CISC 2000 — Computer Science II
Fall, 2014

Note 11/5/2014

1 Review of last class

(a) Modular Design: Class interface file (such as rational.h), class implementation file (such as rational.cpp), and driver/test/main file (main.cpp that makes use of the rational class).

(b) Information hiding: user of a class (such as main.cpp) does not need to know the impelmentation details of the class.

(c) static member variable: all objects of the class share a copy

(d) usage of keyword const: to denote something that cannot be modified...

(e) I/O and file stream: read from and write to disk files, in a way similar to read from keyboard and write to terminal

(f) friend function of a class: example, IsEqual() that compares two rational objects

```
class rational{
public:
   // Usually list all friend functions at the top of public section:
   friend bool IsEqualTo (const rational & n1, const rational &n2);
    ...
};

bool IsEqualTo (cosnt rational & n1, const rational & n2)
{
  return (n1.numerator*n2.denominator == n2.numerator*n1.denominator);
}
```

2 Operator overload

We can overload operators (see appendix for the list) on our class, so that one can compare two objects of our class type, assign one to another, perform numerical operations on objects (addition, subtraction, ...). Furthermore, if we override input/output operators (also called extraction/insertion operators, we can read and write objects similar to reading and writing built-in types.

```
rational a,b;

cin >> a; //If we overload operator >>
cout <<"a's value is" << a << endl; //if we overload operator <<

cin >> b; //If we overload operator >>
cout <<"b's value is" << b << endl; //if we overload operator <<

if (a==b)
   cout << a << "==" << b << end;
else
   cout << a << "!=" << b << endl;

a=-b;  // if we overload negation operator
a = a+b+a; //if we overload +
...
```

(a) Example code in the end shows how to override binary operators as friend functions.

- function name: operator+, operator==
- parameters: the two operands (the two objects to be added, or to be compared
- return type: addition returns the same type as the operands, comparison returns bool (true or false).

(b) Exercise:

i. What happens when the following statement is executed?

```
Money balance, depost, newbalance;
 ...

if (balance + deposit == newbalance)
    cout <<"It's correct\n";
else
    cout <<"Something is wrong!\n";
```

ii. Can you overload operator - to support subtraction?

(c) **Automatic Type Conversion**: Would you be able to add a Money object with an int value?

```
//Suppose now I received a gift of $10
cout <<" Suppose now I received a gift of $10...\n";
my_amount = my_amount+10; (1)
cout <<"After that: I own ";
my_amount.output(cout);
```

When the compiler is parsing the expression (labeled by 1), it searches for functions that overload + that takes an Money object and an int as parameter, i.e., something like:

```
Money operator+ (const Money & amount1, int dollar);
```

There is no such function defined, so the compiler tries to make do with what's avaiable:

```
Money operator+ (const Money & amount1, const Money & amount2);
```

To do that, compiler looks for way to convert **int** to **Money**. That's a constructor's job! We happen to have a constructor that takes a **int** as parameter.
Such kind of automatic conversion also happens to regular functions, and built-in types:

```
double total = 100+30.0;  //100 is converted to double first ...

int value=3+'a';  //'a' is converted to int (97, ascii code),
     // value is set to 100
int sum;
int num;

// sum is the sum of all numebrs...

double avg = sum / num;  //What's wrong here?

double avg = static_cast<double> (sum) / num;
// need to covert sum to double ... More about type casting later.
```

```
// old way:
//   double avg = (double) sum / num;
//   double avg = double (sum) / num;
```

(d) **Overload operator as member function**: For some operators, we can also do it another way. The following is an example:

```
class Money {
public:

    // add invoking object with parameter, and return the result
    Money operator+ (const Money & amount) const;
    ...
};

Money Money::operator+ (const Money & amount) const
{
    Money temp;
    temp.all_cents = all_cents + amount.all_cents;
    return temp;
}

// in main ...
Money my_amount (100,25),your_ammount (75);


Money total_amount = my_amount+your_amount;

my_amount = my_amount+10; // this is fine, first operand is
                          // used as invoking object

my_amount = 10+my_amount; // this does not work any more...
```

(e) **Overload input, output operators**:

i. Examine the usage of input and output operators more closely:

```
int day, month,year;
char c;

cin >> month >> c >> day >> c >> year;

// >> operator is associated from left-to-right, i.e., it's like:
//  ((((cin >> month) >> c) >> day) >> c) >> year;
// i.e., first cin >> month is carried out,
//       the value of this subexpression is then used as first operand
//       for the second >> operation ...
//
// This means that
//   1. the first operand of >> operator is cin, or something similar (
//       like a file stream, or even string).
```

```
//    A common type (class type) of cin, ifstream and istringstream is
//         istream
//    Will revisit this when learning about inheritance
//   2. the return type should be the type of cin ...
```

ii. Examples:

```
class Money {
public:
  friend istream& operator >>(istream& ins, Money& amount);
  //Overloads the >> operator so it can be used to input values of type Money.
  //Notation for inputting negative amounts is as in -$100.00.
  //Precondition: If ins is a file input stream, then ins has already been
  //connected to a file.

  friend ostream& operator <<(ostream& outs, const Money& amount);
  //Overloads the << operator so it can be used to output values of type Money.
  //Precedes each output value of type Money with a dollar sign.
  //Precondition: If outs is a file output stream,
  //then outs has already been connected to a file.
  ...
 private:
   long all_cents;
};
```

```
ostream& operator <<(ostream& outs, const Money& amount)
{
    long positive_cents, dollars, cents;
    positive_cents = labs(amount.all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (amount.all_cents < 0)
        outs << "-$" << dollars << '.';
    else
        outs << "$" << dollars << '.';

    if (cents < 10)
        outs << '0';
    outs << cents;

    return outs;
}
```

