Line 20 declares a range-checked string object and line 21 sets the default comparison routine for this object, should one be needed. Line 22 initializes the object with a string value. Lines 23 through 25 output the lower and upper bounds and the value. Line 26 displays the number of characters in the string by means of a system-supplied string length function and the implicit type conversion operator for the **Range** class. Finally, the program ends with a valid exit code.

The following shows the output from the program:

```
r1 has an inclusive low bound of 2.5, an inclusive high bound of 8.8,
and a value of 4.3
4.3 * 1.9 = 8.17
r2 has an inclusive low bound of D, an inclusive high bound of K,
a value of EFG, and a length of 3
```

inline void set_compare (**Range_Compare** *r_fcn*);
Sets the compare function for this class of **Range**<*Type*,*lbound*,*hbound*>. **Range_Compare** is a function of type **int** (**Function*)(**const** *Type&*, **const** *Type&*).

inline operator Type () **const**;
Overloads the implicit conversion operator for the parameterized type to facilitate mixed-type expressions and statements.

Range Example

3.12 The following program declares two range-checking objects, one of type **double** and one of type **char***. Each has type-specific upper and lower bounds that, if violated, result in a run-time exception. Values are assigned to each object and the implicit use of the type conversion operator is demonstrated.

```
1   #include <COOL/Range.h>                          // Include range header file
2   #include <string.h>                              // C++ ANSI C string functions


3   DECLARE Range<double,2.5,8.8>;                   // Declare range of doubles
4   IMPLEMENT Range<double,2.5,8.8>;                 // Implement range of doubles
5   DECLARE Range<char*,"D","K">;                    // Declare range of strings
6   IMPLEMENT Range<char*,"D","K">;                  // Implement range of strings

7   int my_compare (const charP& s1, const charP& s2) {
8    return (strcmp (s1, s2));
9   }

10  int main (void) {
11                                                   // Range-checked double
12   r1.set(4.3);                                    // Assign value
13   cout << "r1 has an inclusive low bound of " << r1.low(); // Output low and
14   cout << "an inclusive high bound of " << r1.high() << ",\n"; // High
15   cout << "and a value of " << (double)r1 << "\n"; // Output value
16   double d1 = 1.9;                                // Declare a double
17   cout << (double)r1 << " * " << d1 << " = ";     // Output equation
18   r1.set (d1 * r1);                               // Calculate value
19   cout << (double)r1 << "\n"; // And display it
20   Range<charP,"D","K"> r2;                        // Range-checked string
21   r2.set_compare (&my_compare);                   // Set compare function
22   r2.set("EFG");                                  // Assign value
23   cout << "r2 has an inclusive low bound of " << r2.low();
24   cout << "an inclusive high bound of " << r2.high() << ",\n";
25   cout << "a value of " << (char*)r2;             // Output string value
26   cout << ", and a length of " << strlen (r2) << "\n"; // Output length
27   return 0;                                       // Exit with OK status
28  }
```

Line 1 includes the COOL `Range.h` class header file and line 2 includes the COOL `String.h` class header file. Lines 3 through 6 declare and implement two kinds of range-checking objects: one a **double** with a low bound of 2.5 and a high bound of 8.8, and the other a character string object with a low bound of `"D"` and a high bound of `"K"`. Lines 7 through 9 define a comparison function for the range-checked string object, although in this program, it is not actually used. Line 11 declares a range object of type **double** with upper and lower bounds as before and line 12 gives this object a value. Lines 13 and 14 output the lower and upper bounds and line 15 displays the value of the object via a cast. Lines 16 through 18 show the object used in an arithmetic expression and line 19 prints the result.

Range Class

3.11 The parameterized **Range**<*Type*,*lbound*,*hbound*> class enables arbitrary user-defined ranges to be implemented in C++ classes. Typically, but not always, this is used with other number classes to select a range of valid values for a particular numerical type. Features and advantages of this class are discussed in this section. However, complete details of parameterized templates are provided in Section 5.

The **Range**<*Type*,*lbound*,*hbound*> class is publicly derived from the **Range** class and supports user-defined ranges for a type of object or built-in data type. This allows other higher level data structures such as the **Rational** and **Complex** classes to be restricted to a range of values. The programmer does not have to add bounds-checking code to the application. A vector of positive integers, for example, would be easy to declare, facilitating bounds checking restricted to the code that implements the type, not the vector.

The inclusive low and high bounds for the range are specified as arguments to the parameterized type declaration and implementation macro calls. They are declared as C++ constants of the appropriate type. No storage is allocated, and all references are compiled out by the compiler. Once declared, a **Range**<*Type*,*lbound*,*hbound*> object cannot have its upper or lower bounds changed because maintenance of all instances would require significant and unwarranted overhead.

Name: **Range**<*Type*,*lbound*,*hbound*> — A parameterized range

Synopsis: **#include** <COOL/Range.h>

Base Classes: **Range**

Friend Classes: None

Constructors:

Range<*Type*,*lbound*,*hbound*> ();
Creates an empty range object of the specified type and ranges.

