CHAPTER 12
Operator Overloading
Objectives

• To define the Rational class to represent rational numbers (§12.2).
• To understand in general how an operator can be overloaded in C++ (§12.3).
• To know how to overload the relational operators and arithmetic operators (§12.3).
• To know how to overload the shorthand operators (§12.4).
• To know how to overload the array subscript operator [] (§12.5).
• To know how to overload the unary operators (§12.6).
• To know how to overload the prefix and postfix ++ and −− operators (§12.7).
• To know how to overload the stream insertion and extraction operators << and >> (§12.8).
• To define operator functions to perform object conversion (§12.9).
• To overload the = operator to perform a deep copy (§12.11).
12.1 Introduction
In §9.8.9, “String Operators,” you learned how to use the operators to simplify string operations. You can use the + operator to concatenate two strings, the relational operators (==, !==, <, <=, >, and >=) to compare two strings, and the array subscript operator [] to access character. In §12.12, “The C++ vector Class,” you learned how to use the [] operator to access individual elements in a vector. You can define operators in custom classes. This chapter uses the Rational class as an example to demonstrate how to define and use operators. First you will learn how to create a Rational class for supporting rational number operations and then define convenient operators for these operations.

12.2 The Rational Class
A rational number is a number with a numerator and a denominator in the form a/b, where a is the numerator and b is the denominator. For example, 1/3, 3/4, and 10/4 are rational numbers.

A rational number cannot have a denominator of 0, but a numerator of 0 is fine. Every integer a is equivalent to a rational number a/1. Rational numbers are used in exact computations involving fractions; for example, 1/3 = 0.33333.... This number cannot be precisely represented in floating-point format using data type double or float. To obtain the exact result, it is necessary to use rational numbers.

C++ provides data types for integers and floating-point numbers, but not for rational numbers. This section shows how to design a class to represent rational numbers.

A Rational number can be represented using two data fields: numerator and denominator. You can create a Rational number with specified numerator and denominator, or create a default Rational number with numerator 0 and denominator 1. You can add, subtract, multiple, divide, and compare rational numbers. You can also convert a rational number into an integer, floating-point value, or string. The UML class diagram for the Rational class is given in Figure 12.1.

<PD: UML Class Diagram>
Rational
- numerator: long
- denominator: long

+ Rational()
+ Rational(numerator: long, denominator: long)
+ getNumerator(): long
+ getDenominator(): long
+ add(secondRational: Rational &): Rational
+ subtract(secondRational: Rational &): Rational
+ multiply(secondRational: Rational &): Rational
+ divide(secondRational: Rational &): Rational
+ compareTo(secondRational: Rational &): int
+ equals(secondRational: Rational &): bool
+ intValue(): int
+ doubleValue(): double
+ toString(): string
+ gcd(n: long, d: long): long

The numerator of this rational number.
The denominator of this rational number.

Creates a rational number with numerator 0 and denominator 1.
Creates a rational number with specified numerator and denominator.
Returns the numerator of this rational number.
Returns the denominator of this rational number.
Returns the addition of this rational with another.
Returns the subtraction of this rational with another.
Returns the multiplication of this rational with another.
Returns the division of this rational with another.
Returns an int value -1, 0, or 1 to indicate whether this rational number is less than, equal to, or greater than the specified number.
Returns true if this rational number is equal to the specified number.
Returns the numerator / denominator.
Returns the 1.0 * numerator / denominator.
Returns a string in the form “numerator / denominator.” Returns numerator if denominator is 1.
Returns the greatest common divisor between n and d.

Figure 12.1
The properties, constructors, and functions of the Rational class are illustrated in UML.

NOTE

<Side Remark: pass reference>
The Rational parameter is passed by reference using the syntax &secondRational. This improves performance by preventing the compiler from making a copy of the object being passed into the function.

<Side Remark: lowest term>
A rational number consists of a numerator and a denominator. There are many equivalent rational numbers; for example, 1/3 = 2/6 = 3/9 = 4/12. For convenience, 1/3 is used in this example to represent all rational numbers that are equivalent to 1/3. The numerator and the denominator of 1/3 have no common divisor except 1, so 1/3 is said to be in lowest terms.

To reduce a rational number to its lowest terms, you need to find the greatest common divisor (GCD) of the absolute values of its numerator and denominator, and then divide both numerator and denominator by this value. You can use the function for computing the GCD of two integers n and d, as suggested in Listing 4.5, GreatestCommonDivisor.cpp.
The numerator and denominator in a Rational object are reduced to their lowest terms.

As usual, you can first write a test program to create two Rational objects and test its functions (see the Important Pedagogical Tip on page XXX). Listing 12.1 shows the header file for Rational and Listing 12.2 is a test program.

