PROGRAMMING WITH C++

LAB 12

Xi'an Jiaotong-Liverpool University
西交利物浦大学

Example 1: The Rational Class

This example defines the **Rational** class for modelling rational numbers.

C++ provides data types for integers and floating-point numbers but not for **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, multiply, 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 as

Requirements:

Please write a **rational number class** that has a numerator and a denominator in the form a/b, where a is the numerator and b is the denominator.
The **rational class** can add, subtract, multiply, divide, and compare rational numbers.
You can also convert a rational number into an integer, floating-point value, or string.

Example 1: The Rational Class UML

Rational	
-numerator: int	The numerator of this rational number.
-denominator: int	The denominator of this rational number.
+Rational()	Creates a rational number with numerator 0 and denominator 1.
+Rational(numerator: int, denominator: int)	Creates a rational number with specified numerator and denominator.
+getNumerator(): int const	Returns the numerator of this rational number.
+getDenominator(): int const	Returns the denominator of this rational number.
+add(secondRational: Rational&): Rational const	Returns the addition of this rational with another.
+subtract(secondRational: Rational&): Rational const	Returns the subtraction of this rational with another.
+multiply(secondRational: Rational&): Rational const	Returns the multiplication of this rational with another.
+divide(secondRational: Rational&): Rational const	Returns the division of this rational with another.
+compareTo(secondRational: Rational&): int const	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.
+equals(secondRational: Rational&): bool const	Returns true if this rational number is equal to the specified number.
+intValue(): int const	Returns the numerator / denominator.
+doubleValue(): double const	Returns the 1.0 * numerator / denominator.
+toString(): string const	Returns a string in the form "numerator/denominator." Returns numerator if denominator is 1.
-gcd(n: int, d: int): int	Returns the greatest common divisor between n and d.

TestRationalClass.cpp

```
#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 double
  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.compareTo(r1) is " << r1.compareTo(r1) << endl;
  cout << "r1.equals(r1) is "
      << (r1.equals(r1) ? "true" : "false") << endl;
  cout << "r1.equals(r2) is "
      << (r1.equals(r2) ? "true" : "false") << endl;
  return 0;
}
```

Rational.h

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

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

private:
  int numerator;
  int denominator;
  static int gcd(int n, int d);
};

#endif
```

Rational.cpp (1)

```
#include "Rational.h"
#include <sstream> // Used in toString to
convert numbers to strings

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

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

int Rational::getNumerator() const
{
  return numerator;
}

int Rational::getDenominator() const
{
  return denominator;
}
```

```
// Find GCD of two numbers
int Rational::gcd(int n, int d)
{
  int n1 = abs(n);
  int 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) const
{
  int n = numerator * secondRational.getDenominator() +
    denominator * secondRational.getNumerator();
  int d = denominator * secondRational.getDenominator();
  return Rational(n, d);
}

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

Rational.cpp (2)

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

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

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

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

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

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

string Rational::toString() const
{
  stringstream ss;
  ss << numerator;

  if (denominator > 1)
    ss << "/" << denominator;

  return ss.str();
}
```

Example 2: Friend Functions and Classes

Private members of a class cannot be accessed from outside of the class.

Occasionally, it is convenient to allow some trusted functions and classes to access a class's private members.

C++ enables you to use the friend keyword to declare friend functions and friend classes for a class so these functions and classes can access the class's private members.

Example 2-1： Friend Classes

Date.h

```
#ifndef DATE_H
#define DATE_H
class Date
{
public:
  Date(int year, int month, int day)
  {
    this->year = year;
    this->month = month;
    this->day = day;
  }

  friend class AccessDate;

private:
  int year;
  int month;
  int day;
};

#endif
```

TestFriendClass.cpp

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

class AccessDate
{
public:
  static void p()
  {
    Date birthDate(2010, 3, 4);
    birthDate.year = 2000;
    cout << birthDate.year << endl;
  }
};

int main()
{
  AccessDate::p();

  return 0;
}
```

Example 2-2：Friend Functions

TestFriendFunction.cpp

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

class Date
{
public:
  Date(int year, int month, int day)
  {
    this->year = year;
    this->month = month;
    this->day = day;
  }
  friend void p();

private:
  int year;
  int month;
  int day;
};

void p()
{
  Date date(2010, 5, 9);
  date.year = 2000;
  cout << date.year << endl;
}

int main()
{
  p();

  return 0;
}
```

```
huakangleedeMacBook-Pro-7:Downloads huakanglee$ g++ -o TestFriendFunction Tes
tFriendFunction.cpp
huakangleedeMacBook-Pro-7:Downloads huakanglee$ ./TestFriendFunction
2000
```

Example 3: The New Rational Class with Overloading Operators

Operator Functions