Range<*Type*,*lbound*,*hbound*> (**const Type&** *value*);
Creates a range object with the specified value. If *value* is outside of the lower and upper bounds, an Error exception is raised.

Range<*Type*,*lbound*,*hbound*> (**const Range**<*Type*,*lbound*,*hbound*>**&** *r*);
Creates a new range object with the same value as the range object *r*.

Member Functions:

inline const Type& high () **const**;
Returns a reference to the upper limit of the range.

inline const Type& low () **const**;
Returns a reference to the lower limit of the range.

inline Range<*Type*,*lbound*,*hbound*>& **operator=**
(**const Range**<*Type*,*lbound*,*hbound*>**&** *r*);
Overloads the assignment operator for the **Range**<*Type*,*lbound*,*hbound*> class and assigns the range object the value of *r*. This function returns a reference to the updated object.

inline void set (**const Type&** *value*);
Sets the value of the range object to *value* if within the lower and upper limits; otherwise, this function raises an **Error** exception.

Bignum Example

3.10 The following program uses the **Bignum** integer data type in a semantically equivalent manner to the built-in **int** or **long** data types to perform arithmetic and logical operations. The only difference is that the values manipulated are larger than `MAX_INT` or `MAX_LONG` would allow on a 32-bit computer.

```
1   #include <COOL/Bignum.h>                    // Include Bignum class

2   int main (void) {
3    Bignum b1;                                 // Create Bignum object
4    Bignum b2 = "0xFFFFFFFF";                  // Create Bignum object
5    Bignum b3 = "1.2345e30";                   // Create Bignum object
6    cout << "b2 = " << b2 << "\n";             // Display value of b2
7    cout << "b3 = " << b3 << "\n";             // Display value of b3
8    b1 = b2 + b3;                              // Add b2 and b3
9    cout << "b2 + b3 = " << b1 << "\n";        // Display result
10   b1 = b2 - b3;                              // Subtract b3 from b2
11   cout << "b2 - b3 = " << b1 << "\n";        // Display result
12   b1 = b2 * b3;                              // Multiply b2 and b3
13   cout << "b2 * b3 = " << b1 << "\n";        // Display result
14   b1 = b3 / b2;                              // Divide b2 into b3
15   cout << "b3 / b2 = " << b1 << "\n";        // Display result
16   b1 = b3 % b2;                              // Get b3 modulo b2
17   cout << "b3 % b2 = " << b1 << "\n";        // Display result
18     return 0;                                // Exit with status code
19  }
```

Line 1 includes the COOL **Bignum** class header file. Line 3 creates a **bignum** object initialized to zero. Lines 3 and 4 create **Bignum** objects initialized to very large integer values. Lines 5 and 6 output these values on the standard output stream. Lines 8 through 17 compute the sum, difference, product, quotient, and remainder of various **Bignum** values and output the answer. Finally, the program ends with a valid exit code.

The following shows the output from the program:

```
1   b2 = 4294967295
2   b3 = 1234500000000000000000000000000
3   b2 + b3 = 1234500000000000000004294967295
4   b2 - b3 = -1234499999999999999995705032705
5   b2 * b3 = 530213712567450000000000000000000000000000
6   b3 / b2 = 287429429657624435997
7   b3 % b2 = 64281885
```

inline friend Bignum operator– (**const Bignum&** *bn1,*
const Bignum& *bn2*);
Overloads the subtraction operator to provide subtraction for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator* (**const Bignum&** *bn1,* **const Bignum&** *bn2*);
Overloads the multiplication operator to provide multiplication for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator/ (**const Bignum&** *bn1,* **const Bignum&** *bn2*);
Overloads the division operator for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator% (**const Bignum&** *bn1,* **const Bignum&** *bn2*);
Overloads the modulus operator for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator& (**const Bignum&** *bn1,* **const Bignum&** *bn2*);
Overloads the logical AND operator for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator^ (**const Bignum&** *bn1,* **const Bignum&** *bn2*);
Overloads the logical exclusive-or operator for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator| (**const Bignum&** *bn1,* **const Bignum&** *bn2*);
Overloads the logical OR operator for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator>> (**const Bignum&** *bn1,* **const Bignum&** *bn2*);
Overloads the right shift operator for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Bignum operator<< (**const Bignum&** *bn1,* **const Bignum&** *bn2*);
Overloads the left shift operator for the **Bignum** class. A new **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend ostream& operator<< (**ostream&** *os*, **const Bignum&** *bn*);
Overloads the output operator for a reference to a **Bignum** object to provide a formatted output.

inline friend ostream& operator<< (**ostream&** *os*, **const Bignum*** *bn*);
Overloads the output operator for a pointer to a **Bignum** object to provide a formatted output.

void operator|= (const Bignum& *bn*);
Overloads the logical OR with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator>>= (const Bignum& *bn*);
Overloads the right shift with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator<<= (const Bignum& *bn*);
Overloads the left shift with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Boolean operator== (const Bignum& *bn*) const;
Overloads the equality operator for the **Bignum** class. This function returns **TRUE** if the near-infinite precision integers have the same value; otherwise, this function returns **FALSE**.

inline Boolean operator!= (const Bignum& *bn*) const;
Overloads the inequality operator for the **Bignum** class. This function returns **TRUE** if the near-infinite precision integers have different values; otherwise, this function returns **FALSE**.

