CISC 2000 — Computer Science II
Fall, 2014

Note 10/30

1 Review of lab exercises

(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).
Interface file (header file): define the class (the name, public member functions,...) so that one knows *what the class is about, what it can do (member functions provided for the class...)*. It's like a contract between the programmer who designs and implements the class, and the programmer who uses the class type.
Implementation file: how the member functions are implemented.

(b) Information hiding: user of a class (such as main.cpp) does not need to know the impelmentation details of the class.
Benefits: readability, ease of maintenance, you can change implementation file without affect main.cpp, as long as the interface stays the same.

2 Preview of lab8

Abstract Data Type when we design and implement a class, we are buidling a new data type that is hopefully useful for various applications. Therefore we will work on build such a class next: StaticIntArray.

The questions of what member functions to provide for a class shouldn't be tightly coupled with the main function. Usually, we want to provide basic operations for the class:

(a) constructor functions for initializing objects of this type
(b) input, output member functions: for read and write object's value
(c) getter member functions: to retrieve private data member value
(d) setter member functions: to set private data member
(e) comparison, append, addition, substraction ....: depends on what the class represents

3 friend function of a class

When we design and implement the **IsEqualTo** function for **rational** class, we have been design it so that it's comparing the invoking object and the object passed in parameter:

```
bool rational::IsEqualTo (const rational & n)
{
 ...
}

// this is how we call it:

rational a, b;

a.IsEqualTo (b);
```

It is a little unnatural, to pass the two objects differently, although conceptually the two objects being compared should be symmetric.

So let's try to pass both objects by parameters:

```
// comparing n1 with n2
bool rational::IsEqualTo (const rational & n1, const rational & n2)
{
 ...
}

// this is how we call it to compare a, b

rational a, b, c;

c.IsEqualTo (a,b);
//Since it's a member function, we needs to provide invoking object, even though the function
//has nothing to do with the invoking object...
```

This is even more awkard!

So we prefer to make this function a non-member function:

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

This won't compile, as the funciton cannot access private members of the class. We could call member function **get_numerator, get_denominator** to access them. But there is overhead in making function calls...

The solution: make this (non-member) function a friend of the class, by adding the following in the public section of class rational. A friend function of a class can access private members of the class.

```
class rational{
public:
     friend bool IsEqualTo (const rational & n1, const rational &n2);
      ...
```

4 static member variables

Usually, each object of a class type has its own copy of all member variables. For example,

```
Data today, yesterday;
```

In memory, variable **today** has three parts: year, month, and day; variable **yesterday** also has three parts. They have different and separate memories for storing their member variables. (Please be reminded how we draw memory for a variable that is a struct or class type.).

But in some cases, we want all objects of the class type to share a single copy of some member variable. We do this by adding the keyword **static** to the member variables' declaration (in the class definition).

The following are two such cases:

(a) the member variable is a constant, therefore its value will be the same anyway. For example, in the StaticIntArray class, CAPACITY is a constant across all different objects of the type. (Refer to lab8)

(b) We want to keep track information about the whole class: for example, how many objects of this type has been created so far...

```
class rational{
public:
    constructor (int p, int q)
    {
       numerator = p;
       denominator = q;
       counter++;
       cout <<"So far: " << counter << " rational objects created!\n";
    }
...

private:
   static int counter; // one copy of counter for the class,
          // not for individual rational object
 ...
};
```

More on this in lab exercise next week.

5 Usage of keyword const.

(a) To declare a named constant

(b) To denote a read-only parameter (pass-by-reference, or array). If the function includes code that modifies the parameter, the compiler will report compilation errors.

(c) To denote a member function to be const, i.e., the function does not modify the invoking object:

```
class rational {
 ...
 int get_denominator () const;
 bool IsEqual (const rational & n) const;
 ...
 }
```

6 I/O and file stream

(a) The concept of **stream**: Input and ouput to a program are both like a stream of characters: that will be read and write in order.

When you use:

```
int a;
char c;
int b;

cin >> a >> c >> b;
```

The keyboard input (whatever the user types in) are parsed in the order of input, in order to extract an integer value to be assigned to **a**, a character value to be assigned to **c**, ...

Likewise, when you do:

```
cout << month <<"/" << day <<"/" << year;
```

The value of month, slash character, day, ... are sent to the output stream (terminal) one by one, in order.

(b) We can read from a disk file in a way similar to how we have been reading from keyboard. We can write to a disk file in way similar to how we have been writing to terminal. We will practice on this in next lab exercise.

```
#include <fstream>  //This file defines classes (that others have defined and implemented)
      // which allow you to read and write from disk files easily.

ifstream in_file; // declare an object variable named in_file of class ifstream type
// in order to read from a disk

in_file.open ("myinput.txt"); // connect the object with a file named
// myinput.txt in current direcotry

char c;
int month, day, year;

in_file >> month >> c >> day >> c >> year; // to read a date in the
// format of yyyy/month/day...
```