11

Operator Overloading

The whole difference between construction and creation is exactly this: that a thing constructed can only be loved after it is constructed; but a thing created is loved before it exists.
—Gilbert Keith Chesterton

The die is cast.
—Julius Caesar

Our doctor would never really operate unless it was necessary. He was just that way. If he didn't need the money, he wouldn't lay a hand on you.
—Herb Shriner

OBJECTIVES

In this chapter you'll learn:

- What operator overloading is and how it simplifies programming.
- To overload operators for user-defined classes.
- To overload unary and binary operators.
- To convert objects from one class to another class.
- To create `PhoneNumber`, `Array` and `Date` classes that demonstrate operator overloading.
- To use overloaded operators and other features of C++'s `string` class.
- To use keyword `explicit` to prevent the compiler from using single-argument constructors to perform implicit conversions.
- What operator overloading is and how it simplifies programming.

Assignment Checklist

Name: _____ Date: _____

Section: _____

Exercises	Assigned: Circle assignments	Date Due
Prelab Activities		
Matching	YES NO	
Fill in the Blank	11, 12, 13, 14, 15, 16, 17, 18, 19, 20	
Short Answer	21, 22, 23, 24, 25	
Programming Output	26, 27, 28, 29, 30	
Correct the Code	31, 32, 33	
Lab Exercises		
Lab Exercise 1 — String Concatenation	YES NO	
Lab Exercise 2 — Huge Integer	YES NO	
Follow-Up Questions and Activities	1, 2	
Lab Exercise 3 — Rational Numbers	YES NO	
Follow-Up Questions and Activities	1, 2, 3	
Debugging	YES NO	
Labs Provided by Instructor		
1.		
2.		
3.		
Postlab Activities		
Coding Exercises	1, 2, 3, 4	
Programming Challenges	1, 2	

Prelab Activities

Matching

Name: _____ **Date:** _____

Section: _____

After reading Chapter 11 of *C++ How to Program, Seventh Edition*, answer the given questions. These questions are intended to test and reinforce your understanding of key concepts and may be done either before the lab or during the lab.

For each term in the column on the left, write the corresponding letter for the description that best matches it from the column on the right.

Term	Description
___ 1. Self-assignment	a) Enables C++'s operators to have class objects as operands.
___ 2. Dangling pointer	b) A constructor that takes as its argument a reference to an object of the same class as the one in which the constructor is defined.
___ 3. Memberwise assignment	c) A C++ operator that cannot be overloaded.
___ 4. Conversion constructor	d) A constructor that transforms its one parameter into an object of the class.
___ 5. Copy constructor	e) Assigning an object to itself.
___ 6. Operator overloading	f) The default behavior of the = operator.
___ 7. Single-argument constructor	g) Problem that may occur when default memberwise copy is used on objects with dynamically allocated memory.
___ 8. `?:`	h) Any constructor of this type can be thought of as a conversion constructor.
___ 9. Pointer-based arrays	i) Provides member function `substr`.
___ 10. `string`	j) Do not provide range-checking.

Prelab Activities

Name:

Fill in the Blank

Name: _____ **Date:** _____

Section: _____

Fill in the blank for each of the following statements:

11. It is often necessary that non-member `operator` functions be _____ functions.

12. When overloading an operator, the function name must be the keyword _____ followed by the _____ for the operator being overloaded.

13. The _____ and _____ operators may be used by objects of any class without overloading.

14. An operator's precedence, number of operands and _____ cannot be changed by overloading.

15. It is not possible to create _____ for new operators; only a subset of the existing operators may be overloaded.

16. The compiler does not know how to convert between _____ types and built-in types—the programmer must explicitly specify how such conversions occur.

17. An overloaded _____ operator can take an arbitrarily large number of arguments.

18. _____ are invoked whenever a copy of an object is needed.

19. If the left operand of an operator must be an object of a different class, the operator function must be implemented as a _____ function.

20. `string` member function _____ returns the character at the specified location as an *lvalue* or *rvalue*, depending on the context in which the call appears.

Prelab Activities

Name:

Short Answer

Name: _____ **Date:** _____

Section: _____

In the space provided, answer each of the given questions. Your answers should be concise; aim for two or three sentences.

21. What is operator overloading? How does it contribute to C++'s extensibility?

22. How is operator overloading accomplished?

23. Why is choosing not to overload the assignment operator and using default memberwise copy a potentially dangerous thing to do?

Prelab Activities

Name:

Short Answer

24. Why are some operators overloaded as member functions while others are not?

25. How is the increment operator overloaded? How are both prefix increment and postfix increment supported?

Prelab Activities

Name:

Programming Output

Name: _____ **Date:** _____

Section: _____

For each of the given program segments, read the code and write the output in the space provided below each program. [*Note:* Do not execute these programs on a computer.]

26. What is output by the following code? Use class `PhoneNumber` (Fig. 11.3–Fig. 11.4) and the following numbers as input: (333) 555-7777 and (222) 555-9999

```
1  int main()
2  {
3     PhoneNumber bill;
4     PhoneNumber jane;
5
6     cout << "Enter Bill's phone number: ";
7     cin >> bill;
8
9     cout << "Enter Jane's phone number: ";
10    cin >> jane;
11
12    cout << "Bill's number is: " << bill << endl;
13    cout << "Jane's number is: " << jane << endl;
14 } // end main
```

Your answer:

Prelab Activities

Name:

Programming Output

27. What is output by the following program? Use the `PhoneNumber` class shown in Fig. 11.3–Fig. 11.4 and assume that the following phone number is entered:

    `d333qq111w7777`

```
1  int main()
2  {
3     PhoneNumber num;
4
5     cout << "Enter a phone number: ";
6     cin >> num;
7
8     cout << "That number was: " << num << endl;
9  } // end main
```

Your answer::

For *Programming Output Exercises 28* and *29*, use the class definition in Fig. L 11.1–Fig. L 11.2.

```
1  // Array.h
2  // Simple class Array (for integers)
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using namespace std;
8
9  // class Array definition
10 class Array
11 {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14
15 public:
16    Array( int = 10 );                    // default constructor
17    Array( const Array & );               // copy constructor
18    ~Array();                             // destructor
19    int getSize() const;                  // return size
20    const Array &operator=( const Array & ); // assignment operator
21    bool operator==( const Array & ) const;  // equality operator
22
23    // determine if two arrays are not equal and
24    // return true, otherwise return false (uses operator==)
25    bool operator!=( const Array &right ) const
26    {
27       return ! ( *this == right );
28
29    } // end function operator!=
```

Fig. L 11.1 | Array class (Part 1 of 2.)