Listing 12.1 Rational.h (Rational Class Header)

```cpp
#ifndef RATIONL_H
#define RATIONL_H
#include <string>
using namespace std;

class Rational
{
public:
    Rational();
    Rational(long numerator, long denominator);
    long getNumerator();
    long getDenominator();
    Rational add(const Rational &secondRational);
    Rational subtract(const Rational &secondRational);
    Rational multiply(const Rational &secondRational);
    Rational divide(const Rational &secondRational);
    int compareTo(const Rational &secondRational);
    bool equals(const Rational &secondRational);
    int intValue();
    double doubleValue();
    string toString();

private:
    long numerator;
    long denominator;
    static long gcd(long n, long d);
};
#endif
```

Listing 12.2 TestRationalClass.cpp

```cpp
#include "Rational.h"

class Rational
{
public:
    Rational();
    Rational(long numerator, long denominator);
    long getNumerator();
    long getDenominator();
    Rational add(const Rational &secondRational);
    Rational subtract(const Rational &secondRational);
    Rational multiply(const Rational &secondRational);
    Rational divide(const Rational &secondRational);
    int compareTo(const Rational &secondRational);
    bool equals(const Rational &secondRational);
    int intValue();
    double doubleValue();
    string toString();

private:
    long numerator;
    long denominator;
    static long gcd(long n, long d);
};
```
#include <iostream>
#include "Rational.h"
using namespace std;

int main()
{
  // Create and initialize two rational numbers r1 and r2.
  Rational r1(4, 2);
  Rational r2(2, 3);

  // Test toString, add, subtract, multiply, and divide
  cout << r1.toString() << " + " << r2.toString() << " = " << r1.add(r2).toString() << endl;
  cout << r1.toString() << " - " << r2.toString() << " = " << r1.subtract(r2).toString() << endl;
  cout << r1.toString() << " * " << r2.toString() << " = " << r1.multiply(r2).toString() << endl;
  cout << r1.toString() << " / " << r2.toString() << " = " << r1.divide(r2).toString() << endl;

  // Test intValue and doubleValue
  cout << "r2.intValue() is " << r2.intValue() << endl;
  cout << "r2.doubleValue() is " << r2.doubleValue() << endl;

  // Test compareTo and equal
  cout << "r1.compareTo(r2) is " << r1.compareTo(r2) << endl;
  cout << "r2.compareTo(r1) is " << r2.compareTo(r1) << endl;
  cout << "r1.equals(r1) is " << r1.equals(r1) << endl;
  cout << "r1.equals(r2) is " << r1.equals(r2) << endl;

  return 0;
}

<Output>
2 + 2/3 = 8/3
2 - 2/3 = 4/3
2 * 2/3 = 4/3
2 / 2/3 = 3
r2.intValue() is 0
r2.doubleValue() is 0.666667
r1.compareTo(r2) is 1
r2.compareTo(r1) is -1
r1.equals(r1) is 1
r1.equals(r2) is 0

<End Output>

The main function creates two rational numbers, `r1` and `r2` (lines 8-9), and displays the results of `r1 + r2`, `r1 - r2`, `r1 * r2`, and `r1 / r2` (lines 12-19). To perform `r1 + r2`, invoke `r1.add(r2)` to return a new Rational object. Similarly, `r1.subtract(r2)` returns a new Rational object for `r1 - r2`, `r1.multiply(r2)` for `r1 * r2`, and `r1.divide(r2)` for `r1 / r2`.

The `intValue()` function displays the int value of `r1` (line 22). The `doubleValue()` function displays the double value of `r2` (line 23).
Invoking \texttt{r1.compareTo(r2)} (line 26) returns 1 since \texttt{r1} is greater than \texttt{r2}. Invoking \texttt{r2.compareTo(r1)} (line 27) returns -1 since \texttt{r2} is less than \texttt{r1}. Invoking \texttt{r1.compareTo(r1)} (line 28) returns 0 since \texttt{r1} is equal to \texttt{r1}. Invoking \texttt{r1.equals(r1)} (line 29) returns 1 (true) since \texttt{r1} is equal to \texttt{r1}. Invoking \texttt{r1.equals(r2)} (line 30) returns 0 (false) since \texttt{r1} and \texttt{r2} are not equal.

The \texttt{Rational} class is implemented in Listing 12.3.

Listing 12.3 Rational.cpp (The Rational Class)

```cpp
#include "Rational.h"

Rational::Rational()
    : numerator(0),
      denominator(1)
    {

Rational::Rational(long numerator, long denominator)
    : long factor = gcd(numerator, denominator),
        this->numerator = ((denominator > 0) ? 1 : -1) * numerator / factor;
    this->denominator = abs(denominator) / factor;

long Rational::getNumerator()
    { return numerator; }

long Rational::getDenominator()
    { return denominator; }

/** Find GCD of two numbers */
long Rational::gcd(long n, long d)
    { long n1 = abs(n);
```
```cpp
long n2 = abs(d);
  int gcd = 1;
  for (int k = 1; k <= n1 && k <= n2; k++)
    if (n1 % k == 0 && n2 % k == 0)
      gcd = k;
  return gcd;

Rational Rational::add(const Rational &secondRational)
{
  long n = numerator * secondRational.getDenominator() +
           denominator * secondRational.getNumerator();
  long d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}

Rational Rational::subtract(const Rational &secondRational)
{
  long n = numerator * secondRational.getDenominator()
           - denominator * secondRational.getNumerator();
  long d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}

Rational Rational::multiply(const Rational &secondRational)
{
  long n = numerator * secondRational.getNumerator();
  long d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}

Rational Rational::divide(const Rational &secondRational)
{
  long n = numerator * secondRational.getDenominator();
  long d = denominator * secondRational.numerator;
  return Rational(n, d);
}

int Rational::compareTo(const Rational &secondRational)
{
  Rational temp = this->subtract(secondRational);
  if (temp.getNumerator() < 0)
    return -1;
  else if (temp.getNumerator() == 0)
    return 0;
  else
    return 1;
}

bool Rational::equals(const Rational &secondRational)
{
  if (this->compareTo(secondRational) == 0)
    return true;
  else
    return false;
}

int Rational::intValue()
{
  return getNumerator() / getDenominator();
}

double Rational::doubleValue()
{
  return 1.0 * getNumerator() / getDenominator();
}

string Rational::toString()
{
  char s1[20], s2[20];
  itoa(numerator, s1, 10); // Convert int to string s1
  itoa(denominator, s2, 10); // Convert int to string s2
  if (denominator == 1)
```
The rational number is encapsulated in a Rational object. Internally, a rational number is represented in its lowest terms (line 13) and the numerator determines its sign (line 14). The denominator is always positive (line 15).

