11.8 *(Complex Class)* Consider class `Complex` shown in Figs. 11.19–11.21. The class enables operations on so-called *complex numbers*. These are numbers of the form `realPart + imaginaryPart * i`, where i has the value

$$\sqrt{-1}$$

a) Modify the class to enable input and output of complex numbers via overloaded `>>` and `<<` operators, respectively (you should remove the `print` function from the class).
b) Overload the multiplication operator to enable multiplication of two complex numbers as in algebra.
c) Overload the `==` and `!=` operators to allow comparisons of complex numbers.

```
1  // Fig. 11.20: Complex.h
2  // Complex class definition.
3  #ifndef COMPLEX_H
4  #define COMPLEX_H
5
6  class Complex
7  {
8  public:
9     Complex( double = 0.0, double = 0.0 ); // constructor
```

Fig. 11.19 | `Complex` class definition. (Part 1 of 2.)

```
10       Complex operator+( const Complex & ) const; // addition
11       Complex operator-( const Complex & ) const; // subtraction
12       void print() const; // output
13    private:
14       double real; // real part
15       double imaginary; // imaginary part
16    }; // end class Complex
17
18    #endif
```

Fig. 11.19 | Complex class definition. (Part 2 of 2.)

```
1   // Fig. 11.21: Complex.cpp
2   // Complex class member-function definitions.
3   #include <iostream>
4   #include "Complex.h" // Complex class definition
5   using namespace std;
6
7   // Constructor
8   Complex::Complex( double realPart, double imaginaryPart )
9      : real( realPart ),
10     imaginary( imaginaryPart )
11  {
12     // empty body
13  } // end Complex constructor
14
15  // addition operator
16  Complex Complex::operator+( const Complex &operand2 ) const
17  {
18     return Complex( real + operand2.real,
19        imaginary + operand2.imaginary );
20  } // end function operator+
21
22  // subtraction operator
23  Complex Complex::operator-( const Complex &operand2 ) const
24  {
25     return Complex( real - operand2.real,
26        imaginary - operand2.imaginary );
27  } // end function operator-
28
29  // display a Complex object in the form: (a, b)
30  void Complex::print() const
31  {
32     cout << '(' << real << ", " << imaginary << ')';
33  } // end function print
```

Fig. 11.20 | Complex class member-function definitions.

```
1   // Fig. 11.22: fig11_22.cpp
2   // Complex class test program.
3   #include <iostream>
4   #include "Complex.h"
5   using namespace std;
6
```

Fig. 11.21 | Complex numbers. (Part 1 of 2.)

```
int main()
{
   Complex x;
   Complex y( 4.3, 8.2 );
   Complex z( 3.3, 1.1 );

   cout << "x: ";
   x.print();
   cout << "\ny: ";
   y.print();
   cout << "\nz: ";
   z.print();

   x = y + z;
   cout << "\n\nx = y + z:" << endl;
   x.print();
   cout << " = ";
   y.print();
   cout << " + ";
   z.print();

   x = y - z;
   cout << "\n\nx = y - z:" << endl;
   x.print();
   cout << " = ";
   y.print();
   cout << " - ";
   z.print();
   cout << endl;
} // end main
```

```
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. 11.21 | Complex numbers. (Part 2 of 2.)

11.9 ***(HugeInt Class)*** A machine with 32-bit integers can represent integers in the range of approximately –2 billion to +2 billion. This fixed-size restriction is rarely troublesome, but there are applications in which we would like to be able to use a much wider range of integers. This is what C++ was built to do, namely, create powerful new data types. Consider class `HugeInt` of Figs. 11.22–11.24. Study the class carefully, then answer the following:

a) Describe precisely how it operates.
b) What restrictions does the class have?
c) Overload the * multiplication operator.
d) Overload the / division operator.
e) Overload all the relational and equality operators.

[*Note:* We do not show an assignment operator or copy constructor for class `HugeInteger`, because the assignment operator and copy constructor provided by the compiler are capable of copying the entire array data member properly.]

```
// Fig. 11.23: Hugeint.h
// HugeInt class definition.
#ifndef HUGEINT_H
#define HUGEINT_H

#include <iostream>
#include <string>
using namespace std;

class HugeInt
{
   friend ostream &operator<<( ostream &, const HugeInt & );
public:
   static const int digits = 30; // maximum digits in a HugeInt

   HugeInt( long = 0 ); // conversion/default constructor
   HugeInt( const string & ); // conversion constructor

   // addition operator; HugeInt + HugeInt
   HugeInt operator+( const HugeInt & ) const;

   // addition operator; HugeInt + int
   HugeInt operator+( int ) const;

   // addition operator;
   // HugeInt + string that represents large integer value
   HugeInt operator+( const string & ) const;
private:
   short integer[ digits ];
}; // end class HugetInt

#endif
```

Fig. 11.22 | HugeInt class definition.

