool leapYear( int );          // is this a leap year?
17    bool endOfMonth( int );        // is this end of month?
18
19 private:
20    int month;
21    int day;
22    int year;
23
24    static const int days[];       // array of days per month
25    void helpIncrement();          // utility function
26 };
27
28 #endif
```

Fig. 8.6 Class **Date** with overloaded increment operators (part 1 of 7).

```
29 // Fig. 8.6: date1.cpp
30 // Member function definitions for Date class
31 #include <iostream.h>
32 #include "date1.h"
33
34 // Initialize static member at file scope;
35 // one class-wide copy.
36 const int Date::days[] = { 0, 31, 28, 31, 30, 31, 30,
37                            31, 31, 30, 31, 30, 31 };
38
39 // Date constructor
40 Date::Date( int m, int d, int y ) { setDate( m, d, y ); }
41
42 // Set the date
43 void Date::setDate( int mm, int dd, int yy )
44 {
45    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
46    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
47
```

Fig. 8.6 Class **Date** with overloaded increment operators (part 2 of 7).

```
48     // test for a leap year
49     if ( month == 2 && leapYear( year ) )
50        day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
51     else
52        day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
53  }
54
55  // Preincrement operator overloaded as a member function.
56  Date &Date::operator++()
57  {
58     helpIncrement();
59     return *this;  // reference return to create an lvalue
60  }
61
62  // Postincrement operator overloaded as a member function.
63  // Note that the dummy integer parameter does not have a
64  // parameter name.
65  Date Date::operator++( int )
66  {
67     Date temp = *this;
68     helpIncrement();
69
70     // return non-incremented, saved, temporary object
71     return temp;   // value return; not a reference return
72  }
73
74  // Add a specific number of days to a date
75  const Date &Date::operator+=( int additionalDays )
76  {
77     for ( int i = 0; i < additionalDays; i++ )
78        helpIncrement();
79
80     return *this;    // enables cascading
81  }
82
83  // If the year is a leap year, return true;
84  // otherwise, return false
85  bool Date::leapYear( int y )
86  {
87     if ( y % 400 == 0 || ( y % 100 != 0 && y % 4 == 0 ) )
88        return true;   // a leap year
89     else
90        return false;  // not a leap year
91  }
92
```

Fig. 8.6 Class **Date** with overloaded increment operators (part 3 of 7).

```
93  // Determine if the day is the end of the month
94  bool Date::endOfMonth( int d )
95  {
96     if ( month == 2 && leapYear( year ) )
97        return d == 29; // last day of Feb. in leap year
98     else
99        return d == days[ month ];
100 }
101
102 // Function to help increment the date
103 void Date::helpIncrement()
104 {
105    if ( endOfMonth( day ) && month == 12 ) {  // end year
106       day = 1;
107       month = 1;
108       ++year;
```

```
109    }
110    else if ( endOfMonth( day ) ) {            // end month
111       day = 1;
112       ++month;
113    }
114    else       // not end of month or year; increment day
115       ++day;
116 }
117
118 // Overloaded output operator
119 ostream &operator<<( ostream &output, const Date &d )
120 {
121    static char *monthName[ 13 ] = { "", "January",
122       "February", "March", "April", "May", "June",
123       "July", "August", "September", "October",
124       "November", "December" };
125
126    output << monthName[ d.month ] << ' '
127           << d.day << ", " << d.year;
128
129    return output;   // enables cascading
130 }
```

Fig. 8.6 Class **Date** with overloaded increment operators (part 4 of 7).

```
131 // Fig. 8.6: fig08_06.cpp
132 // Driver for class Date
133 #include <iostream.h>
134 #include "date1.h"
135
136 int main()
137 {
138    Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );
```

Fig. 8.6 Class **Date** with overloaded increment operators (part 5 of 7).