The \texttt{gcd()} function (lines 29–41 in the \texttt{Rational} class) is private; it is not intended for use by clients. The \texttt{gcd()} function is only for internal use by the Rational class. The \texttt{gcd()} function is also static, since it is not dependent on any particular \texttt{Rational} object.

The \texttt{abs(x)} function (lines 30–31 in the \texttt{Rational} class) is defined in the standard C++ library that returns the absolute value of \( x \).

Two \texttt{Rational} objects can interact with each other to perform add, subtract, multiply, and divide operations. These functions return a new \texttt{Rational} object (lines 43-71).

The \texttt{compareTo(secondRational)} function (lines 73-82) compares this rational number to the other rational number. It first subtracts this rational by the second rational and saves the result in \( \texttt{temp} \) (line 75). Return -1, 0, or 1, if \( \texttt{temp} \)’s numerator is less than, equal to, or greater than 0.

The \texttt{equals(secondRational)} function (lines 84-90) utilizes the \texttt{compareTo} function to compare this rational number to the other rational number. If this function returns 0, the \texttt{equals} function returns true; otherwise, it returns false.

The functions \texttt{intValue} and \texttt{doubleValue} return an \texttt{int} and \texttt{double} value for this rational number (lines 92-100), respectively.

The \texttt{toString()} function (lines 102-112) returns a string representation of a \texttt{Rational} object in the form numerator/denominator, or simply numerator if denominator is 1. The C string function \texttt{itoa} is used to convert the \texttt{int} value numerator and denominator to a C string \( \texttt{s1} \) and \( \texttt{s2} \) (lines 105-106). The C string function \texttt{strcat} (line 111) is used to concatenate C strings into a new C string. This new C string is passed to the constructor of the C++ string class to create a string object. C string functions were introduced in §7.9.4, “String Functions.”

\textbf{TIP}

\textit{<Side Remark: immutable>}

The get functions for the properties numerator and denominator are provided in the \texttt{Rational} class, but the set functions are not provided, so the contents of a Rational object cannot be changed once the object is created. The \texttt{Rational} class is immutable.

\textbf{TIP}

\textit{<Side Remark: encapsulation>}

The numerator and denominator are represented using two variables. It is possible to use an array of two integers to represent the numerator and denominator. See Exercise 12.2. The
signatures of the public functions in the Rational class are not changed, although the internal representation of a rational number is changed. This is a good example to illustrate the idea that the data fields of a class should be kept private so as to encapsulate the implementation of the class from the use of the class.

12.3 Operator Functions
It is convenient to compare two string objects using an intuitive syntax like:

\[ \text{string1 < string2} \]

Can you compare two Rational objects using a similar syntax like

\[ \text{r1 < r2} \]

Yes. You can define a special function called operator function in the class. The function is just like a regular function except that the function must be named with word operator followed by the actual operator. For example, the following function header

bool operator< (const Rational &secondRational)

defines the < operator function that returns true if this Rational object is less than the specified Rational. You can invoke the function using

\[ \text{r1.operator<}(r2) \]

or simply

\[ \text{r1 < r2} \]

To use this operator, you have to add the function header in the Rational.h header file and implement the function in the Rational.cpp file as follows:

***PD: Please add line numbers in the following code***

<Side Remark: line 1: function operator>

<Side Remark: line 3: invoke compareTo>

```cpp
bool Rational::operator<(const Rational &secondRational)
{
    if (this->compareTo(secondRational) < 0)
        return true;
    else
        return false;
}
```

So the following code

```cpp
Rational r1(4, 2);
Rational r2(2, 3);
cout << "r1 < r2 is " << r1.operator<(r2) << endl;
cout << "r1 < r2 is " << (r1 < r2) << endl;
cout << "r2 < r1 is " << r2.operator<(r1) << endl;
```
C++ allows you to overload most of the operators, as shown in Table 12.1. Table 12.2 shows the four operators that cannot be overloaded. C++ does not allow you to create new operators. The only operators you can overload are in Table 12.1.

### Table 12.1
Operators That Can Be Overloaded

| +  | -  | *  | /  | %  | ^  | &  | |  | ~  | !  | - |
| <  | >  | += | -= | *= | /= | %= | ^= | &= | |= | <<= |
| >> | >>= | <<= |

| == | != | <= | >= | && | || | ++ | -- |

### Table 12.2
Operators That Cannot Be Overloaded

?: . .* ::

#### NOTE

**Side Remark: precedence and associativity**

C++ defines the operator precedence and associativity (see §3.12, “Operator Precedence and Associativity.”) You cannot change the operator precedence and associativity by overloading.

#### NOTE

**Side Remark: number of operands**

Most operators are binary operators. Some are unary. You cannot change the number of operands by overloading. For example, the `/` divide operator is binary and `++` is unary.