3 Appendix

```
//DISPLAY 11.3 Money ClassVersion 1
//Program to demonstrate the class Money.
#include <iostream>
#include <cstdlib>
```

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

//Class for amounts of money in U.S. currency.
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns the sum of the values of amount1 and amount2.

    friend bool operator ==(const Money& amount1, const Money& amount2);
    //Precondition: amount1 and amount2 have been given values.
    //Returns true if amount1 and amount2 have the same value;
    //otherwise, returns false.

    Money(long dollars, int cents);
    //Initializes the object so its value represents an amount with the
    //dollars and cents given by the arguments. If the amount is negative,
    //then both dollars and cents must be negative.

    Money(long dollars);
    //Initializes the object so its value represents $dollars.00.

    Money( );
    //Initializes the object so its value represents $0.00.

    double get_value( );
    //Precondition: The calling object has been given a value.
    //Returns the amount of money recorded in the data of the calling object.

    void input(istream& ins);
    //Precondition: If ins is a file input stream, then ins has already been
    //connected to a file. An amount of money, including a dollar sign, has been
    //entered in the input stream ins. Notation for negative amounts is -$100.00.
    //Postcondition: The value of the calling object has been set to
    //the amount of money read from the input stream ins.
    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then outs has already been
    //connected to a file.
    //Postcondition: A dollar sign and the amount of money recorded
    //in the calling object have been sent to the output stream outs.
private:
    long all_cents;
};



int digit_to_int(char c);
//Function declaration for function used in the definition of Money::input:
//Precondition: c is one of the digits '0' through '9'.
```

```
//Returns the integer for the digit; for example, digit_to_int('3') returns 3.

int main( )
{
    Money your_amount, my_amount(10, 9), our_amount;
    cout << "Enter an amount of money: ";
    your_amount.input(cin);
    cout << "Your amount is ";
    your_amount.output(cout);
    cout << endl;
    cout << "My amount is ";
    my_amount.output(cout);
    cout << endl;

    if (your_amount == my_amount)
        cout << "We have the same amounts.\n";
    else
        cout << "One of us is richer.\n";

    our_amount = your_amount + my_amount;
    your_amount.output(cout);
    cout << " + ";
    my_amount.output(cout);
    cout << " equals ";
    our_amount.output(cout);
    cout << endl;
    return 0;
}


Money::Money(long dollars, int cents)
{
    if(dollars*cents < 0) //If one is negative and one is positive
    {
        cout << "Illegal values for dollars and cents.\n";
        exit(1);
    }
    all_cents = dollars*100 + cents;
}

Money::Money(long dollars) : all_cents(dollars*100)
{
    //Body intentionally blank.
}

Money::Money( ) : all_cents(0)
{
    //Body intentionally blank.
}

double Money::get_value( )
```

```
{
    return (all_cents * 0.01);
}
//Uses iostream, cctype, cstdlib:
void Money::input(istream& ins)
{
    char one_char, decimal_point,
         digit1, digit2; //digits for the amount of cents
    long dollars;
    int cents;
    bool negative;//set to true if input is negative.

    ins >> one_char;
    if (one_char == '-')
    {
        negative = true;
        ins >> one_char; //read '$'
    }
    else
        negative = false;
    //if input is legal, then one_char == '$'

    ins >> dollars >> decimal_point >> digit1 >> digit2;

    if ( one_char != '$' || decimal_point != '.'
         || !isdigit(digit1) || !isdigit(digit2) )
    {
        cout << "Error illegal form for money input\n";
        exit(1);
    }
    cents = digit_to_int(digit1)*10 + digit_to_int(digit2);

    all_cents = dollars*100 + cents;
    if (negative)
        all_cents = -all_cents;
}

//Uses cstdlib and iostream:
void Money::output(ostream& outs)
{
    long positive_cents, dollars, cents;
    positive_cents = labs(all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (all_cents < 0)
        outs << "-$" << dollars << '.';
    else
        outs << "$" << dollars << '.';

    if (cents < 10)
```

```
        outs << '0';
    outs << cents;
}

int digit_to_int(char c)
{
    return (static_cast<int>(c) - static_cast<int>('0') );
}

//Any extra function declarations from Display 11.3 go here.

Money operator +(const Money& amount1, const Money& amount2)
{
    Money temp;
    temp.all_cents = amount1.all_cents + amount2.all_cents;
    return temp;
}

bool operator ==(const Money& amount1, const Money& amount2)
{
    return (amount1.all_cents == amount2.all_cents);
}
```