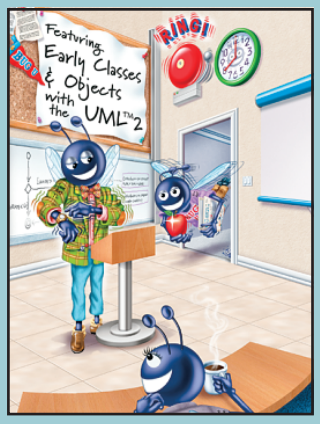
11



Operator Overloading; String and Array Objects

OBJECTIVES

In this chapter you will learn:

- What operator overloading is and how it makes programs more readable and programming more convenient.
- To redefine (overload) operators to work with objects of user-defined classes.
- The differences between overloading unary and binary operators.
- To convert objects from one class to another class.
- When to, and when not to, overload operators.
- To create `PhoneNumber`, `Array`, `String` and `Date` classes that demonstrate operator overloading.
- To use overloaded operators and other member functions of standard library class `string`.
- To use keyword `explicit` to prevent the compiler from using single-argument constructors to perform implicit conversions.

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: Fifth 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 specify how such conversions occur explicitly.

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 as concise as possible; 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
15    return 0;
16
17 } // 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
10    return 0;
11
12 } // 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
8  using std::ostream;
9  using std::istream;
10
11 // class Array definition
12 class Array
13 {
14    friend ostream &operator<<( ostream &, const Array & );
15    friend istream &operator>>( istream &, Array & );
16
17 public:
18    Array( int = 10 );                    // default constructor
19    Array( const Array & );               // copy constructor
20    ~Array();                             // destructor
21    int getSize() const;                  // return size
22    const Array &operator=( const Array & ); // assignment operator
23    bool operator==( const Array & ) const;  // equality operator
24
```

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

Prelab Activities

Name:

Programming Output

```
25     // determine if two arrays are not equal and
26     // return true, otherwise return false (uses operator==)
27     bool operator!=( const Array &right ) const
28     {
29        return ! ( *this == right );
30
31     } // end function operator!=
32
33     int &operator[]( int );              // subscript operator
34     const int &operator[]( int ) const;  // subscript operator
35     static int getArrayCount();          // return number of
36                                          // arrays instantiated
37  private:
38     int size; // size of array
39     int *ptr; // pointer to first element of array
40     static int arrayCount;  // number of Arrays instantiated
41
42  }; // end class Array
43
44  #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
5   using std::cout;
6   using std::cin;
7   using std::endl;
8
9   #include <iomanip>
10
11  using std::setw;
12
13  #include <cstdlib>
14
15  #include <new>
16
17  #include "Array.h"
18
19  // initialize static data member at file scope
20  int Array::arrayCount = 0;   // no objects yet
21
22  // default constructor for class Array (default size 10)
23  Array::Array( int arraySize )
24  {
25     size = ( arraySize > 0 ? arraySize : 10 );
26     ptr = new int[ size ]; // create space for array
27     ++arrayCount;          // count one more object
28
29     for ( int i = 0; i < size; i++ )
30        ptr[ i ] = 0;          // initialize array
31
32  } // end class Array constructor
```

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

Prelab Activities

Name:

Programming Output