Here is another example that overloads the addition operator in the `Rational` class.

```cpp
Rational Rational::operator+(const Rational &secondRational)
{
    return this->add(secondRational);
}
```
So the following code

```cpp
class Rational {
  // ...

  Rational operator+=(const Rational &secondRational) {
    *this = this->add(secondRational);
    return (*this);
  }

  // ...}
```

displays

```output
r1 + r2 is 8/3
```

### 12.4 Overloading the Shorthand Operators

C++ has shorthand operators `+=`, `-=`, `*=` and `/=` for adding, subtracting, multiplying, and dividing a value in a variable. You can overload these operators in the `Rational` class.

Here is an example that overloads the addition assignment operator `+=`.

```cpp
Rational r1(2, 4);
Rational r2 = r1 += Rational(2, 3);
cout << "r1 is " << r1.toString() << endl;
cout << "r2 is " << r2.toString() << endl;
```

displays

```output
r1 is 1/2
r2 is 7/6
```

### 12.5 Overloading the [] Operators

In C++, the array subscript `[]` is an operator. You can overload this operator to access the contents of the object using the array-like
syntax if desirable. For example, you may wish to access the numerator and denominator using \texttt{r[0]} and \texttt{r[1]}.

To enable a \texttt{Rational} object to access its numerator and denominator using the array subscript, declare the following function header in the \texttt{Rational.h} header file:

\begin{verbatim}
long operator[](const int &index);
\end{verbatim}

Implement the function in \texttt{Rational.cpp} as follows:

\begin{verbatim}
***PD: Please add line numbers in the following code***
<Side Remark: line 1: [] function operator>
<Side Remark: line 3: access numerator>
<Side Remark: line 5: access denominator>

long Rational::operator[](const int &index)
{
    if (index == 0)
        return numerator;
    else if (index == 1)
        return denominator;
    else
        {
            cout << "subscript error" << endl;
            exit(1);
        }
}
\end{verbatim}

So the following code

\begin{verbatim}
Rational r2(2, 3);
cout << "r2[0] is " << r2[0] << endl;
cout << "r2[1] is " << r2[1] << endl;
\end{verbatim}

displays

\begin{verbatim}
<output>
r2[0] is 2
r2[1] is 3
<output>
\end{verbatim}

Can you set a new numerator or denominator like an array assignment such as

\begin{verbatim}
r2[0] = 5;
r2[1] = 6;
\end{verbatim}

If you compile it, you will get the following error:

\begin{verbatim}
<output>
Lvalue required in function main()
<output>
\end{verbatim}
<Side Remark: Lvalue>
Lvalue (short for left value) means that it can be assigned a value when used on the left side of the assignment operator. How can you assign a value to r2[0] and r2[1]? The answer is to declare the [] operator to return a reference as follows:

<Side Remark: redeclare function header>
long& operator[](const int &index);

So, the following code

***PD: Please add line numbers in the following code***
<Side Remark: line 2: assign to r2[0]>
<Side Remark: line 3: assign to r2[1]>

Rational r2(2, 3);
r2[0] = 5; // Set numerator to 5
r2[1] = 6; // Set denominator to 6
cout << "r2[0] is " << r2[0] << endl;
cout << "r2[1] is " << r2[1] << endl;
cout << "r2.doubleValue() is " << r2.doubleValue() << endl;

displays

<output>
r2[0] is 5
r2[1] is 6
r2.doubleValue() is 0.833333
<output>

In r2[0], r2 is an object and 0 is the argument to the member function [ ]. When r2[0] is used as an expression, it returns a value for the numerator. When r2[0] is used on the left side of the assignment operator, it denotes the address for the numerator. So, r2[0] = 5 is to assign 5 to numerator.

NOTE
<Side Remark: [] accessor and mutator>
The [] operator functions as both accessor and mutator. For example, you use r2[0] as an accessor to retrieve the numerator in an expression, and use r2[0] = value as a mutator. After adding this operator to the Rational class, the Rational class is mutable.

12.6 Overloading the Unary Operators

The + and - are unary operators. They can be overloaded too. Since the unary operator operates on one operand that is the calling object itself, the unary function operator has no parameters.

The - operator in the Rational class can be implemented as follows:

***PD: Please add line numbers in the following code***
Negating a Rational object is same as negating its numerator (line 3). Line 4 returns the calling object.

So, the following code

```cpp
Rational r2(2, 3);
-r2;  // Negate r2
```

displays

```output
r2 is -2/3
```

12.7 Overloading the ++ and -- Operators

The ++ and -- operators may be prefix or postfix. The prefix ++var or --var first adds or subtracts 1 from the variable and then evaluates to the new value in the var. The postfix var++ or var-- adds or subtracts 1 from the variable, but evaluates to the old value in the var.

