Reminding nearly all C++ in a single exercise : Templates, Operator Overloading, STL, Boost ...

S. Varrette, V.Plugaru and P. Bouvry
<Firstname.Lastname@uni.lu>

Version 1.1

Resources:

- C++ Reference
- SGI Programmer's Guide of STL
- Boost C++ libraries
- Doxygen: Source code documentation generator tool
- Eigen

Exercise 1 *A simple generic matrix interface*

The objective of this exercice is to propose an basic yet flexible interface to deal with matrix operations. Every aspects of C++ will be highlighted.

Part I — Preliminary

Prepare your working environment with Doxygen support (for documentation generation). Kindly pay attention to your code writing convention, in particular to integrate Doxygen comments at the same time you program.
You are encouraged to use CMake. You can adapt and inspire from the following `CMakeLists.txt`:

```
# -*- mode: cmake -*-
#
# [CMake](http://www.cmake.org/) Configuration
#
cmake_minimum_required (VERSION 2.8.12)
project (TutorialMatrixSTL CXX)

# Rely on C++11
SET(CMAKE_CXX_STANDARD 11)
SET(CMAKE_CXX_FLAGS -std=c++11 )

#====== [Boost](http://www.boost.org/) ======
find_package(Boost COMPONENTS random program_options timer system REQUIRED)
include_directories(SYSTEM ${Boost_INCLUDE_DIRS})
list(APPEND EXTRALIBS ${Boost_LIBRARIES})

#======== [Eigen](http://eigen.tuxfamily.org/) ====================
find_package(Eigen REQUIRED)
include_directories(${EIGEN_INCLUDE_DIRS})

# Sources of the main executable
set(matrix_HEADERS Tools.h Matrix.h SquareMatrix.h)

add_executable(matrix_stl main.cpp ${matrix_HEADERS})
target_link_libraries(matrix_stl ${EXTRALIBS})
```

If you do not want to use CMake, prepare a `Makefile` able to compile your code, otherwise the lecturers **WON'T CORRECT** your assignment. Also, your main application (`matrix_stl` in the below example) should fulfill **at least** the following command-line options:

```
$> ./matrix_stl -h
Assignment Matrix / STL, by
   <Lastname> <Firstname> <StudentID>
Available options:
  -h [ --help ]    Display this help message
  -v [ --verbose ] Verbosity level
  --repeat arg (=1) Repeat each test <arg> times
  -m [ --rows ] arg (=3) Number of rows
  -n [ --cols ] arg (=4) Number of columns
  -r [ --random ]  Initialize the matrix with random values
  --min arg (=-10)  Minimal value for the random values
  --max arg (=10)   Maximal value for the random values
```

An easy way to implement such options is to rely on Boost Program Options.

On the UL HPC platform, you'll need to load a certain number of modules, and run the CMake build using:

```
$> module load toolchain/ictce devel/Boost devel/CMake math/Eigen
$> cd  build
$> cmake ../src -DBOOST_ROOT=${EBROOTBOOST} -DEIGEN_INCLUDE_DIR=${EBROOTEIGEN}/include
```

Part II — Generic m-by-n dense matrix in $\mathcal{M}_{m,n}$

This part propose to define a templated class `Matrix<T>` that handles m-by-n dense matrix belonging to $\mathcal{M}_{m,n}$ such as:

$$M = \begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,0} & \dots & a_{m-1,n-1} \end{pmatrix}$$

where $a_{i,j}$ belongs to a given type `T` that support classical operations (+, −, ⋆, << etc.)

1. Define a dedicated exception class (`MatrixRuntimeError`) to be throwned in case of runtime error when using our `Matrix` class. Your class will have to inherit from `std::runtime_error`.