Boolean operator< (const Bignum& *bn*) const;
Overloads the less than operator for the **Bignum** class and returns **TRUE** if the object is less than the specified argument; otherwise, this function returns **FALSE**.

inline Boolean operator<= (const Bignum& *bn*) const;
Overloads the less than or equal operator for the **Bignum** class. This function returns **TRUE** if the object is less than or equal to the value of the specified argument ; otherwise, this function returns **FALSE**.

Boolean operator> (const Bignum& *bn*) const;
Overloads the greater than operator for the **Bignum** class and returns **TRUE** if the object is greater than the specified argument; otherwise, this function returns **FALSE**.

inline Boolean operator>= (const Bignum& *bn*) const;
Overloads the greater than or equal operator for the **Bignum** class. This function returns **TRUE** if the object is greater than or equal to the value of the specified argument ; otherwise, this function returns **FALSE**.

operator short ();
Overloaded operator to provide implicit conversion between **Bignum** objects and the built-in **short** type when appropriate.

inline N_Status status () const;
Returns the numerical exception state of the Bignum object.

Friend Functions:

friend Bignum operator+ (const *Bignum& bn1,* const *Bignum& bn2*);
Overloads the addition operator to provide addition for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Boolean operator! () **const**;
Overloads the unary negation operator for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Bignum operator~ () **const**;
Overloads the unary exclusive-or operator for the **Bignum** class. A new Bignum object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Bignum& operator++ ();
Overloads the increment operator to provide an increment capability for the **Bignum** class. A reference to the modified **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Bignum& operator— ();
Overloads the decrement operator to provide a decrement capability for the **Bignum** class. A reference to the modified **Bignum** object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Bignum& operator+= (**const Bignum&** *bn*);
Overloads the addition with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator–= (**const Bignum&** *bn*);
Overloads the subtraction with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator*= (**const Bignum&** *bn*);
Overloads the multiplication with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator/= (**const Bignum&** *bn*);
Overloads the division with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator%= (**const Bignum&** *bn*);
Overloads the modulus with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator&= (**const Bignum&** *bn*);
Overloads the logical AND with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator^= (**const Bignum&** *bn*);
Overloads the exclusive-or with assignment operator for the **Bignum** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Name: **Bignum** — Infinite precision integers

Synopsis: **#include** <COOL/Bignum.h>

Base Classes: **Generic**

Friend Classes: None

Constructors:

inline Bignum ();
Simple constructor to create a near-infinite precision integer object initialized to zero.

Bignum (**const char*** *str*);
Constructor to create a near-infinite precision integer object from the character string representation *str*.

Bignum (**double** *d*);
Constructor to create a near-infinite precision integer object from the double value *d*.

Bignum (**long** *l*);
Constructor to create a near-infinite precision integer object from the long integer value *l*.

Bignum (**const Bignum&** *bn*);
Constructor to create a near-infinite precision integer object from *bn*.

Member Functions:

operator double ();
Overloaded operator to provide implicit conversion between **Bignum** objects and the built-in **double** type when appropriate.

operator float ();
Overloaded operator to provide implicit conversion between **Bignum** objects and the built-in **float** type when appropriate.

operator int ();
Overloaded operator to provide implicit conversion between **Bignum** objects and the built-in **int** type when appropriate.

operator long ();
Overloaded operator to provide implicit conversion between **Bignum** objects and the built-in **long** type when appropriate.

Bignum operator– () **const**;
Overloads the unary minus operator for the **Bignum** class and returns a new object whose value is the negated value of the object. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Bignum& operator= (**const char*** *str*);
Overloads the assignment operator for the **Bignum** class and assigns the integer representation from the character string *str* to the near-infinite precision integer object. A reference to the updated object is returned.

Bignum& operator= (**const** *Bignum& bn*);
Overloads the assignment operator for the **Bignum** class and assigns *bn* to the near-infinite precision integer object. A reference to the updated object is returned.

Bignum Class

3.9 The **Bignum** class implements near-infinite precision integers and arithmetic by using a dynamic bit vector. A **Bignum** object will grow in size as necessary to hold its integer value. Implicit conversion to the system defined types **short**, **int**, **long**, **float**, and **double** is supported by overloaded operator member functions. Addition and subtraction operators are performed by simple bitwise addition and subtraction on **unsigned short** boundaries with checks for carry flag propagation. The multiplication, division, and remainder operations utilize the algorithms from Knuth's Volume 2 of "*The Art of Computer Programmin*g". However, despite the use of these algorithms and inline member functions, arithmetic operations on **Bignum** objects are considerably slower than the built-in integer types that use hardware integer arithmetic capabilities.