```
139    cout << "d1 is " << d1
140         << "\nd2 is " << d2
141         << "\nd3 is " << d3 << "\n\n";
142
143    cout << "d2 += 7 is " << ( d2 += 7 ) << "\n\n";
144
145    d3.setDate( 2, 28, 1992 );
146    cout << "  d3 is " << d3;
147    cout << "\n++d3 is " << ++d3 << "\n\n";
148
149    Date d4( 3, 18, 1969 );
150
151    cout << "Testing the preincrement operator:\n"
152         << "  d4 is " << d4 << '\n';
153    cout << "++d4 is " << ++d4 << '\n';
154    cout << "  d4 is " << d4 << "\n\n";
155
156    cout << "Testing the postincrement operator:\n"
157         << "  d4 is " << d4 << '\n';
158    cout << "d4++ is " << d4++ << '\n';
159    cout << "  d4 is " << d4 << endl;
160
161    return 0;
162 }
```

Fig. 8.6 Class **Date** with overloaded increment operators (part 6 of 7).

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992

Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969

Testing the postincrement operator:
  d4 is March 19, 1969
d4++ is March 19, 1969
  d4 is March 20, 1969
```

Fig. 8.6 Output from driver for class **Date** (part 7 of 7).

```
1  // Fig. 8.7: complex1.h
2  // Definition of class Complex
3  #ifndef COMPLEX1_H
4  #define COMPLEX1_H
5
6  class Complex {
7  public:
8     Complex( double = 0.0, double = 0.0 );       // constructor
9     Complex operator+( const Complex & ) const;  // addition
10    Complex operator-( const Complex & ) const;  // subtraction
11    const Complex &operator=( const Complex & ); // assignment
12    void print() const;                          // output
13 private:
14    double real;       // real part
15    double imaginary;  // imaginary part
16 };
17
18 #endif
```

Fig. 8.7 Demonstrating class **Complex** (part 1 of 5).

```
19 // Fig. 8.7: complex1.cpp
20 // Member function definitions for class Complex
21 #include <iostream.h>
22 #include "complex1.h"
23
24 // Constructor
25 Complex::Complex( double r, double i )
26    : real( r ), imaginary( i ) { }
27
28 // Overloaded addition operator
29 Complex Complex::operator+( const Complex &operand2 ) const
30 {
31    return Complex( real + operand2.real,
32                    imaginary + operand2.imaginary );
33 }
34
```

```
35  // Overloaded subtraction operator
36  Complex Complex::operator-( const Complex &operand2 ) const
37  {
38     return Complex( real - operand2.real,
39                     imaginary - operand2.imaginary );
40  }
```

Fig. 8.7 Demonstrating class **Complex** (part 2 of 5).

```
41
42  // Overloaded = operator
43  const Complex& Complex::operator=( const Complex &right )
44  {
45     real = right.real;
46     imaginary = right.imaginary;
47     return *this;   // enables cascading
48  }
49
50  // Display a Complex object in the form: (a, b)
51  void Complex::print() const
52     { cout << '(' << real << ", " << imaginary << ')'; }
```

Fig. 8.7 Demonstrating class **Complex** (part 3 of 5).

```
53  // Fig. 8.7: fig08_07.cpp
54  // Driver for class Complex
55  #include <iostream.h>
56  #include "complex1.h"
57
58  int main()
59  {
60     Complex x, y( 4.3, 8.2 ), z( 3.3, 1.1 );
61
62     cout << "x: ";
63     x.print();
64     cout << "\ny: ";
65     y.print();
66     cout << "\nz: ";
67     z.print();
68
69     x = y + z;
70     cout << "\n\nx = y + z:\n";
71     x.print();
72     cout << " = ";
73     y.print();
74     cout << " + ";
75     z.print();
76
77     x = y - z;
78     cout << "\n\nx = y - z:\n";
79     x.print();
80     cout << " = ";
81     y.print();
82     cout << " - ";
83     z.print();
84     cout << endl;
85
86     return 0;
87  }
```

Fig. 8.7 Demonstrating class **Complex** (part 4 of 5).