The proposed interface for the `Matrix` class is the following:

```
template <class T>
class Matrix {
   /* Some friend operators (implemented as non-member functions) */
   template <class U> friend
   std::ostream& operator<<(std::ostream& os, const Matrix<U> & M);
   // /!\ inefficient prototypes for the following operators
   template <class U> friend            /* Addition */
   Matrix<U> operator+(const Matrix<U> & M1, const Matrix<U> & M2);
   template <class U> friend         /* Soustraction */
   Matrix<U> operator-(const Matrix<U> & M1, const Matrix<U> & M2);
   template <class U> friend       /* Multiplication */
   Matrix<U> operator*(const Matrix<U> & M1, const Matrix<U> & M2);
```

```
   template <class U> friend           /* Transpose */
   Matrix<U> operator~(const Matrix<U> & M);
   // -----------------------------------------
public:
   const size_t rows;     /**< number of columns */
   const size_t cols;     /**< number of rows */
protected:
   std::vector<T> _container; /**< The actual container */
   // -----------------------------------------

public:
   // Constructors / Destructors
   Matrix(const size_t m = 2, const size_t n = 2);
   Matrix(const Matrix<T> & M) :
      rows(M.rows), cols(M.cols), _container(M._container) {}
   Matrix(const std::vector<T> & v, const size_t m, const size_t n):
      rows(m), cols(n), _container(v) { assert (m*n == v.size()); }
   template <int R, int C>
   Matrix(const Eigen::Matrix <T,R,C>& M);
   virtual ~Matrix() {}

   // Assignment (eventually with elements assigned the same constant)
   Matrix<T> & operator=(const Matrix<T> & M);
   template <int R, int C>
   Matrix<T> & operator=(const Eigen::Matrix <T,R,C> & M);
   Matrix<T> & operator=(const T & val);

   // comparison
   bool operator==(const Matrix<T>& M) const;
   bool operator==(const ublas::matrix <T> & M) const;
   template <int R, int C>
   bool operator==(const Eigen::Matrix <T,R,C> & M) const;
   bool operator!() const { return iszero(); }

   // Accessor
   // get M(i,j)
   const T & operator() (const size_t i, const size_t j) const;
   T &     operator() (const size_t i, const size_t j);
   const T & operator[] (const size_t i) const { return _container[i]; }
   T &     operator[] (const size_t i) { return _container[i]; }
   const T * data() const { return _container.data(); }
   T * data()       { return _container.data(); }
   std::vector<T> row(const size_t i) const;
   //typename std::vector<T> col(const size_t j) const;

   // Iterators
   typename std::vector<T>::iterator begin() { return _container.begin(); }
   typename std::vector<T>::const_iterator begin() const{ return _container.begin();}
   typename std::vector<T>::iterator begin(const size_t i);
   typename std::vector<T>::const_iterator begin(const size_t i) const;
   typename std::vector<T>::iterator end() { return _container.end(); }
   typename std::vector<T>::const_iterator end() const { return _container.end(); }
   typename std::vector<T>::iterator end(const size_t i);
   typename std::vector<T>::const_iterator end(const size_t i) const;

   /**
    * @return number of T elements in this matrix
    */
   size_t size() const { return rows * cols; }
   /**
```

```
    * Check if the matrix is null (i.e. composed by only 0 elements)
    */
   bool iszero() const;
   // ==== Generator / fill functions ===
   /**
    * Assignement of the elements to incremented values (using std::iota)
    */
   void generate();
   /**
    * Assignement of the elements to a value val (using std::fill)
    */
   void generate(const T & val);
   /**
    * Assignement of the elements to random values (using std::generate)
    * @param gen Functor to the random generator to be used
    */
   template<class RandomGenerator> void generate(RandomGenerator gen);
   /**
    * Transpose the matrix
    */
   Matrix<T> transpose() const;
   // ==== Printing functions ====
   /**
    * Print the j-th column to the output stream s_out
    */
   std::ostream & printColumn(const size_t j = 0,
                        std::ostream &s_out = std::cout) const;
   /**
    * Print the i-th row to the output stream s_out
    */
   std::ostream & printRow(const size_t i=0, std::ostream &s_out = std::cout) const;
   /**
    * Print the content of the Matrix to the output stream s_out
    */
   virtual std::ostream & print(std::ostream & s_out = std::cout) const;
};    // ============ end class Matrix ===================
```

2. *(Matrix constructor)* Implement the default constructor such that an exception is throwned if either `m` or `n` is equal to 0 *i.e.* if the matrix is degenerated.
3. Why having made the destructor of the `Matrix` class virtual ?
4. *(Matrix printing)* Implement the methods `printRow`, `printColumn` and `print`.
   a) Explain why those methods should be `const`
   b) Use these methods to implement the function

```
template <class U> friend
std::ostream& operator<<(std::ostream& os, const Matrix<U> & M);
```

5. *(Matrix accessors M(i,j))* Implement the accessors operators

```
template<class T>
inline const T & Matrix<T>::operator()(const size_t i, const size_t j) const;
template<class T>
inline T & Matrix<T>::operator() (const size_t i, const size_t j);
```

   a) What is the purpose of the `inline` keyword?
   b) Explain why there should exist two versions for these accessors.
6. *(Matrix iterators)* Implement the method permitting to retrieve iterators at the beginning (resp. the end) of the i-th row of the matrix *i.e.*

```
typename std::vector<T>::iterator begin(const size_t i);
typename std::vector<T>::const_iterator begin(const size_t i) const;
```

```
typename std::vector<T>::iterator end(const size_t i);
typename std::vector<T>::const_iterator end(const size_t i) const;
```

a) Explain why the `typename` keyword is mandatory.
b) Explain why two versions of each methods should be defined.