If the ++ and -- are implemented correctly, the following code

```cpp
Rational r2(2, 3);
Rational r3 = ++r2;  // Prefix increment
cout << "r3 is " << r3.toString() << endl;
cout << "r2 is " << r2.toString() << endl;
Rational r1(2, 3);
Rational r4 = r1++;  // Postfix increment
cout << "r1 is " << r1.toString() << endl;
cout << "r4 is " << r4.toString() << endl;
```

should display

```output
r3 is 5/3
r2 is 5/3
r1 is 5/3
r4 is 2/3
```

r4 stores the original value of r1
How does C++ distinguish the prefix ++ or -- function operators from the postfix ++ or -- function operators? C++ declares postfix ++/-- function operators with a special dummy parameter of the int type and the prefix ++ function operator defined with no parameters as follows:

*Side Remark: prefix ++ operator*

Rational operator++();

*Side Remark: postfix ++ operator*

Rational operator++(int dummy)

These two functions can be implemented as follows:

***PD: Please add line numbers in the following code***

*Side Remark: line 3: \( \frac{a}{b} + 1 = \frac{a+b}{b} \)

*Side Remark: line 4: return calling object*

*Side Remark: line 9: create temp*

*Side Remark: line 10: \( \frac{a}{b} + 1 = \frac{a+b}{b} \)

*Side Remark: line 11: return temp object*

```
Rational Rational::operator++()
{ numeraotr += denominator; return *this; }
```

```
Rational Rational::operator++(int dummy)
{ Rational temp(numerator, denominator); numerator += denominator; return temp; }
```

In the prefix ++ function, line 3 adds the denominator to the numerator. This is the new numerator for the calling object after adding 1 to the Rational object. Line 4 returns the calling object.

In the postfix ++ function, line 9 creates a temporary Rational object to store the original calling object. Line 10 increments the calling object. Line 11 returns the original calling object.

12.8 Overloading the << and >> Operators

Note that you have to invoke the toString() function to return a string representation for a Rational object in order to print to the console. For example,

```
cout << r1.toString();
```

Wouldn’t it be nice to be able to display a Rational object directly using a syntax like

```
cout << r1;
```
C++ allows you to overload the stream insertion operator (<<) for sending an object to cout and overload the stream extraction operator (>>) for reading values from cin. Overloading these two operators is different from other operators you have seen in this chapter. Since the first parameter of the (<< (>>) operator is an instance of ostream (istream), these two operators are defined in the ostream and istream classes, not in the Rational class. However, overloading these two operators requires access to the private members of the Rational class, so you need to declare the function operators as friends of the Rational class in the Rational.h header file:

```
friend ostream &operator<<(ostream &stream, const Rational &rational);
```

Implement this function in Rational.cpp as follows:

```
ostream &operator<<(ostream &str, const Rational &rational)
{
    str << rational.numerator << " / " << rational.denominator;
    return str;
}
```

To overload the >> operator, declare the following function header in the Rational.h header file:

```
friend istream &operator>>(istream &stream, Rational &rational);
```

Implement this function in Rational.cpp as follows:

```
istream &operator>>(istream &str, Rational &rational)
{
    cout << "Enter numerator: ";
    str >> rational.numerator;
    cout << "Enter denominator: ";
    str >> rational[1];
    return str;
}
```

The following code gives a test program that uses the overloaded << and >> functions operators.

***PD: Please add line numbers in the following code***

```cpp
#include <iostream>
#include "Rational.h"
using namespace std;

int main()
{
    Rational r1, r2;
    cout << "Enter first rational number" << endl;
    cin >> r1;
```
cout << "Enter second rational number" << endl;
cin >> r2;

cout << "r1 + r2 is " << r1 + r2 << endl;

return 0;

Enter first rational number
Enter numerator: 1
Enter denominator: 2
Enter second rational number
Enter numerator: 3
Enter denominator: 4
r1 + r2 is 5/4

12.9 Object Conversion
You can add an int value with a double value such as

4 + 5.5

In this case, C++ performs automatic type conversion to convert an int value 4 to a double value 4.0.

Can you add a rational number with an int or a double value? Yes. You have to define a function operator to convert an object into int or double. Here is the implementation of the function to convert a Rational object to a double value.

Rational::operator double()
{
    return doubleValue();
}

Don’t forget that you have to add the member function header in the Rational.h header file.

operator double();

in the Rational.h header file.

NOTE

<Side Remark: conversion function syntax>
This is a special syntax for defining conversion functions in C++. There is no return type. The function name is the type that you want the object to be converted to.

So, the following code

***PD: Please add line numbers in the following code***
<Side Remark: line 2: add rational with double>
Rational r1(1, 4);
double d = r1 + 5.1;
cout << "r1 + 5.1 is " << d << endl;

displays
<output>
r1 + 5.1 is 5.35
<output>

The statement in line 2 adds a rational number r1 with a double value 5.1. Since the conversion function is defined to convert a rational number to a double, r1 is converted to a double value 0.25, which is then added with 5.1.

12.10 The New Rational Class
The preceding sections introduced how to overload function operators. You are ready to give a new Rational class with all appropriate function operators. Listings 12.4 and 12.5 show the new Rational.h and Rational.cpp.

Listing 12.5 NewRational.h (New Rational Header)
***PD: Please add line numbers in the following code***
<Side Remark line 23: relational operators>
<Side Remark line 32: arithmetic operators>
<Side Remark line 38: shorthand operators>
<Side Remark line 44: subscript operator>
<Side Remark line 47: prefix ++ operator>
<Side Remark line 51: postfix ++ operator>
<Side Remark line 55: unary + operator>
<Side Remark line 59: << operator>
<Side Remark line 63: convert to double>

```cpp
#ifndef NEWRATIONL_H
#define NEWRATIONL_H
#include <iostream>
using namespace std;

class Rational
{
public:
    Rational();
    Rational(long numerator, long denominator);
    long getNumerator();
    long getDenominator();
    Rational add(const Rational &secondRational);
    Rational subtract(const Rational &secondRational);
    Rational multiply(const Rational &secondRational);
    Rational divide(const Rational &secondRational);
    int compareTo(const Rational &secondRational);
    bool equals(const Rational &secondRational);
    int intValue();
    double doubleValue();
    string toString();
};
```

