ADVANCED MATHEMATICAL PROGRAMMING

MTH399 / U15161 – Level 3

UNIVERSITY OF PORTSMOUTH

Spring Semester – Second Coursework

Academic Session: 2010–2011

INSTRUCTIONS

a) **Deadline: Wednesday, May 16, 3PM** **on Victory**

b) Compress all source code and result files into a zip file and upload it to the Victory assignment; **no printed copies are needed**.

c) **Make sure to label all materials with your reference number**.

d) This is the **second coursework (out of two), worth** 60% of the unit.

e) Complete **all** of the assignment except for whatever carries the label "Bonus".

f) Explain your routines using comments in the code.

g) If you cannot complete a project, upload what you attempted to do.

h) Assignment must be undertaken **alone**.

Part 1. (100 marks) You are expected to define a class `real` representing real numbers with arbitrary precision. Your class must use parameters, functions and operators, whether private, protected or public, for an element $x = \pm \ldots a_2 a_1 a_0 \,.\, d_1 d_2 d_3 \cdots \in \mathbb{R}$, including:

1. an integer describing the sign of x: $+1$ or -1.
2. two integers (preferably `long int` types), n and m, respectively denoting the amount of integer-part and decimal-part digits stored in the decimal representation of x:

$$x = \pm \boxed{a_{n-1} \quad a_{n-2} \quad \cdots \quad a_1 \quad a_0 \;\big|\; d_1 \quad d_2 \quad \cdots \quad d_m}$$

3. two vectors having integer entries: one for the integer part $(\ldots a_2 a_1 a_0)$ and one for the decimal part $(d_1 d_2 d_3 \ldots)$ of x;
4. functions returning:
   a) the `double` version of a `real` variable x, as well as
   b) the `long int` version whenever $x \in \mathbb{Z}$ (or whenever we are only interested in $[x]$);
5. constructors translating familiar types to `real`:
   a) from `double`, e.g. `real::real ( double x, long int digits );` and
   b) from integer – `long` type again, e.g. `real::real ( long int k );`
6. `real` variable operators: equality $=$, addition $+$, subtraction $-$ and (**Bonus**:) multiplication $*$;
7. mixed operators computing:
   a) the division of a `real` type by `long int` type, and
   b) the product of a `real` type by an integer between 0 and 9.
8. specific constructors (besides the ones in **5.**), copy constructors and destructors. Make sure one of the constructors allows you to declare a new `real` in terms of two separate vectors of integers, and another constructor has only a vector and an index and "places the dot" in the vector:

   ```
   real::real ( vector<int> v, long int place )
   ```

9. inspector functions for each of the parameters, as well as for special quantities attached to your class, e.g. a `long int Tol_digit` returning the position of the first non-zero digit d in $x = 0 \,.\, 0 \cdots 0\, d \ldots$ (this will be useful in Part 2), and a function

   ```
   vector<int> real::Vector ( void ) const
   ```

   merging your `integer` and `decimal` parts into a single vector (this can be of some use in **6.**).
10. functions altering your `real` type slightly, e.g.

    ```
    real real::negative ( void ) const
    ```

    (hopefully this needs no explanation) or

    ```
    void real::shift ( long int digits, int direction )
    ```

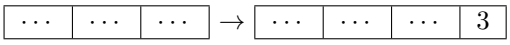
    if we want to change the decimal point's position.

Comments:

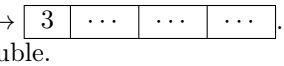
a) It may be convenient for you to define functions relieving `real` (and/or `vector`) types of unnecessary zeros to the left.

b) For reasons seen in the next comment below, the use of STL container `vector` may be more advisable than that of arrays. You can always resize vectors from the left or from the right easily, say

```
v.push_back( 3 );
```

if you want to add 3 to the end of your vector,  and a function such as

```
void push_front ( vector<int> & v, int k )
{
   vector<int>::iterator it;
   it = v.begin( );
   v.insert ( it, 1, k );
   return;
}
```

called as `push_front( v, 3 );` if you want to add it to the first place:  . Doing the same thing with arrays would always be a source of further trouble.

c) In the calculations involved in two of the four arithmetic operations between `real` types, you may find yourselves in the need to define "operations" between STL `vector` types to make things simpler.

d) The division by integers x/K, $x \in \mathbb{R}$, $K \in \mathbb{Z}$ described in **7.** can be simplified by considering x and $1/K$ separately, both seen as `real` types, and then multiplying them. Hence, you need a function of the sorts of

```
real inverse ( long int K, long int digits );
```

in order to obtain the inverse $0.d_1 d_2 \ldots d_{\texttt{digits}}$ of an integer `K` with a given amount of decimal digits. Needless to say, the integer part of $1/K$ will consist of a single element: $a_0 = 0$. Now:

(i) firstly, let 10^j be the smallest power of ten such that $10^j > K$;

(ii) the first $j-1$ entries of the decimal part of $1/K$ will be zeroes: $K^{-1} = 0\,.\,0 \cdots 0\, d_j d_{j+1} \cdots$;

(iii) in order to find entries $d_j, d_{j+1}, \ldots$, the following procedure works: perform integer division of 10^j by K: $10^j = d_j K + r_j$; save d_j and the general d_k is equal to the quotient in the integer division $10 r_{k-1} = d_k K + r_k$, $k = j+1, \ldots,$ `digits`.

Be sure to test these routines in sample numbers $K = 2, 3, 14, 2719, \ldots$ before using them elsewhere.

e) Use separate `C++` and header files consistently.

f) Make sure the amount of constructors and destructors is reasonable, i.e. that you use all of them in the project.

g) Think of ways in which to effectively test the correctness and efficiency of your functions and operators. Make sure you keep track of each property that is being checked, be it by means of exception throwing or by writing down on an output file passed by reference as an input of your test functions – *do not* declare the `ofstream` file variable as global.

Part 2. (**Bonus**) Use the project in Part 1 to approximate the following numbers, each with more correct digits (at least 30) than available with the traditional `double` or `long double` types: π, e and $\sqrt{2}$, $\log 2$ (expressed as $-\log \frac{1}{2}$ if necessary).

There is a diversity of procedures you can use to approximate these numbers without having to resort to division of two `real` types. One possible way is by using truncated Taylor series, e.g.

$$e^x \simeq \sum_{k=0}^{N} \frac{x^k}{k!}, \qquad \arctan x \simeq \sum_{k=0}^{N} \ldots, \qquad \arcsin x \simeq \sum_{k=0}^{N} \ldots, \qquad \text{a long etcetera}$$

which, at most, will require the use of the `inverse` function mentioned in Part 1. For instance, the following property yields good approximations for π (that is, approximations which do not rely on excessively large N):

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239}.$$

Another possible way is using iteration methods such as those due to Newton-Raphson: given an initial condition x_0,

$$\boxed{x_{k+1} = x_k - \tfrac{f(x_k)}{f'(x_k)}, \quad k \geq 0} \tag{1}$$

aimed at approximating solutions of an equation $f(x) = 0$.

Comments:

a) It will be your task, after some minor example checking, to discern an approximate amount of working digits you need in order to ensure convergence.

b) Feel free to merge Parts 1 and 2 in a single project.

c) If you use truncated series, you need to define the Taylor series yourself; for instance, a truncated Taylor series for $f(x) = \ln x$ in powers of $x - a$ (whatever a is) would have a prototype

```
real real_ln ( real x, long digits, long final_digits, long max_iterations );
```

think why.