Homework 6

CS 173: Intermediate Computer Science — Spring 2014
Instructor: Thomas Bressoud — Due: 2014-03-12

- In this assignment you will practice writing code in C++
  - Writing functions for the sorting problems.
  - Writing interaction with objects for the file input and output.
  - Writing a C++ class for the Rational problem.
- The earlier guidelines for documenting and commenting your code hold; you should have appropriate pre- and post- conditions for each function/method that you write. You should use C++ assert() to check for violations of pre-conditions.

Part 1: Sorting

Write a C++ program (in a file called `sort.cpp`) that takes the name of a file of integers on the command line and then sorts the numbers in the file using both selection sort and insertion sort, writing the results to a file. The file will have the number of integers on the first line. Use this value to declare an array of that size and then read the integers into the array. Have your program measure the time of both sorts. When we talk about more efficient sorting algorithms, we will incorporate these new algorithms into this program so that we can compare their efficiency to these standard "dumb" sorts.
Your program should work in the following manner (be careful to follow these instructions carefully):

- Take the input file name from the command line.
- Output the result of selection sort to a file whose name is identical to the input file name with a ".selection" appended to the end. You should include the number of integers as the first line. Add the number of seconds taken by the sort as the last line.
- Output the result of insertion sort to a file whose name is identical to the input file name with a ".insertion" appended to the end. You should include the number of integers as the first line. Add the number of seconds taken by the sort as the last line.

For example, if the input file given is called `input.txt` and has the following contents:

```
4
100
23
73
50
```

then your program should create two files, `input.txt.selection` with contents

```
4
23
50
73
100
1.25
```

and `input.txt.insertion` with contents

```
4
23
50
73
100
1.842
```

(of course, the number of seconds on the last line will be different).

To take your time measurements, we use a Unix-provided function called `gettimeofday()`. At the command line, use `man gettimeofday` to see the documentation, including the required `#include` to use to obtain the proper declarations needed to use this. The function invocation involves passing the address of a structure, which we have not covered yet, so I provide a function definition below that uses `gettimeofday` and returns a real-valued number of seconds that you can use before and after each part of code you wish to time:

```
double clocktime()
{
    struct timeval t;
    double retval;

    gettimeofday(&t, NULL);
    retval = (double)t.tv_sec;
    retval += t.tv_usec / 1000000.0;
    return retval;
}
```

Part 2: Rational

Your task in this part is to build the Rational class in C++ and to test it in a manner similar to running the test suite from our Python version. So that we are all programming to the same interface, I give function declarations for the public part of the interface below. Note that this is a blend between what we did in Python and the development in the book, but not identical to either.

- `Rational(int n = 0, int d = 1);`: constructs a `Rational` object, with (optional) default numerator and denominator as given. If n and d have a nontrivial divisor, n and d should be revised to reduced form. Only the numerator should ever be negative in the resultant rational number, but any combination of negative n and d is permissible on construction. An `assert()` should be used to check for zero denominator.
- `int getNum()`: accessor returning the numerator data member.
- `int getDen()`: accessor returning the denominator data member.
- `void set(int n, int d)`: updates the numerator and denominator data members of the object, following the same stipulations as the constructor (reduced form and only numerator possibly negative in the result).
- `Rational operator+(Rational &other)`: This is equivalent to the Python `__add__`, adding the current Rational object to `other`, a second Rational object, passed by reference, and yielding a brand new Rational object.
- `Rational operator*(Rational &other)`: This is equivalent to the Python `__mul__`, multiplying the current Rational object to `other`, a second Rational object, passed by reference, and yielding a brand new Rational object (in reduced form).
- `bool operator<(Rational &other)`: This is equivalent to the Python `__lt__`, comparing the current Rational object to `other`, a second Rational object, passed by reference, and yielding a `true` if the current object is less than `other` and `false` otherwise.
- `bool operator==(Rational &other)`: This is equivalent to the Python `__eq__`, comparing the current Rational object to `other`, a second Rational object, passed by reference, and yielding a `true` if the current object is equal in value to `other` and `false` otherwise.
- `string str()`: Create and return a C++ string object representation of the rational number in the form of "<numerator>/<denominator>".

In a manner similar to our unit test in Python, you should create a driver main program called `testRational.cpp` that consists of a `main()` function definition whose body is a series of function calls to individual tests. These tests should mimic (but without exception handling) what we did in `test_Rational.py`, with each function focusing on setting up a test scenario and then checking the result against an expectation. You can report success or failure for each test by messages output via `cout` (or `cerr`).

The easiest method to take two C++ source files and both compile them and then combine them into a single executable file is to let `g++` do all the work:

```
$ g++ -g Rational.cpp testRational.cpp -o testRational
```

In class, we will look more closely at individual compilation and the use of a `Makefile` to put together a set of instructions that put together the steps to create an executable file.

Your final submission should consist of the files: `sort.cpp, Rational.h, Rational.cpp, testRational.cpp`. Submit your homework via submitbox.