Prelab Activities

Name:

Programming Output

```
30
31    int &operator[]( int );              // subscript operator
32    const int &operator[]( int ) const;  // subscript operator
33    static int getArrayCount();          // return number of
34                                         // arrays instantiated
35 private:
36    int size; // size of array
37    int *ptr; // pointer to first element of array
38    static int arrayCount;  // number of Arrays instantiated
39
40 }; // end class Array
41
42 #endif // ARRAY_H
```

Fig. L 11.1 | `Array` class. (Part 2 of 2.)

```
1  // Array.cpp
2  // Member function definitions for class Array
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib>
6  #include <new>
7  using namespace std;
8
9  #include "Array.h"
10
11 // initialize static data member at file scope
12 int Array::arrayCount = 0;   // no objects yet
13
14 // default constructor for class Array (default size 10)
15 Array::Array( int arraySize )
16 {
17    size = ( arraySize > 0 ? arraySize : 10 );
18    ptr = new int[ size ]; // create space for array
19    ++arrayCount;          // count one more object
20
21    for ( int i = 0; i < size; i++ )
22       ptr[ i ] = 0;          // initialize array
23
24 } // end class Array constructor
25
26 // copy constructor for class Array
27 // must receive reference to prevent infinite recursion
28 Array::Array( const Array &arrayToCopy ) : size( arrayToCopy.size )
29 {
30    ptr = new int[ size ]; // create space for array
31    ++arrayCount;          // count one more object
32
33    for ( int i = 0; i < size; i++ )
34       ptr[ i ] = arrayToCopy.ptr[ i ];  // copy arayToCopy into object
35
36 } // end copy constructor
37
```

Fig. L 11.2 | `Array.cpp`. (Part 1 of 4.)

Prelab Activities

Name:

Programming Output

```
38  // destructor for class Array
39  Array::~Array()
40  {
41     delete [] ptr;            // reclaim space for array
42     --arrayCount;             // one fewer object
43
44  } // end class Array destructor
45
46  // get size of array
47  int Array::getSize() const
48  {
49     return size;
50
51  } // end function getSize
52
53  // overloaded assignment operator
54  // const return avoids: ( a1 = a2 ) = a3
55  const Array &Array::operator=( const Array &right )
56  {
57     if ( &right != this ) {  // check for self-assignment
58
59        // for arrays of different sizes, deallocate original
60        // left side array, then allocate new left side array
61        if ( size != right.size ) {
62           delete [] ptr;         // reclaim space
63           size = right.size;     // resize this object
64           ptr = new int[ size ]; // create space for array copy
65
66        } // end if
67
68        for ( int i = 0; i < size; i++ )
69           ptr[ i ] = right.ptr[ i ];  // copy array into object
70
71     } // end if
72
73     return *this;   // enables x = y = z;
74
75  } // end function operator=
76
77  // determine if two arrays are equal and
78  // return true, otherwise return false
79  bool Array::operator==( const Array &right ) const
80  {
81     if ( size != right.size )
82        return false;    // arrays of different sizes
83
84     for ( int i = 0; i < size; i++ )
85
86        if ( ptr[ i ] != right.ptr[ i ] )
87           return false; // arrays are not equal
88
89     return true;        // arrays are equal
90
91  } // end function operator==
92
```

Fig. L 11.2 | `Array.cpp`. (Part 2 of 4.)

Prelab Activities

Name:

Programming Output

```
93  // overloaded subscript operator for non-const Arrays
94  // reference return creates an lvalue
95  int &Array::operator[]( int subscript )
96  {
97     // check for subscript out of range error
98     if ( subscript < 0 || subscript >= size ) {
99        cout << "\nError: Subscript " << subscript
100            << " out of range" << endl;
101
102       exit( 1 ); // terminate program; subscript out of range
103
104    } // end if
105
106    return ptr[ subscript ]; // reference return
107
108 } // end function operator[]
109
110 // overloaded subscript operator for const Arrays
111 // const reference return creates an rvalue
112 const int &Array::operator[]( int subscript ) const
113 {
114    // check for subscript out of range error
115    if ( subscript < 0 || subscript >= size ) {
116       cout << "\nError: Subscript " << subscript
117            << " out of range" << endl;
118
119       exit( 1 ); // terminate program; subscript out of range
120
121    } // end if
122
123    return ptr[ subscript ]; // const reference return
124
125 } // end function operator[]
126
127 // return number of Array objects instantiated
128 // static functions cannot be const
129 int Array::getArrayCount()
130 {
131    return arrayCount;
132
133 } // end function getArrayCount
134
135 // overloaded input operator for class Array;
136 // inputs values for entire array
137 istream &operator>>( istream &input, Array &a )
138 {
139    for ( int i = 0; i < a.size; i++ )
140       input >> a.ptr[ i ];
141
142    return input;   // enables cin >> x >> y;
143
144 } // end function operator>>
145
146 // overloaded output operator for class Array
147 ostream &operator<<( ostream &output, const Array &a )
148 {
```

Fig. L 11.2 | Array.cpp. (Part 3 of 4.)

Prelab Activities

Name:

Programming Output

```
149    int i;
150
151    for ( i = 0; i < a.size; i++ ) {
152       output << setw( 12 ) << a.ptr[ i ];
153
154       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
155          output << endl;
156
157    } // end for
158
159    if ( i % 4 != 0 )
160       output << endl;
161
162    return output;
163
164 } // end function operator<<
```

Fig. L 11.2 | Array.cpp. (Part 4 of 4.)