```
33
34  // copy constructor for class Array
35  // must receive reference to prevent infinite recursion
36  Array::Array( const Array &arrayToCopy ) : size( arrayToCopy.size )
37  {
38     ptr = new int[ size ]; // create space for array
39     ++arrayCount;          // count one more object
40
41     for ( int i = 0; i < size; i++ )
42        ptr[ i ] = arrayToCopy.ptr[ i ];  // copy arayToCopy into object
43
44  } // end copy constructor
45
46  // destructor for class Array
47  Array::~Array()
48  {
49     delete [] ptr;            // reclaim space for array
50     --arrayCount;             // one fewer object
51
52  } // end class Array destructor
53
54  // get size of array
55  int Array::getSize() const
56  {
57     return size;
58
59  } // end function getSize
60
61  // overloaded assignment operator
62  // const return avoids: ( a1 = a2 ) = a3
63  const Array &Array::operator=( const Array &right )
64  {
65     if ( &right != this ) {  // check for self-assignment
66
67        // for arrays of different sizes, deallocate original
68        // left side array, then allocate new left side array
69        if ( size != right.size ) {
70           delete [] ptr;         // reclaim space
71           size = right.size;     // resize this object
72           ptr = new int[ size ]; // create space for array copy
73
74        } // end if
75
76        for ( int i = 0; i < size; i++ )
77           ptr[ i ] = right.ptr[ i ];  // copy array into object
78
79     } // end if
80
81     return *this;   // enables x = y = z;
82
83  } // end function operator=
84
85  // determine if two arrays are equal and
86  // return true, otherwise return false
87  bool Array::operator==( const Array &right ) const
88  {
89     if ( size != right.size )
```

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

Prelab Activities

Name:

Programming Output

```
90          return false;    // arrays of different sizes
91
92       for ( int i = 0; i < size; i++ )
93
94          if ( ptr[ i ] != right.ptr[ i ] )
95             return false; // arrays are not equal
96
97       return true;        // arrays are equal
98
99    } // end function operator==
100
101   // overloaded subscript operator for non-const Arrays
102   // reference return creates an lvalue
103   int &Array::operator[]( int subscript )
104   {
105      // check for subscript out of range error
106      if ( subscript < 0 || subscript >= size ) {
107         cout << "\nError: Subscript " << subscript
108              << " out of range" << endl;
109
110         exit( 1 ); // terminate program; subscript out of range
111
112      } // end if
113
114      return ptr[ subscript ]; // reference return
115
116   } // end function operator[]
117
118   // overloaded subscript operator for const Arrays
119   // const reference return creates an rvalue
120   const int &Array::operator[]( int subscript ) const
121   {
122      // check for subscript out of range error
123      if ( subscript < 0 || subscript >= size ) {
124         cout << "\nError: Subscript " << subscript
125              << " out of range" << endl;
126
127         exit( 1 ); // terminate program; subscript out of range
128
129      } // end if
130
131      return ptr[ subscript ]; // const reference return
132
133   } // end function operator[]
134
135   // return number of Array objects instantiated
136   // static functions cannot be const
137   int Array::getArrayCount()
138   {
139      return arrayCount;
140
141   } // end function getArrayCount
142
143   // overloaded input operator for class Array;
144   // inputs values for entire array
145   istream &operator>>( istream &input, Array &a )
146   {
```

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

Prelab Activities

Name:

Programming Output

```
147    for ( int i = 0; i < a.size; i++ )
148       input >> a.ptr[ i ];
149
150    return input;   // enables cin >> x >> y;
151
152 } // end function operator>>
153
154 // overloaded output operator for class Array
155 ostream &operator<<( ostream &output, const Array &a )
156 {
157    int i;
158
159    for ( i = 0; i < a.size; i++ ) {
160       output << setw( 12 ) << a.ptr[ i ];
161
162       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
163          output << endl;
164
165    } // end for
166
167    if ( i % 4 != 0 )
168       output << endl;
169
170    return output;
171
172 } // 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
19    return 0;
20
21 } // end main
```

Prelab Activities

Name:

Programming Output

Your answer:

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 << "Hello";
   else
      cout << "Goodbye" << endl;

   return 0;
} // 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";
   return 0;
} // end main
```

Prelab Activities

Name:

Programming Output

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, syntax 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 at `www.deitel.com` and `www.prenhall.com./deitel`.

Lab Objectives

This lab was designed to reinforce programming concepts from Chapter 11 of *C++ How To Program: Fifth 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.