```
bool Rational::operator<(Rational &secondRational)
{
  return compareTo(secondRational) < 0
}

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

Example 3: The New Rational Class with Overloading Operators

Overloadable Operators

```
+       -       *       /       %       ^       &       |       ~       !       =
<       >       +=      -=      *=      /=      %=      ^=      &=      |=      <<
>>      >>=     <<=     ==      !=      <=      >=      &&      ||      ++      --
->*     ,       ->      []      ()      new     delete
```

Operators that Cannot be Overloaded

```
?:      .       .*      ::
```

Example 3: The New Rational Class with Overloading Operators

The sections in the lecture introduced how to overload function operators. You are ready to give a new Rational class with all appropriate function operators.

RationalWithOperators.h

RationalWithOperators.cpp

TestRationalWithOperators.cpp

TestRationalWithOperators.cpp

```
#include <iostream>
#include <string>
#include "RationalWithOperators.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) ? "true" : "false") << endl;
  cout << r1 << " < " << r2 << " is " <<
    ((r1 < r2) ? "true" : "false") << endl;
  cout << r1 << " == " << r2 << " is " <<
    ((r1 == r2) ? "true" : "false") << endl;
  cout << r1 << " != " << r2 << " is " <<
    ((r1 != r2) ? "true" : "false") << endl;
  // Test toString, 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 augmented 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;
}
```

```
huakangleedeMacBook-Pro-7:Downloads huakanglee$ g++ -o TestRationalWithOperat
ors TestRationalWithOperators.cpp RationalWithOperators.cpp
huakangleedeMacBook-Pro-7:Downloads huakanglee$ ./TestRationalWithOperators
2 > 2/3 is true
2 < 2/3 is false
2 == 2/3 is false
2 != 2/3 is true
2 + 2/3 = 8/3
2 - 2/3 = 4/3
2 * 2/3 = 4/3
2 / 2/3 = 3
r3 is 5/2
r4 is 3/4
r3 is 3/4
r4 is 7/4
1 + 7/4 is 11/4
```

Example 4: Overloading the = Operator

By default, the = operator performs a memberwise copy from one object to the other. The behavior of the = operator is the same as that of 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 Chapter 11, "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 =.

To change the way the default assignment operator = works, you need to overload the = operator. In the Couse.h file, define

```
const Course& operator=(const Course& course)
```

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

```
course1 = course2 = course3;
```

In this statement, course3 is copied to course2, and then returns course2, and then course2 is copied to course1. So the = operator must have a valid return value type.

CourseWithEqualsOperatorOverloaded.h

```
class Course
{
public:
  Course(const string &courseName, int capacity);
  ~Course(); // Destructor
  Course(Course&); // Copy constructor
  string getCourseName() const;
  void addStudent(const string& name);
  void dropStudent(const string& name);
  string* getStudents() const;
  int getNumberOfStudents() const;
  const Course& operator=(const Course& course);

private:
  string courseName;
  string* students;
  int numberOfStudents;
  int capacity;
};
```

CourseWithEqualsOperatorOverloaded.cpp

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

Course::Course(const string &courseName, int capacity)
{
  numberOfStudents = 0;
  this->courseName = courseName;
  this->capacity = capacity;
  students = new string[capacity];
}

Course::~Course()
{
  delete [] students;
}

string Course::getCourseName() const
{
  return courseName;
}
void Course::addStudent(const string &name)
{
  if (numberOfStudents >= capacity)
  {
    cout << "The maximum size of array exceeded" << endl;
    cout << "Program terminates now" << endl;
    exit(0);
  }

  students[numberOfStudents] = name;
  numberOfStudents++;
}

void Course::dropStudent(const string &name)
{
  // Left as an exercise
}

string* Course::getStudents() const
{
  return students;
}

int Course::getNumberOfStudents() const
{
  return numberOfStudents;
}

Course::Course(Course& course) // Copy constructor
{
  courseName = course.courseName;
  numberOfStudents = course.numberOfStudents;
  capacity = course.capacity;
  students = new string[capacity];
}

const Course& Course::operator=(const Course& course)
{
  courseName = course.courseName;
  numberOfStudents = course.numberOfStudents;
  capacity = course.capacity;
  students = new string[capacity];