461
/** Define function operators for relational operators **/
bool operator<(const Rational &secondRational);
bool operator<=(const Rational &secondRational);
bool operator>(const Rational &secondRational);
bool operator>=(const Rational &secondRational);
bool operator!=(const Rational &secondRational);
bool operator==(const Rational &secondRational);

/** Define function operators for arithmetic operators **/
Rational operator+(const Rational &secondRational);
Rational operator-(const Rational &secondRational);
Rational operator*(const Rational &secondRational);
Rational operator/(const Rational &secondRational);

/** Define function operators for shorthand operators **/
Rational operator+=(const Rational &secondRational);
Rational operator-=(const Rational &secondRational);
Rational operator*=(const Rational &secondRational);
Rational operator/=(const Rational &secondRational);

/** Define function operators for array [] **/
long & operator[](const int &index);

/** Define function operators for prefix ++ and -- **/
Rational operator++();
Rational operator--();

/** Define function operators for postfix ++ and -- **/
Rational operator++(int dummy);
Rational operator--(int dummy);

/** Define function operators for unary + and - **/
Rational operator+();
Rational operator-();

/** Define the output and input operator **/
friend ostream & operator<<(ostream &stream, const Rational &rational);
friend istream & operator>>(istream &stream, Rational &rational);

/** Define function operator for conversion **/
operator double();

private:
   long numerator;
   long denominator;
   static long gcd(long n, long d);
};
#endif

Listing 12.5 NewRational.cpp (New Rational Implementation)
***PD: Please add line numbers in the following code***
<Side Remark line 1: include Rational header>
```cpp
#include "NewRational.h"

Rational::Rational()
{
    numerator = 0;
    denominator = 1;
}

Rational::Rational(long numerator, long denominator)
{
    long factor = gcd(numerator, denominator);
    this->numerator = ((denominator > 0) ? 1 : -1) * numerator / factor;
    this->denominator = abs(denominator) / factor;
}

long Rational::getNumerator()
{
    return numerator;
}

long Rational::getDenominator()
{
    return denominator;
}

/** Find GCD of two numbers */
long Rational::gcd(long n, long d) {
    long n1 = abs(n);
    long n2 = abs(d);
    int gcd = 1;

    for (int k = 1; k <= n1 && k <= n2; k++)
    {
        if (n1 % k == 0 && n2 % k == 0)
        {
            gcd = k;
        }
    }
    return gcd;
}

Rational Rational::add(const Rational &secondRational)
{
    long n = numerator * secondRational.getDenominator() +
             denominator * secondRational.getNumerator();
    long d = denominator * secondRational.getDenominator();
    return Rational(n, d);
}
```
Rational Rational::subtract(const Rational &secondRational) {
    long n = numerator * secondRational.getDenominator()
        - denominator * secondRational.getNumerator();
    long d = denominator * secondRational.getDenominator();
    return Rational(n, d);
}

Rational Rational::multiply(const Rational &secondRational) {
    long n = numerator * secondRational.getNumerator();
    long d = denominator * secondRational.getDenominator();
    return Rational(n, d);
}

Rational Rational::divide(const Rational &secondRational) {
    long n = numerator * secondRational.getDenominator();
    long d = denominator * secondRational.getNumerator();
    return Rational(n, d);
}

int Rational::compareTo(const Rational &secondRational) {
    Rational temp = this->subtract(secondRational);
    if (temp.getNumerator() < 0)
        return -1;
    else if (temp.getNumerator() == 0)
        return 0;
    else
        return 1;
}

bool Rational::equals(const Rational &secondRational) {
    if (this->compareTo(secondRational) == 0)
        return true;
    else
        return false;
}

int Rational::intValue() {
    return getNumerator() / getDenominator();
}

double Rational::doubleValue() {
    return 1.0 * getNumerator() / getDenominator();
}

string Rational::toString()
```c
char s1[20], s2[20];
itoa(numerator, s1, 10); // Convert int to string s1
itoa(denominator, s2, 10); // Convert int to string s2

if (denominator == 1)
    return string(s1);
else
    return string(strcat(strcat(s1, "/"), s2));

// Define function operators for relational operators
bool Rational::operator<(const Rational &secondRational)
{
    return (this->compareTo(secondRational) < 0);
}

bool Rational::operator<=(const Rational &secondRational)
{
    return (this->compareTo(secondRational) <= 0);
}

bool Rational::operator>(const Rational &secondRational)
{
    return (this->compareTo(secondRational) > 0);
}

bool Rational::operator>=(const Rational &secondRational)
{
    return (this->compareTo(secondRational) >= 0);
}

bool Rational::operator!=(const Rational &secondRational)
{
    return (this->compareTo(secondRational) != 0);
}

bool Rational::operator==(const Rational &secondRational)
{
    return (this->compareTo(secondRational) == 0);
}

// Define function operators for arithmetic operators
Rational Rational::operator+(const Rational &secondRational)
{
    return this->add(secondRational);
}

Rational Rational::operator-(const Rational &secondRational)
{
    return this->subtract(secondRational);
}

Rational Rational::operator*(const Rational &secondRational)
{
    return this->multiply(secondRational);
}

Rational Rational::operator/(const Rational &secondRational)
{
    return this->divide(secondRational);
}
```
return this->multiply(secondRational);