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

Fig. 8.7 Demonstrating class **Complex** (part 5 of 5).

```
1  // Fig. 8.8: hugeint1.h
2  // Definition of the HugeInt class
3  #ifndef HUGEINT1_H
4  #define HUGEINT1_H
5
6  #include <iostream.h>
7
8  class HugeInt {
9     friend ostream &operator<<( ostream &, HugeInt & );
10 public:
11    HugeInt( long = 0 );       // conversion/default constructor
12    HugeInt( const char * );           // conversion constructor
13    HugeInt operator+( HugeInt & );    // add another HugeInt
14    HugeInt operator+( int );          // add an int
15    HugeInt operator+( const char * ); // add an int in a char *
16 private:
17    short integer[30];
18 };
19
20 #endif
```

Fig. 8.8 A user-defined huge integer class (part 1 of 5).

```
21 // Fig. 8.8: hugeint1.cpp
22 // Member and friend function definitions for class HugeInt
23 #include <string.h>
24 #include "hugeint1.h"
25
26 // Conversion constructor
27 HugeInt::HugeInt( long val )
28 {
29    int i;
30
31    for ( i = 0; i <= 29; i++ )
32       integer[ i ] = 0;   // initialize array to zero
33
34    for ( i = 29; val != 0 && i >= 0; i-- ) {
35       integer[ i ] = val % 10;
36       val /= 10;
37    }
38 }
39
40 HugeInt::HugeInt( const char *string )
41 {
42    int i, j;
43
```

```
44     for ( i = 0; i <= 29; i++ )
45        integer[ i ] = 0;
46
47     for ( i = 30 - strlen( string ), j = 0; i <= 29; i++, j++ )
48        integer[ i ] = string[ j ] - '0';
49  }
50
51  // Addition
52  HugeInt HugeInt::operator+( HugeInt &op2 )
53  {
54     HugeInt temp;
55     int carry = 0;
56
57     for ( int i = 29; i >= 0; i-- ) {
58        temp.integer[ i ] = integer[ i ] +
59                            op2.integer[ i ] + carry;
60
61        if ( temp.integer[ i ] > 9 ) {
62           temp.integer[ i ] %= 10;
63           carry = 1;
64        }
65        else
66           carry = 0;
67     }
68
69     return temp;
70  }
```

Fig. 8.8 A user-defined huge integer class (part 2 of 5).

```
71
72  // Addition
73  HugeInt HugeInt::operator+( int op2 )
74     { return *this + HugeInt( op2 ); }
75
76  // Addition
77  HugeInt HugeInt::operator+( const char *op2 )
78     { return *this + HugeInt( op2 ); }
79
80  ostream& operator<<( ostream &output, HugeInt &num )
81  {
82     int i;
83
84     for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i++ )
85        ; // skip leading zeros
86
87     if ( i == 30 )
88        output << 0;
89     else
90        for ( ; i <= 29; i++ )
91           output << num.integer[ i ];
92
93     return output;
94  }
```

Fig. 8.8 A user-defined huge integer class (part 3 of 5).

```
95  // Fig. 8.8: fig08_08.cpp
96  // Test driver for HugeInt class
97  #include <iostream.h>
98  #include "hugeint1.h"
99
100 int main()
101 {
102    HugeInt n1( 7654321 ), n2( 7891234 ),
103            n3( "99999999999999999999999999999" ),
104            n4( "1" ), n5;
105
106    cout << "n1 is " << n1 << "\nn2 is " << n2
107         << "\nn3 is " << n3 << "\nn4 is " << n4
108         << "\nn5 is " << n5 << "\n\n";
109
110    n5 = n1 + n2;
111    cout << n1 << " + " << n2 << " = " << n5 << "\n\n";
112
113    cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 )
114         << "\n\n";
115
116    n5 = n1 + 9;
117    cout << n1 << " + " << 9 << " = " << n5 << "\n\n";
```

Fig. 8.8 A user-defined huge integer class (part 4 of 5).

```
118
119    n5 = n2 + "10000";
120    cout << n2 << " + " << "10000" << " = " << n5 << endl;
121
122    return 0;
123 }
```

```
n1 is 7654321
n2 is 7891234
n3 is 99999999999999999999999999999
n4 is 1
n5 is 0

7654321 + 7891234 = 15545555

99999999999999999999999999999 + 1
= 100000000000000000000000000000

7654321 + 9 = 7654330

7891234 + 10000 = 7901234
```

Fig. 8.8 A user-defined huge integer class (part 5 of 5).