```
// Fig. 11.24: Hugeint.cpp
// HugeInt member-function and friend-function definitions.
#include <cctype> // isdigit function prototype
#include "Hugeint.h" // HugeInt class definition
using namespace std;

// default constructor; conversion constructor that converts
// a long integer into a HugeInt object
HugeInt::HugeInt( long value )
{
   // initialize array to zero
   for ( int i = 0; i < digits; i++ )
      integer[ i ] = 0;

   // place digits of argument into array
   for ( int j = digits - 1; value != 0 && j >= 0; j-- )
   {
      integer[ j ] = value % 10;
      value /= 10;
   } // end for
} // end HugeInt default/conversion constructor

```

Fig. 11.23 | HugeInt class member-function and friend-function definitions. (Part 1 of 3.)

```
// conversion constructor that converts a character string
// representing a large integer into a HugeInt object
HugeInt::HugeInt( const string &number )
{
   // initialize array to zero
   for ( int i = 0; i < digits; i++ )
      integer[ i ] = 0;

   // place digits of argument into array
   int length = number.size();

   for ( int j = digits - length, k = 0; j < digits; j++, k++ )
      if ( isdigit( number[ k ] ) ) // ensure that character is a digit
         integer[ j ] = number[ k ] - '0';
} // end HugeInt conversion constructor

// addition operator; HugeInt + HugeInt
HugeInt HugeInt::operator+( const HugeInt &op2 ) const
{
   HugeInt temp; // temporary result
   int carry = 0;

   for ( int i = digits - 1; i >= 0; i-- )
   {
      temp.integer[ i ] = integer[ i ] + op2.integer[ i ] + carry;

      // determine whether to carry a 1
      if ( temp.integer[ i ] > 9 )
      {
         temp.integer[ i ] %= 10;  // reduce to 0-9
         carry = 1;
      } // end if
      else // no carry
         carry = 0;
   } // end for

   return temp; // return copy of temporary object
} // end function operator+

// addition operator; HugeInt + int
HugeInt HugeInt::operator+( int op2 ) const
{
   // convert op2 to a HugeInt, then invoke
   // operator+ for two HugeInt objects
   return *this + HugeInt( op2 );
} // end function operator+

// addition operator;
// HugeInt + string that represents large integer value
HugeInt HugeInt::operator+( const string &op2 ) const
{
   // convert op2 to a HugeInt, then invoke
   // operator+ for two HugeInt objects
   return *this + HugeInt( op2 );
} // end operator+

// overloaded output operator
ostream& operator<<( ostream &output, const HugeInt &num )
{
```

Fig. 11.23 | `HugeInt` class member-function and `friend`-function definitions. (Part 2 of 3.)

```
   int i;

   for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= HugeInt::digits ); i++ )
      ; // skip leading zeros

   if ( i == HugeInt::digits )
      output << 0;
   else
      for ( ; i < HugeInt::digits; i++ )
         output << num.integer[ i ];

   return output;
} // end function operator<<
```

Fig. 11.23 | `HugeInt` class member-function and `friend`-function definitions. (Part 3 of 3.)

```
// Fig. 11.25: fig11_25.cpp
// HugeInt test program.
#include <iostream>
#include "Hugeint.h"
using namespace std;

int main()
{
   HugeInt n1( 7654321 );
   HugeInt n2( 7891234 );
   HugeInt n3( "99999999999999999999999999999" );
   HugeInt n4( "1" );
   HugeInt n5;

   cout << "n1 is " << n1 << "\nn2 is " << n2
      << "\nn3 is " << n3 << "\nn4 is " << n4
      << "\nn5 is " << n5 << "\n\n";

   n5 = n1 + n2;
   cout << n1 << " + " << n2 << " = " << n5 << "\n\n";

   cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 ) << "\n\n";

   n5 = n1 + 9;
   cout << n1 << " + " << 9 << " = " << n5 << "\n\n";

   n5 = n2 + "10000";
   cout << n2 << " + " << "10000" << " = " << n5 << endl;
} // end main
```

```
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. 11.24 | Huge integers.

11.11 *(Polynomial Class)* Develop class `Polynomial`. The internal representation of a `Polynomial` is an array of terms. Each term contains a coefficient and an exponent, e.g., the term

$$2x^4$$

has the coefficient 2 and the exponent 4. Develop a complete class containing proper constructor and destructor functions as well as *set* and *get* functions. The class should also provide the following overloaded operator capabilities:

a) Overload the addition operator (+) to add two `Polynomials`.
b) Overload the subtraction operator (-) to subtract two `Polynomials`.
c) Overload the assignment operator to assign one `Polynomial` to another.
d) Overload the multiplication operator (*) to multiply two `Polynomials`.
e) Overload the addition assignment operator (+=), subtraction assignment operator (-=), and multiplication assignment operator (*=).