Description of the Problem

String concatenation requires two operands—the two strings that are to be concatenated. In the text, 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

```
1  // Lab 1: String.h
2  // Header file for class String.
```

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

```
3  #ifndef STRING_H
4  #define STRING_H
5
6  #include <iostream>
7  using std::cout;
8  using std::ostream;
9
10 #include <cstring>
11 #include <cassert>
12
13 class String
14 {
15    friend ostream &operator<<( ostream &output, const String &s );
16 public:
17    String( const char * const = "" ); // conversion constructor
18    String( const String & ); // copy constructor
19    ~String(); // destructor
20    const String &operator=( const String & );
21    /* Write a prototype for the operator+ member function */
22 private:
23    char *sPtr; // pointer to start of string
24    int length; // string length
25 }; // end class String
26
27 #endif
```

```
1  // Lab 1: String.cpp
2  // Member-function definitions for String.cpp
3  #include <iostream>
4  using std::cout;
5  using std::ostream;
6
7  #include <cstring> // strcpy and strcat prototypes
8  #include "String.h" // String class definition
9
10 // conversion constructor: convert a char * to String
11 String::String( const char * const zPtr )
12 {
13    length = strlen( zPtr ); // compute length
14    sPtr = new char[ length + 1 ]; // allocate storage
15    assert( sPtr != 0 ); // terminate if memory not allocated
16    strcpy( sPtr, zPtr ); // copy literal to object
17 } // end String conversion constructor
18
19 // copy constructor
20 String::String( const String &copy )
21 {
22    length = copy.length; // copy length
23    sPtr = new char[ length + 1 ]; // allocate storage
```

Lab Exercises

Name:

Lab Exercise 1 — String Concatenation

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

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

Lab Exercises

Name:

Lab Exercise 1 — String Concatenation

```
6
7  #include "String.h"
8
9  int main()
10 {
11    String string1, string2( "The date is" );
12    String string3( " August 1, 1993" );
13
14    // test overloaded operators
15    cout << "string1 = string2 + string3\n";
16    /* Write a statement to concatenate string2 and string3,
17       and assign the result to string1 */
18    cout << "\"" << string1 << "\" = \"" << string2 << "\" + \""
19       << string3 << "\"" << endl;
20    return 0;
21 } // 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 at `www.deitel.com` and `www.prenhall.com./deitel`.

Lab Objectives

This lab was designed to reinforce programming concepts from Chapter 11 of *C++ How To Program: Fifth 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

999999999999999999999999999999 + 1
= 1000000000000000000000000000000

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

Template

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

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

Lab Exercises

Name:

Lab Exercise 2 — Huge Integer

```
58
59  // addition operator; HugeInt + HugeInt
60  HugeInt HugeInt::operator+( const HugeInt &op2 ) const
61  {
62     HugeInt temp; // temporary result
63     int carry = 0;
64
65     for ( int i = 29; i >= 0; i-- )
66     {
67        temp.integer[ i ] =
68           integer[ i ] + op2.integer[ i ] + carry;
69
70        // determine whether to carry a 1
71        if ( temp.integer[ i ] > 9 )
72        {
73           temp.integer[ i ] %= 10;  // reduce to 0-9
74           carry = 1;
75        } // end if
76        else // no carry
77           carry = 0;
78     } // end for
79
80     return temp; // return copy of temporary object
81  } // end function operator+
82
83  // addition operator; HugeInt + int
84  HugeInt HugeInt::operator+( int op2 ) const
85  {
86     // convert op2 to a HugeInt, then invoke
87     // operator+ for two HugeInt objects
88     return *this + HugeInt( op2 );
89  } // end function operator+
90
91  // addition operator;
92  // HugeInt + string that represents large integer value
93  HugeInt HugeInt::operator+( const char *op2 ) const
94  {
95     // convert op2 to a HugeInt, then invoke
96     // operator+ for two HugeInt objects
97     return *this + HugeInt( op2 );
98  } // end function operator+
99
100 // equality operator; HugeInt == HugeInt
101 /* Write a definition for the == operator */
102
103 // inequality operator; HugeInt != HugeInt
104 /* Write a definition for the != operator
105    by calling the == operator */
106
107 // less than operator; HugeInt < HugeInt
108 /* Write a definition for the < operator */
109
110 // less than or equal operator; HugeInt <= HugeInt
111 /* Write a definition for the <= operator
112    by calling the < and == operators */
113
```

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

Lab Exercises

Name:

Lab Exercise 2 — Huge Integer