  return *this;
}
```

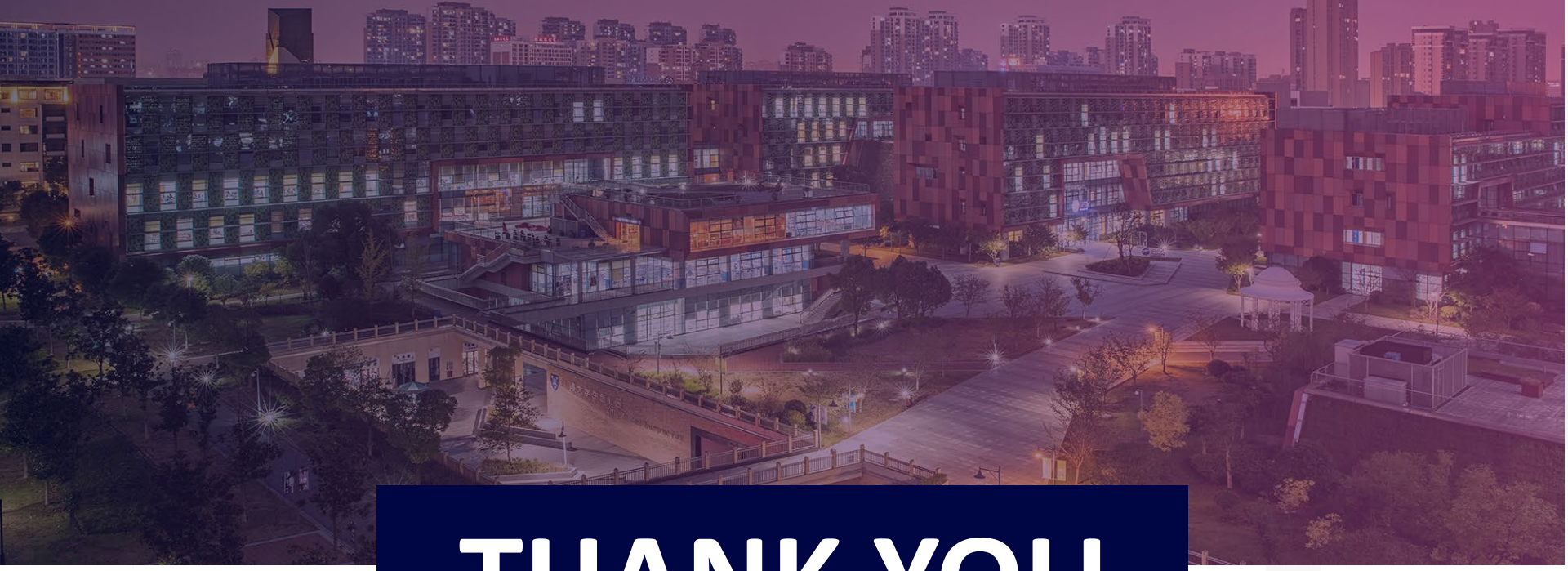
CustomAssignmentDemo.cpp

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

void printStudent(string names[], int size)
{
  for (int i = 0; i < size; i++)
    cout << names[i] << (i < size - 1 ? ", " : " ");
}

int main()
{
  Course course1("Java Proramming", 10);
  course1.addStudent("Peter Pan"); // Add a student to
course1

  Course course2("C++ Programming", 10);
  course2 = course1; // Create course2 as a copy of
course1
  course2.addStudent("Lisa Ma"); // Add a student to
course2
  cout << "students in course1: ";
  printStudent(course1.getStudents(),
course1.getNumberOfStudents());
  cout << endl;

  cout << "students in course2: ";
  printStudent(course2.getStudents(),
course2.getNumberOfStudents());
  cout << endl;

  return 0;
}
```

CourseWithEqualsOperatorOverloaded.h

CourseWithEqualsOperatorOverloaded.cpp

```
[huakangleedeMacBook-Pro-7:Downloads huakanglee$ g++ -o CustomAssignmentDemo C
 ustomAssignmentDemo.cpp CourseWithAssignmentOperatorOverloaded.cpp
[huakangleedeMacBook-Pro-7:Downloads huakanglee$ ./CustomAssignmentDemo
 students in course1: Peter Pan
 students in course2: Peter Pan, Lisa Ma
```



THANK YOU

VISIT US

WWW.XJTLU.EDU.CN

FOLLOW US

Xi'an Jiaotong-Liverpool University
西交利物浦大学