28. What is output by the following code? Use the definition of class Array provided in Fig. L 11.1–Fig. L 11.2.

```
1  #include "Array.h"
2
3  int main()
4  {
5     cout << "# of arrays instantiated = "
6          << Array::getArrayCount() << '\n';
7
8     Array integers1( 4 );
9     Array integers2;
10
11    cout << "# of arrays instantiated = "
12         << Array::getArrayCount() << "\n";
13
14    Array integers3( 8 ), *intptr = &integers2;
15
16    cout << "# of arrays instantiated = "
17         << Array::getArrayCount() << "\n\n";
18 } // end main
```

Your answer:

Prelab Activities

Name:

Programming Output

29. What is the output of the following program? Use the `Array` class shown in Fig. L 11.1–Fig. L 11.2.

```
#include "Array.h"

int main()
{
   Array integers1( 4 );
   Array integers2( 4 );

   if ( integers1 != integers2 )
      cout << "The arrays are different";
   else
      cout << "The arrays are identical" << endl;
} // end main
```

Your answer:

30. What is the output of the following program? Use class `Date` (Fig. 11.12–Fig. 11.13).

```
#include "Date.h"

int main()
{
   Date d1;
   Date d2( 1, 1, 1984 );
   Date d3( 8, 12, 1981 );

   cout << "d1 is " << d1
        << "\nd2 is " << d2
        << "\nd3 is " << d3 << "\n\n";

   cout << "d2 += 7 is " << ( d2 += 7 ) << "\n\n";
   cout << "d3++ is " << d3++ << "\n\n";
   cout << "d3 now is " << d3 << "\n\n";
   cout << "++d1 is " << ++d1 << "\n";
} // end main
```

Your answer:

Prelab Activities

Name:

Correct the Code

Name: _____ **Date:** _____

Section: _____

For each of the given program segments, determine if there is an error in the code. If there is an error, specify whether it is a logic or compilation error, circle the error in the program, and write the corrected code in the space provided after each problem. If the code does not contain an error, write "no error." [*Note:* It is possible that a program segment may contain multiple errors.]

31. The following code is part of a header file for class `PhoneNumber`. It overloads the `@` operator to perform stream insertion.

```
1  class PhoneNumber
2  {
3     friend ostream &operator@( ostream &, const PhoneNumber & );
```

Your answer:

32. The following code is part of a program that uses the class `Complex`. [*Note:* To view the class definition and member functions for `Complex`, see Fig. 11.19 and Fig. 11.20.

```
1  Complex x, y( 5.2, 9.1 );
2
3  x += y;
4  cout << "x is: ";
5  x.print();
```

Your answer:

Prelab Activities

Name:

Correct the Code

33. The following code is the prototype for the copy constructor for class `Sample`:

```
Sample( const Sample );
```

Your answer:

Lab Exercises

Lab Exercise 1 — String Concatenation

Name: _____ **Date:** _____

Section: _____

This problem is intended to be solved in a closed-lab session with a teaching assistant or instructor present. The problem is divided into five parts:

1. Lab Objectives
2. Description of the Problem
3. Sample Output
4. Program Template (Fig. L 11.3–Fig. L 11.5)
5. Problem-Solving Tips

The program template represents a complete working C++ program, with one or more key lines of code replaced with comments. Read the problem description and examine the sample output; then study the template code. Using the problem-solving tips as a guide, replace the `/* */` comments with C++ code. Compile and execute the program. Compare your output with the sample output provided. The source code for the template is available from the Companion Website for *C++ How to Program, Seventh Edition* at `www.pearsonhighered.com/deitel/`.

Lab Objectives

This lab was designed to reinforce programming concepts from Chapter 11 of *C++ How To Program, Seventh Edition*. In this lab, you will practice:

- Overloading the + operator to allow `String` objects to be concatenated.
- Writing function prototypes for overloaded operators.
- Using overloaded operators.
- The String case study and its code are available online at

```
www.deitel.com/bookresources/cpphtp7/StringCaseStudy.pdf
www.deitel.com/bookresources/cpphtp7/StringCaseStudyCode.zip
```

Description of the Problem

String concatenation requires two operands—the two strings that are to be concatenated. In the `String` case study, we showed how to implement an overloaded concatenation operator that concatenates the second `String` object to the right of the first `String` object, thus modifying the first `String` object. In some applications, it is desirable to produce a concatenated `String` object without modifying the `String` arguments. Implement `operator+` to allow operations such as

```
string1 = string2 + string3;
```

in which neither operand is modified.

Sample Output

```
string1 = string2 + string3
"The date is August 1, 1993" = "The date is" + " August 1, 1993"
```

Lab Exercises

Name:

Lab Exercise 1 — String Concatenation

Template

```
// Lab 1: String.h
// Header file for class String.
#ifndef STRING_H
#define STRING_H

#include <iostream>
#include <cstring>
#include <cassert>
using namespace std;

class String
{
   friend ostream &operator<<( ostream &output, const String &s );
public:
   String( const char * const = "" ); // conversion constructor
   String( const String & ); // copy constructor
   ~String(); // destructor
   const String &operator=( const String & );
   /* Write a prototype for the operator+ member function */
private:
   char *sPtr; // pointer to start of string
   int length; // string length
}; // end class String

#endif
```

Fig. L 11.3 | Contents of `String.h`.

```
// Lab 1: String.cpp
// Member-function definitions for String.cpp
#include <iostream>
#include <cstring> // strcpy and strcat prototypes
using namespace std;

#include "String.h" // String class definition

// conversion constructor: convert a char * to String
String::String( const char * const zPtr )
{
   length = strlen( zPtr ); // compute length
   sPtr = new char[ length + 1 ]; // allocate storage
   assert( sPtr != 0 ); // terminate if memory not allocated
   strcpy( sPtr, zPtr ); // copy literal to object
} // end String conversion constructor

// copy constructor
String::String( const String &copy )
{
   length = copy.length; // copy length
   sPtr = new char[ length + 1 ]; // allocate storage
   assert( sPtr != 0 ); // ensure memory allocated
   strcpy( sPtr, copy.sPtr ); // copy string
} // end String copy constructor
```

Fig. L 11.4 | Contents of `String.cpp` (Part 1 of 2.)

Lab Exercises

Name:

Lab Exercise 1 — String Concatenation

```
26
27  // destructor
28  String::~String()
29  {
30     delete [] sPtr; // reclaim string
31  } // end destructor
32
33  // overloaded = operator; avoids self assignment
34  const String &String::operator=( const String &right )
35  {
36     if ( &right != this ) // avoid self assignment
37     {
38        delete [] sPtr; // prevents memory leak
39        length = right.length; // new String length
40        sPtr = new char[ length + 1 ]; // allocate memory
41        assert( sPtr != 0 ); // ensure memory allocated
42        strcpy( sPtr, right.sPtr ); // copy string
43     }
44     else
45        cout << "Attempted assignment of a String to itself\n";
46
47     return *this; // enables concatenated assignments
48  } // end function operator=
49
50  // concatenate right operand and this object and store in temp object
51  /* Write the header for the operator+ member function */
52  {
53     /* Declare a temporary String variable named temp */
54
55     /* Set temp's length to be the sum of the two argument Strings' lengthes */
56     /* Allocate memory for temp.length + 1 chars and assign the pointer to temp.sPtr */
57     assert( sPtr != 0 ); // terminate if memory not allocated
58     /* Copy the left String argument's contents into temp.sPtr */
59     /* Write a call to strcat to concatenate the string in right
60        onto the end of the string in temp */
61     /* Return the temporary String */
62  } // end function operator+
63
64  // overloaded output operator
65  ostream & operator<<( ostream &output, const String &s )
66  {
67     output << s.sPtr;
68     return output; // enables concatenation
69  } // end function operator<<
```