```
114 // greater than operator; HugeInt > HugeInt
115 /* Write a definition for the > operator
116    by calling the <= operator */
117
118 // greater than or equal operator; HugeInt >= HugeInt
119 /* Write a definition for the >= operator
120    by calling the > and == operators */
121
122 // overloaded output operator
123 ostream& operator<<( ostream &output, const HugeInt &num )
124 {
125    int i;
126
127    for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i++ )
128       ; // skip leading zeros
129
130    if ( i == 30 )
131       output << 0;
132    else
133
134       for ( ; i <= 29; i++ )
135          output << num.integer[ i ];
136
137    return output;
138 } // 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 std::cout;
5  using std::endl;
6
7  #include "Hugeint.h"
8
9  int main()
10 {
11    HugeInt n1( 7654321 );
12    HugeInt n2( 7891234 );
13    HugeInt n3( "99999999999999999999999999999" );
14    HugeInt n4( "1" );
15    HugeInt result;
16
17    cout << "n1 is " << n1 << "\nn2 is " << n2
18       << "\nn3 is " << n3 << "\nn4 is " << n4
19       << "\nresult is " << result << "\n\n";
20
21    // test relational and equality operators
22    if ( n1 == n2 )
23       cout << "n1 equals n2" << endl;
24
25    if ( n1 != n2 )
26       cout << "n1 is not equal to n2" << endl;
27
```

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

Lab Exercises

Name:

Lab Exercise 2 — Huge Integer

```
28      if ( n1 < n2 )
29         cout << "n1 is less than n2" << endl;
30
31      if ( n1 <= n2 )
32         cout << "n1 is less than or equal to n2" << endl;
33
34      if ( n1 > n2 )
35         cout << "n1 is greater than n2" << endl;
36
37      if ( n1 >= n2 )
38         cout << "n1 is greater than or equal to n2" << endl;
39
40      result = n1 + n2;
41      cout << n1 << " + " << n2 << " = " << result << "\n\n";
42
43      cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 ) << "\n\n";
44
45      result = n1 + 9;
46      cout << n1 << " + " << 9 << " = " << result << endl;
47
48      result = n2 + "10000";
49      cout << n2 << " + " << "10000" << " = " << result << endl;
50
51      return 0;
52   } // 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 at `www.deitel.com` and `www.prenhall.com./deitel`.

Lab Objectives

This lab was designed to reinforce programming concepts from Chapter 11 of *C++ How To Program: Fifth 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  using std::exit;
5
6  #include <iostream>
7  using std::cout;
8  using std::endl;
9
10 #include "RationalNumber.h"
11
12 // RationalNumber constructor sets n and d and calls reduction
13 /* Implement the RationalNumber constructor. Validate d first to ensure that
14    it is a positive number and set it to 1 if not. Call the reduction utility
15    function at the end */
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 * */
25
26 // overloaded / operator
27 /* Write definition for overloaded operator /. Check if the client is
28    attempting to divide by zero and report an error message if so */
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 // overloaded != operator
46 /* Write definition for operator != */
47
48 // function printRational definition
49 void RationalNumber::printRational() const
50 {
51    if ( numerator == 0 ) // print fraction as zero
52       cout << numerator;
53    else if ( denominator == 1 ) // print fraction as integer
54       cout << numerator;
55    else
56       cout << numerator << '/' << denominator;
57 } // end function printRational
```

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

Lab Exercises

Name:

Lab Exercise 3 — Rational Numbers

