Chapter 8 (Part 2)

Destructor & Operator Overloading

(P.331)

Sample Code of CRational Addition

```
#include <iostream>
using namespace std;                // See P.83
```

```
class CRational
{       public:
            int numerator;          // 分子
            int denominator;        // 分母
            CRational(int p=0, int q=1)     : numerator(p), denominator(q)
            {
                if (q==0)
                {
                    cout << "Error! The denominator cannot be 0.\n";
                    exit(1);
                }
            }
            void Print()    //印出分子&分母
            {
                cout << numerator;
                if (denominator != 1)
                    cout << "/" << denominator;
            }
            void Add(CRational a, CRational b) //將兩分數相加
            {
                int c;
                int d;
                c = a.numerator*b.denominator+a.denominator*b.numerator;
                d = b.denominator*a.denominator;
                cout << c;                  // Simply print out, so you cannot
                if(d != 1)                  // add three rational numbers
                    cout << "/" << d ;
            }
};
```

Another Example

```
class CRational {
    public:
    int numerator;              // 分子
    int denominator;            // 分母
    CRational(int p=0, int q=1){
        numerator=p;
        denominator=q;
     }

 CRational Add(CRational data_a,CRational data_b){
     numerator = data_a.numerator*data_b.denominator+
                    data_b.numerator*data_a.denominator;
     denominator = data_a.denominator*data_b.denominator;
     }
};
```

Call the member function from main()

```
int main(){
    int input_n,input_d;
    while(1){
        printf("please input two Rational\n");
        printf("the a's numerator is:");
        scanf("%d",&input_n);
        printf("the a's denominator is:");
        scanf("%d",&input_d);
        CRational a(input_n,(input_d==0?1:input_d));

        printf("the b's numerator is:");
        scanf("%d",&input_n);
        printf("the b's denominator is:");
        scanf("%d",&input_d);
        CRational b(input_n,(input_d==0?1:input_d));
        CRational c;
        c.Add(a,b);
        c.Print();
    }
    return 0;
}
```

c = a + b
This is good,
but …

You should handle this in the constructor, not here for every input.

How do we calculate d = a + b + c?

- For c = a + b:
  - c.Add(a, b);
- For d = a + b + c:
  - temp.Add(a,b);
  - d.Add(temp,c);
- For e = a + b + c + d:
  - temp1.Add(a, b);
  - temp2.Add(c, d);
  - e.Add(temp1, temp2);
- After we declare a class CRational, it will be more natural if we can simply write `c = a + b` instead of `c.Add(a,b)`. Can we do that?

Operator Overloading

- **Operator overloading** is a very convenient capability.
  - It allows you to make standard C++ operators, such as +, -, * and so on, work with objects of your own data types.
  - We want to write
    - d = a + b
  - instead of
    - d = a.Add(b)
  - We want to write
    - d = a + b + c;
  - instead of
    - d = a.Add(b.Add(c));

Implementing an Overloaded Operator

- Suppose you already have a member function:

```
CRational Add(const CRational& b) {
    return CRational(numerator*b.denominator +
          denominator*b.numerator,
          denominator * b.denominator);
}
```

- You only need to add another member function:

```
CRational operator+(const CRational& b) {
    return Add(b);
}
```

The word `operator` here is a keyword.

Using an Overloaded Operator

- CRational a(2, 6);
- CRational b(1, 3);
- CRational c = a + b;

- c = a.operator+(b);

- d = a + b + c;
- d = a.operator+(b.operator+(c))

Hands-on: Operator Overloading

- Enhance the class of rational numbers, by defining operator*() to support Operator Overloading.
- You may test your class with the following main program:

```
#include "rational.h"
int main() {
    CRational a(1, 2);
    CRational b(2, 3);
    CRational c = a + b; c.Print();
    c = a.Mul(b);
    c = a * b; c.Print();
    return 0;
}
```

Ex8_03.cpp on P.334

```
bool CBox::operator> (CBox& aBox) const
      {
            return this->Volume() > aBox.Volume();
      }
```

- The left operand is defined implicitly by the pointer **this**.
  - Q: Can we omit "`this->`"?
- The basic > operator returns a value of type int
  - 1 for true (P.56, non-zero value as true)
  - 0 for false.
- It will be automatically converted to `bool`.