Rational Rational::operator/(const Rational &secondRational)
{
    return this->divide(secondRational);
}

// Define function operators for shorthand operators
Rational Rational::operator+=(const Rational &secondRational)
{
    *this = this->add(secondRational);
    return (*this);
}

Rational Rational::operator-=(const Rational &secondRational)
{
    *this = this->subtract(secondRational);
    return (*this);
}

Rational Rational::operator*=(const Rational &secondRational)
{
    *this = this->multiply(secondRational);
    return (*this);
}

Rational Rational::operator/=(const Rational &secondRational)
{
    *this = this->divide(secondRational);
    return *this;
}

// Define function operator []
long& Rational::operator[](const int &index)
{
    if (index == 0)
        return numerator;
    else if (index == 1)
        return denominator;
    else
    {
        cout << "subscript error" << endl;
        exit(0);
    }
}

// Define function operators for prefix ++ and --
Rational Rational::operator++()
{
    numerator += denominator;
    return *this;
}

Rational Rational::operator--()
numerator -= denominator;
return *this;

// Define function operators for postfix ++ and --
Rational Rational::operator++(int dummy)
{
    Rational temp(numerator, denominator);
    numerator += denominator;
    return temp;
}

Rational Rational::operator--(int dummy)
{
    Rational temp(numerator, denominator);
    numerator -= denominator;
    return temp;
}

// Define function operators for unary + and -
Rational Rational::operator+()
{
    return *this;
}

Rational Rational::operator-()
{
    numerator *= -1;
    return *this;
}

// Define the output and input operator
ostream &operator<<(ostream &str, const Rational &rational)
{
    str << rational.numerator << " / " << rational.denominator;
    return str;
}

istream &operator>>(istream &str, Rational &rational)
{
    cout << "Enter numerator: ";
    str >> rational.numerator;
    cout << "Enter denominator: ";
    str >> rational[1];
    return str;
}

// Define function operator for conversion
Rational::operator double()
{
    return doubleValue();
}
Listing 12.6 gives a program for testing the new Rational class.

```cpp
#include <iostream>
#include <string>
#include "NewRational.h"
using namespace std;

int main()
{
    // Create and initialize two rational numbers r1 and r2.
    Rational r1(4, 2);
    Rational r2(2, 3);

    // Test relational operators
    cout << r1 << " > " << r2 << " is " << (r1 > r2) << endl;
    cout << r1 << " < " << r2 << " is " << (r1 < r2) << endl;
    cout << r1 << " == " << r2 << " is " << (r1 == r2) << endl;
    cout << r1 << " != " << r2 << " is " << (r1 != r2) << endl;

    // TesttoString, add, subtract, multiply, and divide operators
    cout << r1 << " + " << r2 << " = " << r1 + r2 << endl;
    cout << r1 << " - " << r2 << " = " << r1 - r2 << endl;
    cout << r1 << " * " << r2 << " = " << r1 * r2 << endl;
    cout << r1 << " / " << r2 << " = " << r1 / r2 << endl;

    // Test shorthand operators
    Rational r3(1, 2);
    r3 += r1;
    cout << "r3 is " << r3 << endl;

    // Test function operator []
    Rational r4(1, 2);
    r4[0] = 3; r4[1] = 4;
    cout << "r4 is " << r4 << endl;

    // Test function operators for prefix ++ and --
    r3 = r4++;
    cout << "r3 is " << r3 << endl;
    cout << "r4 is " << r4 << endl;

    // Test function operator for conversion
    cout << "1 + " << r4 << " is " << (1 + r4) << endl;

    return 0;
}
```

<Output>

```
2 / 1 > 2 / 3 is 1
2 / 1 < 2 / 3 is 0
2 / 1 == 2 / 3 is 0
2 / 1 != 2 / 3 is 1
2 / 1 + 2 / 3 = 2.66667
2 / 1 - 2 / 3 = 1.33333
2 / 1 * 2 / 3 = 1.33333
2 / 1 / 2 / 3 = 3
r3 is 5 / 2
r4 is 3 / 4
```
r3 is 3 / 4
r4 is 7 / 4
1 + 7 / 4 is 2.75

<End Output>

12.11 Overloading the = Operators

By default, the = operator performs a memberwise copy from one object to the other. For example, the following code copies r2 to r1.

***PD: Please add line numbers in the following code***

.Side Remark: line 3: copy r2 to r1>

```cpp
Rational r1(1, 2);
Rational r2(4, 5);
r1 = r2;
cout << "r1 is " << r1.toString() << endl;
cout << "r2 is " << r2.toString() << endl;
```

So, the output is

<output>

r1 is 4/5
r2 is 4/5
<output>

.Side Remark: shallow copy>

The behavior of the = operator is same as the default copy constructor. It performs a shallow copy, meaning that if the data field is a pointer to some object, the address of the pointer is copied rather than its contents. In §12.7, “Customizing Copy Constructors,” you learned how to customize the copy constructor to perform a deep copy. However, customizing the copy constructor does not change the default behavior of the assignment copy operator =. For example, the Person class defined in Figure 11.1 (in Chapter 11) has a pointer data field named birthDate which points to a Date object. The Date class is defined in Figure 11.2. If you run the following code using the assignment operator to assign person2 to person1, as shown in line 23 in Listing 12.7, both person1 and person2 point to the same birthDate, as shown in Figure 11.5.