```
58
59 // function reduction definition
60 void RationalNumber::reduction()
61 {
62    int largest, gcd = 1;  // greatest common divisor;
63
64    largest = ( numerator > denominator ) ? numerator: denominator;
65
66    for ( int loop = 2; loop <= largest; loop++ )
67        if ( numerator % loop == 0 && denominator % loop == 0 )
68           gcd = loop;
69
70    numerator /= gcd;
71    denominator /= gcd;
72 } // 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 std::cout;
5  using std::endl;
6
7  #include "RationalNumber.h"
8
9  int main()
10 {
11    RationalNumber c( 7, 3 ), d( 3, 9 ), x;
12
13    c.printRational();
14    cout << " + ";
15    d.printRational();
16    cout << " = ";
17    x = c + d; // test overloaded operators + and =
18    x.printRational();
19
20    cout << '\n';
21    c.printRational();
22    cout << " - ";
23    d.printRational();
24    cout << " = ";
25    x = c - d; // test overloaded operators - and =
26    x.printRational();
27
28    cout << '\n';
29    c.printRational();
30    cout << " * ";
31    d.printRational();
32    cout << " = ";
33    x = c * d; // test overloaded operators * and =
34    x.printRational();
35
36    cout << '\n';
37    c.printRational();
```

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

Lab Exercises

Name:

Lab Exercise 3 — Rational Numbers

```
38      cout << " / ";
39      d.printRational();
40      cout << " = ";
41      x = c / d; // test overloaded operators / and =
42      x.printRational();
43
44      cout << '\n';
45      c.printRational();
46      cout << " is:\n";
47
48      // test overloaded greater than operator
49      cout << ( ( c > d ) ? "  > " : "  <= " );
50      d.printRational();
51      cout << " according to the overloaded > operator\n";
52
53      // test overloaded less than operator
54      cout << ( ( c < d ) ? "  < " : "  >= " );
55      d.printRational();
56      cout << " according to the overloaded < operator\n";
57
58      // test overloaded greater than or equal to operator
59      cout << ( ( c >= d ) ? "  >= " : "  < " );
60      d.printRational();
61      cout << " according to the overloaded >= operator\n";
62
63      // test overloaded less than or equal to operator
64      cout << ( ( c <= d ) ? "  <= " : "  > " );
65      d.printRational();
66      cout << " according to the overloaded <= operator\n";
67
68      // test overloaded equality operator
69      cout << ( ( c == d ) ? "  == " : "  != " );
70      d.printRational();
71      cout << " according to the overloaded == operator\n";
72
73      // test overloaded inequality operator
74      cout << ( ( c != d ) ? "  != " : "  == " );
75      d.printRational();
76      cout << " according to the overloaded != operator" << endl;
77      return 0;
78   } // 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

```
1  // Debugging: Decimal.h
2
3  #ifndef DECIMAL_H
4  #define DECIMAL_H
5
6  #include <iostream>
7
8  using std::ostream;
9  using std::istream;
10
11 // class Decimal definition
12 class Decimal
13 {
14 public:
15    friend istream operator>>( istream &, const Decimal & );
16    Decimal( double = 0.0 );
17
18    void setInteger( double );
19    void setDecimal( double );
20
21    Decimal &operator=( const Decimal );
22    Decimal +( Decimal );
23    Decimal +=( Decimal ) const;
24    Decimal &operator++();
25    Decimal operator++( double );
26    bool operator==( const Decimal );
27 private:
28    friend ostream &operator<<( const Decimal & );
29    double integer;
30    double decimal;
31 }; // end class Decimal
32
33 #endif // DECIMAL_H
```

Fig. L 11.12 | Decimal.h.

```
1  // Debugging: Decimal.cpp
2
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7
8  #include <cmath>
9
10 #include "Decimal.h"
11
12 // constructor
13 Decimal::Decimal( double n )
14 {
15    decimal = modf( n, &integer );
16 } // end class Decimal constructor
17
```

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

Lab Exercises

Name:

Debugging

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

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

Lab Exercises

Name:

Debugging

```
74  // function operator+ definition
75  Decimal Decimal::operator+( Decimal d )
76  {
77     Decimal result;
78
79     result.setDecimal( decimal + d.decimal );
80     result.setInteger( integer + d.integer );
81
82     if ( result.decimal >= 1 )
83     {
84        result.decimal--;
85        result.integer++;
86
87     } // end if
88     else if ( result.decimal <= -1 )
89     {
90        result.decimal++;
91        result.integer--;
92     } // end if
93
94     return result;
95  } // end function operator+
96
97  // function operator+= definition
98  Decimal Decimal::operator+=( Decimal d ) const
99  {
100    *this = *this += d;
101    return *this;
102 } // end function operator+=
103
104 // function operator++ definition
105 Decimal &Decimal::operator++()
106 {
107    integer++;
108    return integer;
109 } // end function operator++
110
111 // function operator++ definition
112 Decimal Decimal::operator++( double )
113 {
114    Decimal temp = *this;
115
116    integer++;
117    return *this;
118 } // end function operator++
119
120 // function operator== definition
121 bool Decimal::operator==( const Decimal d )
122 {
123    return ( integer == d.integer && decimal == d.decimal );
124 } // 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
5  using std::cout;
6  using std::endl;
7  using std::cin;
8
9  #include "Decimal.h"
10
11 int main()
12 {
13    Decimal test1;
14    Decimal test2;
15    Decimal test3( 1.234 );
16
17    cout << "Initial values:\n"
18         << test1 << endl << test2 << endl << test3
19         << endl << endl;
20
21    cin >> test1 >> test2;
22
23    cout << "The sum of test1 and test2 is: "
24         << test1 + test2 << endl;
25    test3 += test1++ + ++test2;
26
27    cout << "\nfinal values:\n"
28         << "test1 = " << test1 << endl
29         << "test2 = " << test2 << endl
30         << "test3 = " << test3 << endl;
31
32    if ( test1 != test3 )
33       cout << "test1 and test3 are not equal to each other\n";
34
35    return 0;
36 } // 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 std::ostream;
5  using std::endl;
6
7  class Polygon
8  {
9  public:
10    Polygon( int = 0, int = 0 );
11    ~Polygon();
12
13    void addVertex( int, int );
14    int getNumVertices() const;
15 private:
16    int *xPts;
17    int *yPts;
18    int numVertices;
19 }; // end class Polygon
20
21 // default constructor
22 Polygon::Polygon( int x, int y )
23 {
24    xPts = new int[ 1 ];
25    yPts = new int[ 1 ];
26
27    xPts[ 0 ] = x;
28    yPts[ 0 ] = y;
29    numVertices = 1;
30 } // end class Polygon constructor
31
32 // destructor
33 Polygon::~Polygon()
34 {
35    delete [] xPts;
36    delete [] yPts;
37 } // end class Polygon destructor
38
39 // function addVertex definition
40 void Polygon::addVertex( int x, int y )
41 {
```

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

Postlab Activities

Name:

Coding Exercises

```
42     int *copyX = new int[ numVertices + 1 ];
43     int *copyY = new int[ numVertices + 1 ];
44
45     for ( int i = 0; i < numVertices; i++ )
46     {
47        copyX[ i ] = xPts[ i ];
48        copyY[ i ] = yPts[ i ];
49     } // end for
50
51     copyX[ numVertices ] = x;
52     copyY[ numVertices ] = y;
53
54     delete [] xPts;
55     delete [] yPts;
56
57     xPts = copyX;
58     yPts = copyY;
59     numVertices++;
60  } // end function addVertex
61
62  // function getNumVertices
63  int Polygon::getNumVertices() const
64  {
65     return numVertices;
66  } // 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 `Polygon`s for equality. This member function should return a `bool`ean 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 at `www.deitel.com` and `www.prenhall.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. 11.5 of *C++ How to Program: Fifth 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
```

Postlab Activities

Name:

Programming Challenges

Postlab Activities

Name:

Programming Challenges