Overloading the Assignment Operator

- What's wrong with the default assignment?
  - It simply provides a member-by-member copying process, similar to that of the default copy constructor.
  - They suffer from the same problem (P.442), when some data members are allocated dynamically.

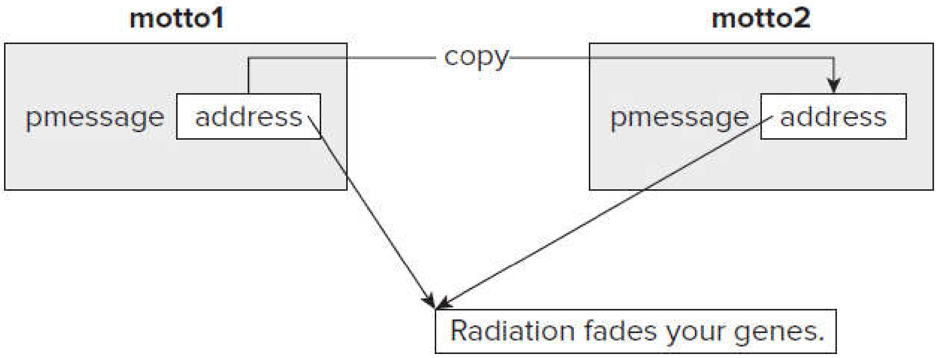

FIGURE 8-1
(P.442)

Fixing the Problem

```
CMessage& operator= (const CMessage& aMess)
{ // buffer = string1;
  // Release memory for 1st operand
  delete [] pmessage;
  pmessage = new char [ strlen(aMess.pmessage) + 1];

  // Copy 2nd operand string to 1st
  strcpy(this->pmessage, aMess.pmessage);

  // Return a reference to 1st operand
  return *this;
}
```

Why Do You Need to Return Something?

- Consider this statement
  - motto1 = motto2 = motto3;
- The assignment operator is right-associative, so it translates into
  - motto1 = (motto2.operator=(motto3));
  - motto1.operator=(motto2.operator=(motto3));
- You must at least return a CMessage object.

Why Do You Need to Return a Reference?

- Consider another example
  - (motto1 = motto2) = motto3;
- This translates into
  - (motto1.operator=(motto2))   = motto3;
- If the return type is merely CMessage instead of a reference, a temporary copy of the original object is returned.
  - Then you are assigning a value to a temporary object!
  - Make sure that your return type is CMessage&.

Check Addresses, If Equal

- The first thing that the operator function does is to delete the memory allocated to the first object (See P.340 again), and reallocate sufficient memory to accommodate the new string.
- What happens to this statement?
  - `motto1 = motto1`
- Add this checking: (P.342)

```
if (this == &aMess)
      return *this;
```

Overloading the Addition Operator

- Suppose we define the sum of two CBox object as a CBox object which is large enough to contain the other two boxes stacked on top of each other.
- See Figure 8-3 (P.345).

```
CBox CBox::operator+(const CBox& aBox) const
{
return CBox(
m_Length > aBox.m_Length ? m_Length : aBox.m_Length,
m_Width > aBox.m_Width ? m_Width : aBox.m_Width,
m_Height + aBox.m_Height);
}
```

- Ex8_06.cpp on P.346

2 Versions of Operator Overloading

- To define operator+() for class CRational, we have two possibilities:
  1. Define it as a member function CRational.
     - CRational operator+(const CRational& B)
  2. Define it as a standalone function which takes two parameters.
     - CRational operator+(const CRational& A, const CRational& B)

- Let’s see the following sample code:

The Same main()

```
int main()
{
    CRational a(1, 2);
    CRational b(1, 3);
    CRational c = a + b;
    c.Print();
    return 0;
}
```

Member Function

```
class CRational {
public:
    CRational(int n = 0, int d = 1): m_numerator(n), m_denominator(d) {}
    void Print() { std::cout << m_numerator << '/' << m_denominator << std::endl; }

    CRational operator+(const CRational& B) {
        CRational c;
        c.m_numerator = m_numerator*B.m_denominator + m_denominator*B.m_numerator;
        c.m_denominator = m_denominator*B.m_denominator;
        return c;
    }

    int m_numerator;
    int m_denominator;
};
```

Standalone Function

```
class CRational {
public:
    CRational(int n = 0, int d = 1): m_numerator(n), m_denominator(d) {}
    void Print() { std::cout << m_numerator << '/' << m_denominator << std::endl; }

    int m_numerator;
    int m_denominator;
};
```

To make the code simple, these data members are public.

```
CRational operator+(const CRational& A, const CRational& B) {
    CRational c;
    c.m_numerator = A.m_numerator*B.m_denominator + A.m_denominator*B.m_numerator;
    c.m_denominator = A.m_denominator*B.m_denominator;
    return c;
}
```

Friend Functions of a Class

- Sometime, for some reason, you want certain selected functions that are not members of a class to be able to access data members of a class.
- Such functions are called friend functions of a class, and are defined using the keyword `friend`.
- In the above two examples, if the data members are private, the standalone function must be declared as a friend.
  - On the contrary, the member function can always access private/public members.

Standalone Function

```
class CRational {
friend CRational operator+(const CRational& A, const CRational& B);

public:
    CRational(int n = 0, int d = 1): m_numerator(n), m_denominator(d) {}
    void Print() { std::cout << m_numerator << '/' << m_denominator << std::endl; }

private:
    int m_numerator;
    int m_denominator;
};
```

You may also declare main() as a friend: `friend int main();`

```
CRational operator+(const CRational& A, const CRational& B) {
    CRational c;
    c.m_numerator = A.m_numerator*B.m_denominator + A.m_denominator*B.m_numerator;
    c.m_denominator = A.m_denominator*B.m_denominator;
    return c;
}
```

cout << CRational

- The statement
  - `cout << b`
- is essentially
  - `operator<<(cout, b)`

Must be declared as a friend function to access private members.

```
friend ostream& operator<<(ostream& output, const CRational& b)
{
    output << b.m_numerator;
    if (b.m_denominator > 1)
        output << '/' << b.m_denominator;
    return output;
}
```

Sample Code

```
#include <iostream>
using std::cout;
using std::endl;
using std::ostream;

class CRational {
friend ostream& operator<<(ostream& output, const CRational& b);
public:
    CRational(int n=0, int d=1): numerator(n), denominator(d) {}
    CRational Add(const CRational& b) {
        return CRational(numerator*b.denominator + denominator*b.numerator,
                denominator*b.denominator);
    }
    CRational operator+(const CRational& b) {
        return this->Add(b);
    }
private:
    int numerator;
    int denominator;
};
```

Sample Code (cont.)

```
int main()
{
    CRational a(1, 2);
    CRational b(1, 3);
    CRational c = a + b;
    cout << c << endl;
    return 0;
}

ostream& operator<<(ostream& output, const CRational& b) {
    output << b.numerator;
    if (b.denominator > 1)
        output << '/' << b.denominator;
    return output;
}
```

Exercise: cin >> CRational

- Design a friend function operator>>() for your CRational so that your program can "cin >> a;" to read a rational number.
- Suppose the input rational number always contains both the numerator and the denominator. For example, "1/2" and "3/1".
- You may test your class with the following main program:

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
using std::istream;
using std::ostream;

#include "rational.h"

int main()
{
    CRational a, b, c;
    cin >> a >> b;
    c = a + b;
    cout << c << endl;
    return 0;
}
```