Fig. L 11.4 | Contents of `String.cpp`. (Part 2 of 2.)

```
1  // Lab 1: StringCat.cpp
2  // Demonstrating overloaded + operator that does not modify operands
3  #include <iostream>
4  using namespace std;
5
6  #include "String.h"
7
8  int main()
9  {
```

Fig. L 11.5 | Contents of `StringCat.cpp`. (Part 1 of 2.)

Lab Exercises

Name:

Lab Exercise 1 — String Concatenation

```
10     String string1, string2( "The date is" );
11     String string3( " August 1, 1993" );
12
13     // test overloaded operators
14     cout << "string1 = string2 + string3\n";
15     /* Write a statement to concatenate string2 and string3,
16        and assign the result to string1 */
17     cout << "\"" << string1 << "\" = \"" << string2 << "\" + \""
18        << string3 << "\"" << endl;
19  } // end main
```

Fig. L 11.5 | Contents of `StringCat.cpp`. (Part 2 of 2.)

Problem-Solving Tips

1. The overloaded + operator should be a member function of class `String` and should take one parameter, a `const` reference to a `String`.
2. The + operator function should use return type `String`.
3. The `strcat` function can be used to concatenate pointer-based strings.

Lab Exercises

Name:

Lab Exercise 2 — Huge Integer

Name: _____ **Date:** _____

Section: _____

This problem is intended to be solved in a closed-lab session with a teaching assistant or instructor present. The problem is divided into six parts:

1. Lab Objectives
2. Description of the Problem
3. Sample Output
4. Program Template (Fig. L 11.6–Fig. L 11.8)
5. Problem-Solving Tip
6. Follow-Up Questions and Activities

The program template represents a complete working C++ program, with one or more key lines of code replaced with comments. Read the problem description and examine the sample output; then study the template code. Using the problem-solving tip as a guide, replace the `/* */` comments with C++ code. Compile and execute the program. Compare your output with the sample output provided. Then answer the follow-up questions. The source code for the template is available from the Companion Website for *C++ How to Program, Seventh Edition* at `www.pearsonhighered.com/deitel/`.

Lab Objectives

This lab was designed to reinforce programming concepts from Chapter 11 of *C++ How To Program, Seventh Edition*. In this lab, you will practice:

- Overloading arithmetic and comparison operators to enhance a huge integer class, `HugeInt`.
- Writing function prototypes for overloaded operators.
- Calling overloaded operator functions.

The follow-up questions and activities also will give you practice:

- Analyzing algorithms.

Description of the Problem

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. 11.8–11.10. Study the class carefully, then overload the relational and equality operators. [*Note:* We do not show an assignment operator or copy constructor for class `HugeInt`, because the assignment operator and copy constructor provided by the compiler are capable of copying the entire array data member properly.]

Lab Exercises

Name:

Lab Exercise 2 — Huge Integer

Sample Output

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

n1 is not equal to n2
n1 is less than n2
n1 is less than or equal to n2
7654321 + 7891234 = 15545555

99999999999999999999999999999 + 1
= 100000000000000000000000000000

7654321 + 9 = 7654330
7891234 + 10000 = 7901234
12341234 - 7888 = 12333346
```

Template

```
// Lab 2: Hugeint.h
// HugeInt class definition.
#ifndef HUGEINT_H
#define HUGEINT_H

#include <iostream>
using namespace std;

class HugeInt
{
   friend ostream &operator<<( ostream &, const HugeInt & );
public:
   HugeInt( long = 0 ); // conversion/default constructor
   HugeInt( const char * ); // 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 char * ) const;

   /* Write prototypes for the six relational and equality operators */

   int getLength() const;
private:
   short integer[ 30 ];
}; // end class HugeInt

#endif
```

Fig. L 11.6 | Contents of Hugeint.h.

Lab Exercises

Name:

Lab Exercise 2 — Huge Integer

```
1  // Lab 2: Hugeint.cpp
2  // HugeInt member-function and friend-function definitions.
3  #include <iostream>
4  #include <cctype> // isdigit function prototype
5  #include <cstring> // strlen function prototype
6  using namespace std;
7
8  #include "Hugeint.h" // HugeInt class definition
9
10 // default constructor; conversion constructor that converts
11 // a long integer into a HugeInt object
12 HugeInt::HugeInt( long value )
13 {
14    // initialize array to zero
15    for ( int i = 0; i <= 29; i++ )
16       integer[ i ] = 0;
17
18    // place digits of argument into array
19    for ( int j = 29; value != 0 && j >= 0; j-- )
20    {
21       integer[ j ] = value % 10;
22       value /= 10;
23    } // end for
24 } // end HugeInt default/conversion constructor
25
26 // conversion constructor that converts a character string
27 // representing a large integer into a HugeInt object
28 HugeInt::HugeInt( const char *string )
29 {
30    // initialize array to zero
31    for ( int i = 0; i <= 29; i++ )
32       integer[ i ] = 0;
33
34    // place digits of argument into array
35    int length = strlen( string );
36
37    for ( int j = 30 - length, k = 0; j <= 29; j++, k++ )
38
39       if ( isdigit( string[ k ] ) )
40          integer[ j ] = string[ k ] - '0';
41
42 } // end HugeInt conversion constructor
43
44 // get function calculates length of integer
45 int HugeInt::getLength() const
46 {
47    int i;
48
49    for ( i = 0; i <= 29; i++ )
50       if ( integer[ i ] != 0 )
51          break; // break when first digit is reached
52
53    return 30 - i; // length is from first digit (at i) to end of array
54 } // end function getLength
55
```

Fig. L 11.7 | Contents of `HugeInt.cpp`. (Part 1 of 3.)

Lab Exercises

Name:

Lab Exercise 2 — Huge Integer