NOTE: The **Bignum** class requires that the built-in type **long** is larger than the built-in type **short** and can accommodate the result of multiplying two **short** values. The maximum positive value that can be represented by the **Bignum** class is:
`(2^(sizeof(unsigned long) * sizeof(unsigned short)))-1.`

The **Bignum** class supports the parsing of character string representations of all the literal number formats. The following table shows an example of a character string representation on the left and a brief description of the interpreted meaning on the right:

Character String Representation	*Interpreted Meaning*
1234	1234
1234l	1234
1234L	1234
1234u	1234
1234U	1234
1234ul	1234
1234UL	1234
01234	1234 in octal (leading 0)
0x1234	1234 in hexadecimal (leading 0x)
0X1234	1234 in hexadecimal (leading 0X)
123.4	123 (value truncated)
1.234e2	123 (exponent expanded/truncated)
1.234e–5	0 (truncated value less than 1)

The **Bignum** class implements common arithmetic exception handling and provides application support for detecting negative infinity, positive infinity, overflow, and underflow that may result from an operation. If one of these conditions is detected, an exception is raised. The programmer can provide an exception handler at runtime to take care of this problem. If no such handler is available, an error message is printed and program execution terminates. See Section 13 for more information on the COOL exception handling mechanism.

Rational Example

3.8 The following program uses the **Rational** class and the built-in **float** type to illustrate the added precision available for calculations involving multiplication, division, addition, and determining the remainder for numeric ratios. The first half of the program calculates answers for problems using the **Rational** class. The second half calculates answers for the same problems using the built-in **float** type. The results from each are printed on the standard output stream for comparison of precision.

```
1   #include <COOL/Rational.h>                  // Include COOL Rational class

2   int main (void) {
3    Rational r1 (10,3);                        // Create rational object
4    Rational r2 (-4,27), r3;                   // Create rational objects
5    r3 = r1 + r2;                              // Calculate sum of values
6    cout << r1 << " + " << r2 << " = " << r3 << "\n"; // And display result
7    r3 = r1 * r2;                              // Calculate product of values
8    cout << r1 << " * " << r2 << " = " << r3 << "\n"; // And display result
9    r3 = r1 / r2;                              // Calculate quotient of values
10   cout << r1 << " / " << r2 << " = " << r3 << "\n"; // And display result
11   r3 = r1 % r2;                              // Calculate remainder of values
12   cout << r1 << " % " << r2 << " = " << r3 << "\n"; // And display result

13   double d1 = double (10.0 / 3.0);           // Create double ratio
14   double d2 = double (-4.0 / 27.0), d3;      // Create double ratios
15   d3 = d1 + d2;                              // Calculate sum of values
16   cout << d1 << " + " << d2 << " = " << d3 << "\n"; // And display result
17   d3 = d1 * d2;                              // Calculate product of values
18   cout << d1 << " * " << d2 << " = " << d3 << "\n"; // And display result
19   d3 = d1 / d2;                              // Calculate quotient of values
20   cout << d1 << " / " << d2 << " = " << d3 << "\n"; // And display result
21   return 0;                                  // Return valid success code
22  }
```

Line 1 includes the COOL `Rational.h` class header file. Lines 3 and 4 declare three rational objects (`r1`, `r2`, `r3`), the first two of which have initial values of 10/3 and –4/27, respectively. Lines 5 and 6 calculate the sum of the two rational objects, assign it to the third, and display the answer. Likewise, lines 7 and 8 calculate the product, lines 9 and 10 calculate the quotient, and lines 11 and 12 calculate the remainder of the same two rational numbers. Lines 13 through 20 perform the same calculations with the built-in type **double** as were performed in lines 3 through 10. As indicated from the results, a loss of precision occurs from the floating point calculations, thus highlighting the potential benefit of using the ratios maintained by the **Rational** number class. Finally, the program ends with a valid exit code.

The following shows the output from the program:

```
10/3 + -4/27 = 86/27
10/3 * -4/27 = -40/81
10/3 / -4/27 = -45/2
10/3 % -4/27 = 2/27
3.33333 + -.148148 = 3.18519
3.33333 * -.148148 = -.493827
3.33333 / -.148148 = -22.5
```

friend Rational operator* (**const Rational&** *r1*, **const Rational&** *r2*);
Overloads the multiplication operator for the **Rational** class. A new rational object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend Rational operator/ (**const Rational&** *r1*, **const Rational&** *r2*);
Overloads the division operator for the **Rational** class. A new rational object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Rational operator% (**const Rational&** *r1*, **const Rational& r2**);
Overloads the modulus operator for the **Rational** class. A new rational object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend ostream& operator<< (**ostream&** *os*, **const Rational&** *r*);
Overloads the output operator for a reference to a rational object to provide a formatted output capability.

inline friend ostream& operator<< (**ostream&** *os*, **const Rational*** *r*);
Overloads the output operator for a pointer to a rational object to provide a formatted output capability.

inline Rational& operator— ();
Provides a decrement capability for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised. A reference to the modified Rational object is returned.

inline Boolean operator== (**const Rational&** *r*) **const**;
Overloads the equality operator for the **Rational** class. This function returns **TRUE** if the rational numbers have the same value; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Rational&** *r*) **const**;
Overloads the inequality operator for the **Rational** class. This function returns **TRUE** if the rational numbers have different values; otherwise, this function returns **FALSE**.