Listing 12.7 DefaultAssignmentDemo.cpp (Copying Object)

***PD: Please add line numbers in the following code***

.Side Remark line 2: include Person header>
.Side Remark line 7: display person1>
.Side Remark line 9: display year>
.Side Remark line 10: display person2>
.Side Remark line 12: display year>
.Side Remark line 17: create person1>
.Side Remark line 18: create person2>
.Side Remark line 23: copy person2 to person1>
.Side Remark line 28: modify person2’s year>

```cpp
#include <iostream>
#include "Person.h"
using namespace std;
```
void displayPerson(Person &person1, Person &person2)
{
    cout << "\tt\person1 id: " << person1.getId() << endl;
    cout << "\tt\person1 birth year: " <<
            person1.getBirthDate() -> getYear() << endl;
    cout << "\tt\person2 id: " << person2.getId() << endl;
    cout << "\tt\person2 birth year: " <<
            person2.getBirthDate() -> getYear() << endl;
}

int main()
{
    Person person1(111, 1970, 5, 3);
    Person person2(222, 2000, 11, 8);
    cout << "After creating person1 and person2" << endl;
    displayPerson(person1, person2);
    
    person1 = person2; // Copy person2 to person1
    cout << "\nAfter copying person2 to person1" << endl;
    displayPerson(person1, person2);

    person2.getBirthDate() -> setYear(1963);
    cout << "\nAfter modifying person2's birthDate" << endl;
    displayPerson(person1, person2);

    cout << "\n" << (person1.getBirthDate() == person2.getBirthDate());
    return 0;
}

<Output>
After creating person1 and person2
person1 id: 111
person1 birth year: 1970
person2 id: 222
person2 birth year: 2000

After copying person2 to person1
person1 id: 222
person1 birth year: 2000
person2 id: 222
person2 birth year: 2000

After modifying person2's birthDate
person1 id: 222
person1 birth year: 1963
person2 id: 222
person2 birth year: 1963

1
<End Output>

To change the way the default assignment operator = works, you need to overload the = operator. In the Person.h file, define
const Person operator=(const Person &person)

Why the return type is Person not void? C++ allows multiple assignments such as:

```
person1 = person2 = person3;
```

In this statement, person3 is copied to person2, and then returns person2, and then person2 is copied to person1. So the = operator must have a valid return value type.

In the Person.cpp, implement the function as follows:

```cpp
const Person operator=(const Person &person) {
    id = person.id;
    Date *p = person.getBirthDate();
    birthDate = new Date(*p); // Create a new Date object
    return *this;
}
```

Line 6 returns the calling object using *this. Note that this is the pointer to the calling object, so *this refers to the calling object.

If you run Listing 12.7 now, person1 and person2 will have their independent Date objects for birthDate.

Chapter Summary

- C++ allows you to overload operators to simplify operations for objects.

- You can overload nearly all operators except ?, ., .*, and ::. You cannot change the operator precedence and associativity by overloading.

- In C++, the array subscript [] is an operator. You can overload this operator to access the contents of the object using the array-like syntax if desirable.

- You can overload the prefix and postfix ++ and --.

- You can overload the << and >> operators for input and output.

- You implement a cast operator to convert an object to a primitive type value.

Review Questions
Section 12.3 Operator Functions

12.1 How do you define an operator function for overloading an operator?

12.2 List the operators that cannot be overloaded?

12.3 Can you change the operator precedence or associativity by overloading?

Section 12.4 Overloading the Shorthand Operators

12.4 When you overload a shorthand operator such as +=, should the function be void or nonvoid?

Section 12.5 Overloading the [] Operators

12.5 What should be function signature for the [] operator?

Section 12.6 Overloading the Unary Operators

12.6 What should be function signature for the unary + operator?

Section 12.7 Overloading the ++ and -- Operators

12.7 What should be function signature for the prefix ++ operator? What should be function signature for the postfix ++ operator?

Section 12.8 Overloading the << and >> Operators

12.8 What should be function signature for the << operator? What should be function signature for the >> operator?

12.9 If you overload the << operator as follows, do you still need to declare it function operator friend in the Rational class?

```cpp
ostream &operator<<(ostream &str, const Rational &rational)
{
    str << rational.getNumerator() << " / " << rational.getDenominator();
    return str;
}
```

Section 12.9 Object Conversion
12.10
What should be function signature to convert an object to the **int** type?

**Programming Exercises**

12.1
**Using the Rational class** Write a program that will compute the following summation series using the **Rational** class:

\[
\frac{1}{2} + \frac{2}{3} + \frac{3}{4} + \ldots + \frac{98}{99} + \frac{99}{100}
\]

12.2*
**Demonstrating the benefits of encapsulation** Rewrite the **Rational** class in §12.2 using a new internal representation for numerator and denominator. Declare an array of two integers as follows:

```cpp
long r[2];
```

Use `r[0]` to represent the numerator and `r[1]` to represent the denominator. The signatures of the functions in the **Rational** class are not changed, so a client application that uses the previous **Rational** class can continue to use this new **Rational** class without any modifications.

12.3*
**The Circle class** Implement the relational operators in the **Circle** class to order the **Circle** objects according to their radii.

12.4*
**The StackOfIntegers class** §11.11 defined the **StackOfIntegers** class. Implement the subscript operator `[]` in this class to access the elements via the `[]` operator.