```
56  // addition operator; HugeInt + HugeInt
57  HugeInt HugeInt::operator+( const HugeInt &op2 ) const
58  {
59     HugeInt temp; // temporary result
60     int carry = 0;
61
62     for ( int i = 29; i >= 0; i-- )
63     {
64        temp.integer[ i ] =
65           integer[ i ] + op2.integer[ i ] + carry;
66
67        // determine whether to carry a 1
68        if ( temp.integer[ i ] > 9 )
69        {
70           temp.integer[ i ] %= 10;  // reduce to 0-9
71           carry = 1;
72        } // end if
73        else // no carry
74           carry = 0;
75     } // end for
76
77     return temp; // return copy of temporary object
78  } // end function operator+
79
80  // addition operator; HugeInt + int
81  HugeInt HugeInt::operator+( int op2 ) const
82  {
83     // convert op2 to a HugeInt, then invoke
84     // operator+ for two HugeInt objects
85     return *this + HugeInt( op2 );
86  } // end function operator+
87
88  // addition operator;
89  // HugeInt + string that represents large integer value
90  HugeInt HugeInt::operator+( const char *op2 ) const
91  {
92     // convert op2 to a HugeInt, then invoke
93     // operator+ for two HugeInt objects
94     return *this + HugeInt( op2 );
95  } // end function operator+
96
97  // equality operator; HugeInt == HugeInt
98  /* Write a definition for the == operator */
99
100 // inequality operator; HugeInt != HugeInt
101 /* Write a definition for the != operator
102    by calling the == operator */
103
104 // less than operator; HugeInt < HugeInt
105 /* Write a definition for the < operator */
106
107 // less than or equal operator; HugeInt <= HugeInt
108 /* Write a definition for the <= operator
109    by calling the < and == operators */
110
111 // greater than operator; HugeInt > HugeInt
112 /* Write a definition for the > operator
```

Fig. L 11.7 | Contents of HugeInt.cpp. (Part 2 of 3.)

Lab Exercises

Name:

Lab Exercise 2 — Huge Integer

```
113    by calling the <= operator */
114
115 // greater than or equal operator; HugeInt >= HugeInt
116 /* Write a definition for the >= operator
117    by calling the > and == operators */
118
119 // overloaded output operator
120 ostream& operator<<( ostream &output, const HugeInt &num )
121 {
122    int i;
123
124    for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i++ )
125       ; // skip leading zeros
126
127    if ( i == 30 )
128       output << 0;
129    else
130
131       for ( ; i <= 29; i++ )
132          output << num.integer[ i ];
133
134    return output;
135 } // end function operator<<
```

Fig. L 11.7 | Contents of `HugeInt.cpp`. (Part 3 of 3.)

```
1  // Lab 2: HugeIntTest.cpp
2  // HugeInt test program.
3  #include <iostream>
4  using namespace std;
5
6  #include "Hugeint.h"
7
8  int main()
9  {
10    HugeInt n1( 7654321 );
11    HugeInt n2( 7891234 );
12    HugeInt n3( "99999999999999999999999999999" );
13    HugeInt n4( "1" );
14    HugeInt result;
15
16    cout << "n1 is " << n1 << "\nn2 is " << n2
17       << "\nn3 is " << n3 << "\nn4 is " << n4
18       << "\nresult is " << result << "\n\n";
19
20    // test relational and equality operators
21    if ( n1 == n2 )
22       cout << "n1 equals n2" << endl;
23
24    if ( n1 != n2 )
25       cout << "n1 is not equal to n2" << endl;
26
27    if ( n1 < n2 )
28       cout << "n1 is less than n2" << endl;
29
```

Fig. L 11.8 | Contents of `HugeIntTest.cpp`. (Part 1 of 2.)

Lab Exercises

Name:

Lab Exercise 2 — Huge Integer

```
30     if ( n1 <= n2 )
31        cout << "n1 is less than or equal to n2" << endl;
32
33     if ( n1 > n2 )
34        cout << "n1 is greater than n2" << endl;
35
36     if ( n1 >= n2 )
37        cout << "n1 is greater than or equal to n2" << endl;
38
39     result = n1 + n2;
40     cout << n1 << " + " << n2 << " = " << result << "\n\n";
41
42     cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 ) << "\n\n";
43
44     result = n1 + 9;
45     cout << n1 << " + " << 9 << " = " << result << endl;
46
47     result = n2 + "10000";
48     cout << n2 << " + " << "10000" << " = " << result << endl;
49  } // end main
```

Fig. L 11.8 | Contents of `HugeIntTest.cpp`. (Part 2 of 2.)

Problem-Solving Tip

1. You can implement the `!=`, `>`, `>=` and `<=` operators in terms of the overloaded `==` and `<` operators.

Follow-Up Questions and Activities

1. Describe precisely how the overloaded addition operator for `HugeInt` operates.

2. What restrictions does the class have?

Lab Exercises

Name:

Lab Exercise 3 — Rational Numbers

Name: _____ **Date:** _____

Section: _____

This problem is intended to be solved in a closed-lab session with a teaching assistant or instructor present. The problem is divided into six parts:

1. Lab Objectives
2. Description of the Problem
3. Sample Output
4. Program Template (Fig. L 11.9–Fig. L 11.11)
5. Problem-Solving Tips
6. Follow-Up Questions and Activities

The program template represents a complete working C++ program, with one or more key lines of code replaced with comments. Read the problem description and examine the sample output; then study the template code. Using the problem-solving tips as a guide, replace the `/* */` comments with C++ code. Compile and execute the program. Compare your output with the sample output provided. Then answer the follow-up questions. The source code for the template is available from the Companion Website for *C++ How to Program, Seventh Edition* at `www.pearsonhighered.com/deitel/`.

Lab Objectives

This lab was designed to reinforce programming concepts from Chapter 11 of *C++ How To Program, Seventh Edition*. In this lab, you will practice:

- Overloading operators to create a class capable of storing rational numbers (fractions) and performing rational number arithmetic.
- Writing function prototypes for overloaded operators.
- Implementing overloaded operator functions.

The follow-up questions and activities also will give you practice:

- Overloading the << operator.
- Making a class more robust to prevent runtime errors.

Description of the Problem

Create a class `RationalNumber` (fractions) with the following capabilities:

a) Create a constructor that prevents a `0` denominator in a fraction, reduces or simplifies fractions that are not in reduced form and avoids negative denominators.

b) Overload the addition, subtraction, multiplication and division operators for this class.

c) Overload the relational and equality operators for this class.

Lab Exercises

Name:

Lab Exercise 3 — Rational Numbers

Sample Output