Boolean operator< (**const Rational&** *r*) **const**;
Overloads the less-than-operator for the **Rational** class and returns **TRUE** if the object is less than the specified argument; otherwise, this function returns **FALSE**.

inline Boolean operator<= (**const Rational&** *r*) **const**;
Overloads the less-than-or-equal operator for the **Rational** class. This function returns **TRUE** if the object is less than or equal to the value of the specified argument; otherwise, this function returns **FALSE**.

Boolean operator> (**const Rational&** *r*) **const**;
Overloads the greater-than operator for the **Rational** class and returns **TRUE** if the object is greater than the specified argument; otherwise, this function returns **FALSE**.

inline Boolean operator>= (**const Rational&** *r*) **const**;
Overloads the greater-than-or-equal operator for the **Rational** class. This function returns **TRUE** if the object is greater than or equal to the value of the specified argument ; otherwise, this function returns **FALSE**.

long round () **const**;
Returns an integer that represents the value of the rational object truncated towards the nearest integer.

operator short ();
Overloaded operator to provide implicit conversion between rational objects and the built-in **short** type when appropriate.

inline N_Status status () **const**;
Returns the numerical exception state of the rational object.

inline long truncate () **const**;
Returns an integer that represents the value of the rational object truncated towards zero.

Friend Functions:

friend Rational operator+ (**const Rational&** *r1*, **const Rational&** *r2*);
Overloads the addition operator for the **Rational** class. A new rational object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend Rational operator– (**const Rational&** *r1*, **const Rational&** *r2*);
Overloads the subtraction operator to provide subtraction for the **Rational** class. A new rational object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

operator int ();
Overloaded operator to provide implicit conversion between rational objects and the built-in **int** type when appropriate.

Rational& invert ();
Returns a reference to the inverse of the rational number object.

operator long ();
Overloaded operator to provide implicit conversion between rational objects and the built-in **long** type when appropriate.

inline long numerator () **const**;
Returns the numerator value of the object.

inline Rational operator–();
Overloads the unary minus operator for the **Rational** class and returns a new object whose value is the negated value of the object. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Rational& operator= (**const Rational&** *r*);
Overloads the assignment operator for the **Rational** class and assigns one rational number to have the value of another. A reference to the updated object is returned.

void operator+= (**const Rational&** *r*);
Overloads the addition-with-assignment operator for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline void operator–= (**const Rational&** *r*);
Overloads the subtraction-with-assignment operator for the **Rational** class. If the operation results in an arithmetic exception of some type, the appropriate exception is raised.

void operator*= (**const Rational&** *r*);
Overloads the multiplication-with-assignment operator for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline void operator/= (**const Rational&** *r*);
Overloads the division-with-assignment operator for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

void operator%= (**const Rational&** *r*);
Overloads the modulus with assignment operator for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Boolean operator!() **const**;
Overloads the logical NOT operator for the **Rational** class and returns **TRUE** if the complex number has a zero value; otherwise, this function returns **FALSE**.

inline Rational& operator++ ();
Provides an increment capability for the **Rational** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised. A reference to the modified Rational object is returned.

Rational Class

3.7 The **Rational** class provides infinite precision rational numbers and arithmetic using the built-in **long** type for the numerator and denominator objects. Consequently, a rational object will grow in 32-bit chunks as necessary. Implicit conversion to the system-defined types **short**, **int**, **long**, **float**, and **double** is supported by overloaded operator member functions. However, arithmetic operations on rational objects are slower than the built-in integer types.

The **Rational** class implements common arithmetic exception handling and provides application support for detecting negative infinity, positive infinity, overflow, and underflow that may result from an operation. If one of these conditions is detected or if an attempt to convert from a **Rational** with no value to a built-in type is made, an exception is raised. The programmer can provide an exception handler at runtime to take care of this problem. If no such handler is available, an error message is printed and program execution terminates. See Section 13 for more information on the COOL exception handling mechanism.

Name: **Rational** — Infinite precision rational numbers

Synopsis: **#include** <COOL/Rational.h>

Base Classes: **Generic**

Friend Classes: None

Constructors:

inline Rational ();
Simple constructor to create a new rational object.

Rational (**long** *n,* **long** *d* = 1);
Constructor that specifies an integer numerator and optional denominator arguments to create a new rational object.

Rational (**const Rational&** *r*);
Constructor that takes a reference to an existing rational object and creates a new object with the same value.

Member Functions:

inline long ceiling () **const**;
Returns an integer that represents the value of the rational object truncated towards positive infinity.

inline long denominator () **const**;
Returns the denominator value of the object.

inline operator double ();
Overloaded operator to provide implicit conversion between rational objects and the built-in **double** type when appropriate.

operator float ();
Overloaded operator to provide implicit conversion between rational objects and the built-in **float** type when appropriate.

inline long floor () **const**;
Returns an integer that represents the value of the rational object truncated towards negative infinity.

```
22                  in_parallel (in_series (resistor (100.0), inductor (0.2)),
23                               in_parallel (capacitor (0.000001),
24                                    resistor (10000000.0))));
25      cout << "Circuit impedance is " << circuit << " at frequency " <<
                    FREQUENCY << "\n";
26      return 0;                                    // Exit with OK status
27    }
```

Line 1 includes the COOL `Complex.h` class header file. Lines 2 and 3 define a frequency constant and a value `OMEGA` based upon pi and the frequency and is used in calculating impedance formulas. Lines 4 through 6 define a function for calculating the impedance of two components placed in series. Similarly, lines 7 through 9 define a function for calculating the impedance of two components placed in parallel. Lines 10 through 18 provide functions for determining the impedance of resistors, inductors, and capacitors, based upon their tolerances. Line 21 is the heart of the program that calls the necessary functions to calculate the impedance of a circuit. Finally, the result is sent to the standard output and the program ends with a successful exit code. Figure 3.1 illustrates the circuit used in this example program.

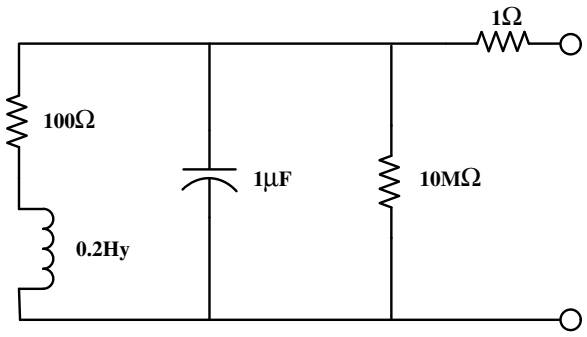


Figure 3.1

The following shows the output from the program:

```
Circuit impedance is (2000.6,0.00747744) at frequency 346.87
```

inline friend Complex operator– (**const Complex&** *c1*,
const Complex& *c2*);
Overloads the subtraction operator for the **Complex** class. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend Complex operator* (**const Complex&** *c1*,
const Complex& *c2*);
Overloads the multiplication operator for the **Complex** class. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

friend Complex operator/ (**const Complex&** *c1*, **const Complex&** *c2*);
Overloads the division operator to provide division for the **Complex** class. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline friend ostream& operator<< (**ostream&** *os*, **const Complex&** *c*);
Overloads the output operator for a reference to a complex object to provide a formatted output.

inline friend ostream& operator<< (**ostream&** *os*, **const Complex*** *c*);
Overloads the output operator for a pointer to a complex object to provide a formatted output.

Complex Example

3.6 The following impedance example using complex numbers is accredited to a LISP program published in *LISP*, written by Patrick Henry Winston and Berthold Klaus Paul Horn. This example calculates the impedance of an electrical circuit operating at a given frequency by using standard formulas from basic hardware design texts.

```
1   #include <COOL/Complex.h>                    // Include complex header file
2   #define FREQUENCY 346.87
3   #define OMEGA (2 * 3.14159265358979323846 * FREQUENCY)

4   inline Complex in_series (const Complex& c1, const Complex& c2) {
5    return (c1+c2);
6   }

7   inline Complex in_parallel (const Complex& c1, const Complex& c2) {
8    return ((c1.invert() + c2.invert()).invert ());
9   }

10  inline Complex resistor (double r) {
11   return Complex (r);
12  }

13  inline Complex inductor (double i) {
14   return (Complex (0.0, i * OMEGA));
15  }

16  inline Complex capacitor (double c) {
17   return (Complex (0.0, -1.0 / (c * OMEGA)));
18  }

19  int main (void) {
20   Complex circuit;
21   circuit = in_series (resistor (1.0),
```

inline Complex& operator— ();
Overloads the decrement operator to provide a decrement capability for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised. A reference to the updated complex object is returned.

inline Boolean operator! () **const**;
Overloads the logical NOT operator for the **Complex** class and returns **TRUE** if the complex number has a zero value; otherwise, this function returns **FALSE**.

inline Boolean operator== (**const Complex&** *c*) **const**;
Overloads the equality operator for the **Complex** class. This function returns **TRUE** if the complex numbers have the same value; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Complex&** *c*) **const**;
Overloads the inequality operator for the **Complex** class. This function returns **TRUE** if the complex numbers have different values; otherwise, this function returns **FALSE**.

inline double real () **const**;
Returns the real part of the complex number.

operator short ();
Overloaded operator to provide implicit conversion between complex objects and the built-in **short** type when appropriate.

inline Complex sin (**Complex&** *c*) **const**;
Calculates the sine of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Complex sinh (**Complex&** *c*) **const**;
Calculates the hyperbolic sine of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline N_Status status () **const**;
Returns the numerical exception state of the complex object.

inline Complex tan (**Complex&** *c*) **const**;
Calculates the tangent of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Complex tanh (**Complex&** *c*) **const**;
Calculates the hyperbolic tangent of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Friend Functions:

inline friend Complex operator+ (**const Complex&** *c1*,
const Complex& *c2*);
Overloads the addition operator to provide addition for the **Complex** class. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

operator float ();
Overloaded operator to provide implicit conversion between complex objects and the built-in **float** type when appropriate.

inline double imaginary () **const**;
Returns the imaginary part of the complex number.

inline Complex invert () const;
Returns the reciprocal of a complex number. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

operator int ();
Overloaded operator to provide implicit conversion between complex objects and the built-in **int** type when appropriate.

operator long ();
Overloaded operator to provide implicit conversion between complex objects and the built-in **long** type when appropriate.