```
7/3 + 1/3 = 8/3
7/3 - 1/3 = 2
7/3 * 1/3 = 7/9
7/3 / 1/3 = 7
7/3 is:
  > 1/3 according to the overloaded > operator
  >= 1/3 according to the overloaded < operator
  >= 1/3 according to the overloaded >= operator
  > 1/3 according to the overloaded <= operator
  != 1/3 according to the overloaded == operator
  != 1/3 according to the overloaded != operator
```

Template

```
1  // Lab 3: RationalNumber.h
2  // RationalNumber class definition.
3  #ifndef RATIONAL_NUMBER_H
4  #define RATIONAL_NUMBER_H
5
6  class RationalNumber
7  {
8  public:
9     RationalNumber( int = 0, int = 1 ); // default constructor
10    /* Write prototype for operator + */
11    /* Write prototype for operator - */
12    /* Write prototype for operator * */
13    /* Write prototype for operator / */
14
15    // relational operators
16    /* Write prototype for operator > */
17    /* Write prototype for operator < */
18    /* Write prototype for operator >= */
19    /* Write prototype for operator <= */
20
21    // equality operators
22    /* Write prototype for operator == */
23    /* Write prototype for operator != */
24
25    void printRational() const; // display rational number
26 private:
27    int numerator; // private variable numerator
28    int denominator; // private variable denominator
29    void reduction(); // function for fraction reduction
30 }; // end class RationalNumber
31
32 #endif
```

Fig. L 11.9 | RationalNumber.h.

Lab Exercises

Name:

Lab Exercise 3 — Rational Numbers

```
1  // Lab 3: RationalNumber.cpp
2  // RationalNumber member-function definitions.
3  #include <cstdlib>
4  #include <iostream>
5  using namespace std;
6
7  #include "RationalNumber.h"
8
9  // RationalNumber constructor sets n and d and calls reduction
10 /* Implement the RationalNumber constructor. Validate d first to ensure that
11    it is a positive number and set it to 1 if not. Call the reduction utility
12    function at the end */
13
14 // overloaded + operator
15 /* Write definition for overloaded operator + */
16
17 // overloaded - operator
18 /* Write definition for overloaded operator - */
19
20 // overloaded * operator
21 /* Write definition for overloaded operator * */
22
23 // overloaded / operator
24 /* Write definition for overloaded operator /. Check if the client is
25    attempting to divide by zero and report an error message if so */
26
27 // overloaded > operator
28 /* Write definition for operator > */
29
30 // overloaded < operator
31 /* Write definition for operator < */
32
33 // overloaded >= operator
34 /* Write definition for operator >= */
35
36 // overloaded <= operator
37 /* Write definition for operator <= */
38
39 // overloaded == operator
40 /* Write definition for operator == */
41
42 // overloaded != operator
43 /* Write definition for operator != */
44
45 // function printRational definition
46 void RationalNumber::printRational() const
47 {
48    if ( numerator == 0 ) // print fraction as zero
49       cout << numerator;
50    else if ( denominator == 1 ) // print fraction as integer
51       cout << numerator;
52    else
53       cout << numerator << '/' << denominator;
54 } // end function printRational
55
```

Fig. L 11.10 | `RationalNumber.cpp`. (Part 1 of 2.)

Lab Exercises

Name:

Lab Exercise 3 — Rational Numbers

```
56  // function reduction definition
57  void RationalNumber::reduction()
58  {
59     int largest, gcd = 1;  // greatest common divisor;
60
61     largest = ( numerator > denominator ) ? numerator: denominator;
62
63     for ( int loop = 2; loop <= largest; loop++ )
64        if ( numerator % loop == 0 && denominator % loop == 0 )
65           gcd = loop;
66
67     numerator /= gcd;
68     denominator /= gcd;
69  } // end function reduction
```

Fig. L 11.10 | RationalNumber.cpp. (Part 2 of 2.)

```
1   // Lab 3: RationalTest.cpp
2   // RationalNumber test program.
3   #include <iostream>
4   using namespace std;
5
6   #include "RationalNumber.h"
7
8   int main()
9   {
10     RationalNumber c( 7, 3 ), d( 3, 9 ), x;
11
12     c.printRational();
13     cout << " + ";
14     d.printRational();
15     cout << " = ";
16     x = c + d; // test overloaded operators + and =
17     x.printRational();
18
19     cout << '\n';
20     c.printRational();
21     cout << " - ";
22     d.printRational();
23     cout << " = ";
24     x = c - d; // test overloaded operators - and =
25     x.printRational();
26
27     cout << '\n';
28     c.printRational();
29     cout << " * ";
30     d.printRational();
31     cout << " = ";
32     x = c * d; // test overloaded operators * and =
33     x.printRational();
34
35     cout << '\n';
36     c.printRational();
37     cout << " / ";
38     d.printRational();
```

Fig. L 11.11 | RationalTest.cpp (Part 1 of 2.)

Lab Exercises

Name:

Lab Exercise 3 — Rational Numbers