Complex operator– ();
Overloads the unary minus operator for the **Complex** class and returns a new object whose value is the negated real value of the object. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

Complex& operator= (**const Complex&** *c*);
Overloads the assignment operator for the **Complex** class and assigns one complex number to have the value of another. A reference to the updated object is returned.

inline void operator+= (**const Complex&** *c*);
Overloads the addition-with-assignment operator for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline void operator–= (**const Complex&** *c*);
Overloads the subtraction-with-assignment operator for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline void operator*= (**const Complex&** *c*);
Overloads the multiplication-with-assignment operator for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline void operator/= (**const Complex&** *c*);
Overloads the division-with-assignment operator for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Complex& operator++ ();
Overloads the increment operator to provide an increment capability for the **Complex** class. If the operation results in an arithmetic error of some type, the appropriate exception is raised. A reference to the updated complex object is returned.

Complex Class

3.5 The **Complex** class is a complex number class with basic arithmetic support, conversion to and from built-in types, and simple arithmetic exception handling. A **Complex** object has the same precision and range of values as the system-defined type **double.** Implicit conversion to the system-defined types **short**, **int**, **long**, **float**, and **double** is supported by overloaded operator member functions. However, despite the implicit conversions and judicious use of inline member functions, arithmetic operations on **Complex** objects are slower than the built-in types.

The **Complex** class implements common arithmetic exception handling and provides application support for detecting negative infinity, positive infinity, overflow, and underflow that may result from an operation. If one of these conditions is detected or an attempt to convert from a **Complex** with no value to a built-in type is made, an exception is raised. The programmer can provide an exception handler at runtime to take care of this problem. If no such handler is available, an error message is printed and program execution ends. See Section 13 for more information on the COOL exception handling mechanism.

Name: **Complex** — Complex number class

Synopsis: **#include** <COOL/Complex.h>

Base Classes: None

Friend Classes: None

Constructors:

inline Complex ();
Creates a new complex number object initialized to floating point zero.

inline Complex (**double** *real,* **double** *imaginary* = 0.0);
Creates a new complex number object whose real part is set to *real* and whose imaginary part is initialized to the value of *imaginary*.

inline Complex (**const Complex&** *c*);
Creates a new complex number object whose real and imaginary parts are initialized to the values of those of another complex number *c*.

Member Functions:

inline Complex conjugate () const;
Calculates the conjugate of a complex number and returns a new object whose value is the negated imaginary value of the object. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Complex cos (**Complex&** *c*) **const**;
Calculates the cosine of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

inline Complex cosh (**Complex&** *c*) **const**;
Calculates the hyperbolic cosine of a complex number *c*. A new complex object is returned as the result. If the operation results in an arithmetic error of some type, the appropriate exception is raised.

operator double ();
Overloaded operator to provide implicit conversion between complex objects and the built-in **double** type when appropriate.

inline void set_seed (**int** *seed*);

Sets the seed value for the currently-selected random number generator function and reinitilizes the state.

Random Class Example

3.4 The following program creates two random number objects using different generator algorithms to provide random numbers within a specified range. The first uses a variation of the system-supplied `rand()` function and the second a three-congruential linear generator. Ten random numbers from each are sent to the standard output.

```
1   #include <COOL/Random.h>                          // Include Random class

2   int main (void) {
3     Random r1 (SIMPLE, 1, 3.0, 9.0);                // Simple rand() generator
4     Random r2 (THREE_CONGRUENTIAL,1,5.0,11.5);      // Highly random generator
5     cout << "Simple random number generator:\n";     // Output banner title
6     for (int i = 0; i < 10; i++)                    // Generate 10 random numbers
7       cout << " Random number " << i << " is: " << r1.next () << "\n";
8     cout << "\nThree congruential linear random number generator:\n";
9     for (i = 0; i < 10; i++)                        // Generate 10 random numbers
10      cout << " Random number " << i << " is: " << r2.next () << "\n";
11    return (0);                                     // Exit with OK status
12  }
```

Line 1 includes the COOL `Random.h` class header file. Line 3 defines a random number generator of type `SIMPLE` for generator numbers within the range of 3.0 to 9.0 inclusive. Line 4 defines a random number generator of type `THREE_CONGRUENTIAL` for generator numbers within the range of 5.0 to 11.5 inclusive. Lines 6 through 10 utilize two loops to generate and print ten numbers from each generator. Finally, the program ends with a valid exit code.

The following shows the output from the program:

Simple random number generator:

```
Random number 0 is: 6.08322
Random number 1 is: 4.05445
Random number 2 is: 4.85191
Random number 3 is: 6.2072
Random number 4 is: 8.68577
Random number 5 is: 4.03042
Random number 6 is: 7.21339
Random number 7 is: 4.35858
Random number 8 is: 5.96864
Random number 9 is: 3.74832
```

Three congruential linear random number generator:

```
Random number 0 is: 9.26861
Random number 1 is: 7.84012
Random number 2 is: 8.84924
Random number 3 is: 7.22898
Random number 4 is: 8.1818
Random number 5 is: 7.3039
Random number 6 is: 9.18251
Random number 7 is: 10.0368
Random number 8 is: 10.3957
Random number 9 is: 11.3929
```

The **Random** class allows an application to select one of five types of random number generators based upon the usage requirements. Each generator function has different characteristics and all are defined to be of type *RNG_TYPE*. The **SIMPLE** and **SHUFFLE** functions use the system **rand** function, while the **ONE_CONGRUENTIAL**, **THREE_CONGRUENTIAL**, and **SUBTRACTIVE** functions are self-contained, portable implementations. Following are descriptions of each generator function.

- **SIMPLE** — When speed is the predominant concern, this function uses the system-supplied **rand** function. Although sequential correlation of successive random values is a high probability, this function at least ensures that the value's least significant bits are as random as the most significant bits. In many system random generator functions, the value's least significant bits are often less random than the most significant bits.

- **SHUFFLE** — This function uses the **rand** function and a shuffling procedure. Random numbers are stored in a buffer and selected randomly to break up sequential correlation in the system-supplied function.

- **ONE_CONGRUENTIAL** — This self-contained function uses one linear congruential generator instead of the **rand** function to implement a portable random number generator. This guarantees no sequential correlation between the random values returned.

- **THREE_CONGRUENTIAL** — This portable function uses three linear congruential generators to implement a random number generator whose period is essentially infinite and has no sequential correlations.

- **SUBTRACTIVE** — This function implements a portable random number generator that does not use linear congruential generators, but rather an original subtractive member function as suggested in Volume 2 of *The Art of Computer Programmin*g, written by Donald Knuth.

Name: **Random** — A portable, user-selectable random number generator

Synopsis: **#include** <COOL/Random.h>

Base Classes: **Generic**

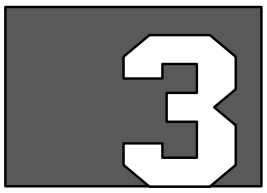
Friend Classes: None

Constructor: **Random** (**RNG_TYPE** *r_type*, **int** *seed* = 1, **float** *lower* = 0.0,
float *upper* = 100.0);
Constructor for a floating-point random number generator that initializes the selected random number generator function with the user-supplied *seed* value.

Member Functions: **inline double next** ();
Returns the next double floating-point random number within the user-specified range.

inline int get_seed () **const;**
Returns the seed value for the currently-selected random number generator.

inline void set_rng (**RNG_TYPE** *r_type*);
Sets the random number generator function to the type selected by the user and reinitializes the state.



NUMBER CLASSES

Introduction

3.1 Simple integers and floating point numbers do not provide the needed precision for many applications. The COOL number classes are a collection of numerically-oriented classes that augment the built-in numerical data types to provide such features as extended precision, range-checked types, and complex numbers. The following classes are discussed in this section:

- **Random**
- **Complex**
- **Rational**
- **Bignum**
- **Range**<*Type,lbound,hbound*>

The **Random** class implements five variations of random number generators, each with different portability, efficiency, and accuracy characteristics. The **Complex** class implements the complex number type for C++ and provides all of the basic arithmetic and trigonometric functions. The **Rational** class uses the built-in **long** type to implement an extended precision rational data type for resolving inadequate round-off or truncation results from the built-in numerical data types. The **Bignum** class implements near-infinite precision integers and arithmetic by using a dynamic bit vector.

Finally, the parameterized **Range**<*Type,lbound,hbound*> class enables arbitrary user-defined ranges to be implemented in C++ classes. Typically, but not always, this is used with other number classes to select a range of valid values for a particular numerical type. Features and advantages of the **Range**<*Type,lbound,hbound*> class are discussed in this section. However, complete details of parameterized templates are provided in Section 5.

Requirements

3.2 This section discusses the number classes. It assumes you have a working understanding of the C++ language and type system. In addition, you should understand the distinction between the concepts and ideas associated with overloaded operators and friend functions.

Random Class

3.3 The **Random** class provides several general-purpose random number generators with features similar to those as described in Chapter 7 of *Numerical Recipes in C*, written by William T. Vetterling. The ANSI C draft standard specifies the **rand** function that allows an application to obtain successive random numbers in a sequence by repeated calls. However, system-supplied random number generators in the form of the **rand** function are generally of poor quality, particularly when true random distribution over a range is important. Specifically, system random number generators are almost always linear congruential generators whose period is not very large. The ANSI C draft specification only requires a modulus of 32767, which can be disastrous for such uses as a Monte Carlo integration over 10^6 points.

Printed on: Wed Apr 18 07:03:17 1990

Last saved on: Tue Apr 17 13:44:06 1990

Document: s3

For: skc