```
39      cout << " = ";
40      x = c / d; // test overloaded operators / and =
41      x.printRational();
42
43      cout << '\n';
44      c.printRational();
45      cout << " is:\n";
46
47      // test overloaded greater than operator
48      cout << ( ( c > d ) ? "  > " : "  <= " );
49      d.printRational();
50      cout << " according to the overloaded > operator\n";
51
52      // test overloaded less than operator
53      cout << ( ( c < d ) ? "  < " : "  >= " );
54      d.printRational();
55      cout << " according to the overloaded < operator\n";
56
57      // test overloaded greater than or equal to operator
58      cout << ( ( c >= d ) ? "  >= " : "  < " );
59      d.printRational();
60      cout << " according to the overloaded >= operator\n";
61
62      // test overloaded less than or equal to operator
63      cout << ( ( c <= d ) ? "  <= " : "  > " );
64      d.printRational();
65      cout << " according to the overloaded <= operator\n";
66
67      // test overloaded equality operator
68      cout << ( ( c == d ) ? "  == " : "  != " );
69      d.printRational();
70      cout << " according to the overloaded == operator\n";
71
72      // test overloaded inequality operator
73      cout << ( ( c != d ) ? "  != " : "  == " );
74      d.printRational();
75      cout << " according to the overloaded != operator" << endl;
76  } // end main
```

Fig. L 11.11 | `RationalTest.cpp`. (Part 2 of 2.)

Problem-Solving Tips

1. When comparing `RationalNumbers`, you can cast the `numerator` to a `double` and then divide by the `denominator` to determine the value of that `RationalNumber` as a `double`. The `<=`, `>=`, `>` and `!=` operators can be implemented in terms of `==` and `<`.
2. To implement the arithmetic operators, use the following formulas:

   Addition: *(a/b) + (c/d) = (ad + bc) / (bd).*
   Subtraction: *(a/b) – (c/d) = (ad – bc) / (bd).*
   Multiplication: *(a/b) * (c/d) = (ac) / (bd).*
   Division: *(a/b) / (c/d) = (ad) / (bc).*

   Remember to check for division by zero.

Lab Exercises

Name:

Lab Exercise 3 — Rational Numbers

Follow-Up Questions and Activities

1. Rewrite the `printRational` member function as an overloaded `<<` `friend` function.

2. Make the `RationalNumber` class more robust by providing additional tests for division by zero in each of the relational operators that divides a `numerator` by a `denominator`.

3. Is it possible to add another overloaded `operator>` function that returns a pointer to the larger of the two rational numbers? Why or why not?

Lab Exercises

Name:

Debugging

Name: _____ **Date:** _____

Section: _____

The program (Fig. L 11.12–Fig. L 11.14) in this section does not run properly. Fix all the compilation errors so that the program will compile successfully. Once the program compiles, compare the output with the sample output, and eliminate any logic errors that may exist. The sample output demonstrates what the program's output should be once the program's code has been corrected.

Sample Output

[*Note:* There may be rounding errors due to the conversion from a floating-point number to an integer.]

```
Initial values:
0
0
1.23

Enter a number: 2.345
Enter a number: 3.456
The sum of test1 and test2 is: 5.80

final values:
test1 = 3.34
test2 = 4.45
test3 = 8.03
test1 and test3 are not equal to each other
```

```
Initial values:
0
0
1.23

Enter a number: 0
Enter a number: -2.234
The sum of test1 and test2 is: -2.24

final values:
test1 = 1
test2 = -1.24
test3 = 0
```

Lab Exercises

Name:

Debugging

Broken Code

```
// Debugging: Decimal.h

#ifndef DECIMAL_H
#define DECIMAL_H

#include <iostream>
using namespace std;

// class Decimal definition
class Decimal
{
public:
   friend istream operator>>( istream &, const Decimal & );
   Decimal( double = 0.0 );

   void setInteger( double );
   void setDecimal( double );

   Decimal &operator=( const Decimal );
   Decimal +( Decimal );
   Decimal +=( Decimal ) const;
   Decimal &operator++();
   Decimal operator++( double );
   bool operator==( const Decimal );
private:
   friend ostream &operator<<( const Decimal & );
   double integer;
   double decimal;
}; // end class Decimal

#endif // DECIMAL_H
```

Fig. L 11.12 | Decimal.h.

```
// Debugging: Decimal.cpp

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

#include "Decimal.h"

// constructor
Decimal::Decimal( double n )
{
   decimal = modf( n, &integer );
} // end class Decimal constructor

// function operator<< definition
friend ostream & operator<<( const Decimal &d )
{
   double n = 0;

```

Fig. L 11.13 | Decimal.cpp. (Part 1 of 3.)

Lab Exercises

Name:

Debugging

```
20      n = floor( d.decimal * 100 );
21
22      if ( n < 0 )
23         n = 0 - dec;
24
25      if ( d.decimal != 0 ) {
26         output << floor( d.integer ) << ".";
27
28         if ( n > 10 )
29            output << n;
30         else
31            output << "0" << n;
32      } // end if
33      else
34         output << d.integer;
35
36   } // end function operator<<
37
38   // function operator>> definition
39   friend istream operator>>( istream &input, const Decimal &d )
40   {
41      double n;
42
43      cout << "Enter a number: ";
44      istream >> n;
45      decimal = modf( n, &integer );
46      return input;
47
48   } // end function operator>>
49
50   // function operator= definition
51   Decimal &Decimal::operator=( const Decimal d )
52   {
53      integer = d.integer;
54      decimal = d.decimal;
55
56      return *this;
57   } // end function operator=
58
59   // function setDecimal definition
60   void Decimal::setDecimal( double d )
61   {
62      decimal = d;
63   } // end function setDecimal
64
65   // function setInteger definition
66   void Decimal::setInteger( double i )
67   {
68      integer = i;
69   } // end function setInteger
70
71   // function operator+ definition
72   Decimal Decimal::operator+( Decimal d )
73   {
74      Decimal result;
75
```

Fig. L 11.13 | Decimal.cpp (Part 2 of 3.)

Lab Exercises

Name:

Debugging

```
76     result.setDecimal( decimal + d.decimal );
77     result.setInteger( integer + d.integer );
78
79     if ( result.decimal >= 1 )
80     {
81        --result.decimal;
82        ++result.integer;
83
84     } // end if
85     else if ( result.decimal <= -1 )
86     {
87        ++result.decimal;
88        --result.integer;
89     } // end if
90
91     return result;
92  } // end function operator+
93
94  // function operator+= definition
95  Decimal Decimal::operator+=( Decimal d ) const
96  {
97     *this = *this += d;
98     return *this;
99  } // end function operator+=
100
101 // function operator++ definition
102 Decimal &Decimal::operator++()
103 {
104    ++integer;
105    return integer;
106 } // end function operator++
107
108 // function operator++ definition
109 Decimal Decimal::operator++( double )
110 {
111    Decimal temp = *this;
112
113    ++integer;
114    return *this;
115 } // end function operator++
116
117 // function operator== definition
118 bool Decimal::operator==( const Decimal d )
119 {
120    return ( integer == d.integer && decimal == d.decimal );
121 } // end function operator==
```

Fig. L 11.13 | `Decimal.cpp`. (Part 3 of 3.)

Lab Exercises

Name:

Debugging

```
1  // Debugging: debugging.cpp
2
3  #include <iostream>
4  using namespace std;
5
6  #include "Decimal.h"
7
8  int main()
9  {
10    Decimal test1;
11    Decimal test2;
12    Decimal test3( 1.234 );
13
14    cout << "Initial values:\n"
15         << test1 << endl << test2 << endl << test3
16         << endl << endl;
17
18    cin >> test1 >> test2;
19
20    cout << "The sum of test1 and test2 is: "
21         << test1 + test2 << endl;
22    test3 += test1++ + ++test2;
23
24    cout << "\nfinal values:\n"
25         << "test1 = " << test1 << endl
26         << "test2 = " << test2 << endl
27         << "test3 = " << test3 << endl;
28
29    if ( test1 != test3 )
30       cout << "test1 and test3 are not equal to each other\n";
31 } // end main
```

Fig. L 11.14 | debugging.cpp.

Postlab Activities

Coding Exercises

Name: _____ **Date:** _____

Section: _____

These coding exercises reinforce the lessons learned in the lab and provide additional programming experience outside the classroom and laboratory environment. They serve as a review after you have completed the *Prelab Activities* and *Lab Exercises* successfully.

For each of the following problems, write a program or a program segment that performs the specified action. Each problem refers to class `Polygon` (Fig. L 11.15). This class contains a dynamically allocated array of *x*-coordinates and a dynamically allocated array of *y*-coordinates. These coordinates store the polygon's vertices. The `Polygon` constructor takes the initial vertex for the `Polygon`.

```
1  // Coding Exercises: Polygon.cpp
2  // class Polygon definition
3  #include <iostream>
4  using namespace std;
5
6  class Polygon
7  {
8  public:
9     Polygon( int = 0, int = 0 );
10    ~Polygon();
11
12    void addVertex( int, int );
13    int getNumVertices() const;
14 private:
15    int *xPts;
16    int *yPts;
17    int numVertices;
18 }; // end class Polygon
19
20 // default constructor
21 Polygon::Polygon( int x, int y )
22 {
23    xPts = new int[ 1 ];
24    yPts = new int[ 1 ];
25
26    xPts[ 0 ] = x;
27    yPts[ 0 ] = y;
28    numVertices = 1;
29 } // end class Polygon constructor
30
31 // destructor
32 Polygon::~Polygon()
33 {
34    delete [] xPts;
35    delete [] yPts;
36 } // end class Polygon destructor
37
38 // function addVertex definition
39 void Polygon::addVertex( int x, int y )
40 {
```

Fig. L 11.15 | `Polygon.cpp`. (Part 1 of 2.)

Postlab Activities

Name:

Coding Exercises

```
41     int *copyX = new int[ numVertices + 1 ];
42     int *copyY = new int[ numVertices + 1 ];
43
44     for ( int i = 0; i < numVertices; i++ )
45     {
46        copyX[ i ] = xPts[ i ];
47        copyY[ i ] = yPts[ i ];
48     } // end for
49
50     copyX[ numVertices ] = x;
51     copyY[ numVertices ] = y;
52
53     delete [] xPts;
54     delete [] yPts;
55
56     xPts = copyX;
57     yPts = copyY;
58     ++numVertices;
59  } // end function addVertex
60
61  // function getNumVertices
62  int Polygon::getNumVertices() const
63  {
64     return numVertices;
65  } // end function getNumVertices
```

Fig. L 11.15 | `Polygon.cpp`. (Part 2 of 2.)

1. Overload the stream-insertion operator << to output a `Polygon` object. The prototype for the << operator is as follows:

```
ostream &operator<<( ostream &, const Polygon & );
```

Postlab Activities

Name:

Coding Exercises

2. Create a copy constructor for class `Polygon`. The prototype for the copy constructor is as follows:

```
Polygon( Polygon * );
```

3. Overload the == operator to compare two `Polygons` for equality. This member function should return a `boolean` value. The prototype for the == operator is as follows:

```
bool operator==( Polygon & );
```

Postlab Activities

Name:

Coding Exercises

4. Overload the = operator to assign one `Polygon` object to another. This member function should return a `const` reference to the `Polygon` object invoking the member function. This member function also should test for self assignment. The prototype for the = operator is as follows:

```
const Polygon &operator=( const Polygon & );
```

Postlab Activities

Name:

Programming Challenges

Name: _____ **Date:** _____

Section: _____

The *Programming Challenges* are more involved than the *Coding Exercises* and may require a significant amount of time to complete. Write a C++ program for each of the problems in this section. The answers to these problems are available from the Companion Website for *C++ How to Program, Seventh Edition* at `www.pearsonhighered.com/deitel/`. Pseudocode, hints and/or sample outputs are provided to aid you in your programming.

1. Consider class `Complex` shown in Fig. 11.19–Fig. 11.20. 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 through the overloaded >> and << operators, respectively. (You should remove the `print` member function from the class.)

   b) Overload the multiplication operator to enable multiplication of two complex numbers as in algebra. Complex number multiplication is performed as follows:

   $$(a + bi) * (c + di) = (ac - bd) + (ad + bc)i$$

   c) Overload the == and != operators to allow comparisons of complex numbers.

Hints:

- When overloading the stream-extraction operator, use the `ignore` member function of the class `istream`. See Fig. 11.4 of *C++ How to Program, Seventh Edition* for an example.
- When overloading the assignment operator, return the `this` pointer to enable cascaded calls.
- The overloaded << and >> operators should be `friend` functions.
- All other overloaded operator functions should be `const` member functions.
- Sample output:

```
Enter a complex number in the form: (a, b)
? (0, 0)
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)
k: (0, 0)

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)

x = y * z:
(23.21, 31.79) = (4.3, 8.2) * (3.3, 1.1)

(23.21, 31.79) != (0, 0)

(0, 0) == (0, 0)
```

Postlab Activities

Name:

Programming Challenges

2. Develop class `Polynomial`. The internal representation of a `Polynomial` is an array of terms. Each term contains a coefficient and an exponent. 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 addition assignment operator (+=) and subtraction assignment operator (-=).

Hints:

- Use your overloaded addition, subtraction and multiplication operators to implement their respective assignment operators.
- Sample output:

```
Enter number of polynomial terms: 3

Enter coefficient: 12
Enter exponent: 1

Enter coefficient: 4
Enter exponent: 2

Enter coefficient: 6
Enter exponent: 3

Enter number of polynomial terms: 3

Enter coefficient: -5
Enter exponent: 1

Enter coefficient: 3
Enter exponent: 2

Enter coefficient: 1
Enter exponent: 3
First polynomial is:
12x+4x^2+6x^3
Second polynomial is:
-5x+3x^2+1x^3

Adding the polynomials yields:
7x+7x^2+7x^3

+= the polynomials yields:
7x+7x^2+7x^3

Subtracting the polynomials yields:
17x+1x^2+5x^3

-= the polynomials yields:
17x+1x^2+5x^3
```