7. * Implement the three `generate` methods, using respectively the following STL algorithms
   a) `std::iota`
   b) `std::fill`
   c) `std::generate`
8. *** This question is dedicated to the definition of two classes `Generator` and `RealGenerator` (belonging to the namespace `Random`) used to generate random objects using the Boost C++ libraries.
   a) Ensure that the Boost C++ libraries are installed.
   b) Try and test the use of the Boost Random Number Library. Adapt your `Makefile` (or `CMakeLists.txt`) accordingly.
   c) We assume here a type `T` for which there exists a constructor that takes an integer argument (of type `IntType`). Note that this holds for any integer type (such as `int` or `unsigned long`) but also for any class that match this kind of implementation:

```
template<class IntType = int>
class T {
 public:
 T(const IntType & val); // constructor with an integer argument
 ...
};
```

Implement the generator class `Random::Generator` that make use of the Boost Random library to generate random elements of type `T` (by calling the constructor of `T` on uniformly chosen integers). You will typically use the following interface:

```
#include <boost/random.hpp>
namespace Random {
 template<class T,
        class IntType = int,
        class Engine = boost::mt19937>
 class Generator {
    typedef /* TO BE COMPLETED */ distribution_t;
    distribution_t _dist; /**< type of distribution */
 public:
    Generator(const IntType & min = IntType(0),
    const IntType & max = IntType(100));
    virtual ~Generator() {}
    IntType min() const { return _dist.min(); }
    IntType max() const { return _dist.max(); }
    T operator()();
 };
}; // namespace Random
```

   d) We assume now a type `T` for which there exists a constructor that takes a real number argument (of type `RealType`). This holds for any real number type (such as `float` or `double`) but also for any class that match this kind of implementation:

```
template<class RealType = double>
class T {
```

```
  public:
  T(const RealType & val); // constructor with an real number argument
  ...
};
```

Use the same approach expounded in the previous question to implement a class `Random::RealGenerator` as follows:

```
#include <boost/random.hpp>
namespace Random {
  template<class T,
         class RealType = double,
         class Engine = boost::mt19937>
  class RealGenerator {
    /* ... */
  };
```

e) Test both classes, first on traditional vectors (`std::vector`), then over the method

```
  template<class T>
  template<class RandomGenerator>
  void Matrix<T>::generate(RandomGenerator gen);
```

f) Adapt your main application (and the command-line options) to generate and print a random matrix using the previous methods.

9. *(Matrix assignment by the '=' operator)*

a) Implement the method

```
  template<class T>
  Matrix<T> & Matrix<T>::operator=(const Matrix<T> & M);
```

In particular, ensure the dimention are correct (or throw an exception) and deal with auto-assignment instructions (*i.e* `M = M`).

b) Implement the variant

```
  template<class T>
  inline Matrix<T> & Matrix<T>::operator=(const T & val);
```

where all elements are assigned the value `val`.
Whereas you could use the `Matrix<T>::generate(val)` method, propose an alternative approach based on `std::vector<T>::assign()`

10. *(Matrix comparison)*

a) Implement the operator == :

```
  template<class T>
  bool operator==(const Matrix<T>& M) const;
```

b) ** We now study three versions of the method `iszero()`, all based on the STL `find_if` algorithm and:

i – using a predicate `isNonZero` (to be defined).
ii – using a predicate `isZero` (to be defined) and the STL negator `std::not1`
iii – using the STL operator class `std::not_equal_to` and the STL parameter binder `std::bind2nd`

c) Why is it important to make all your defined predicates derive from the STL function objects `std::unary_function<Arg,Res>` or `std::binary_function<Arg1,Arg2,Res>` ?

d) * Assuming it is not the case, how would you use a classical function as a predicate? Validate this approach on the `iszero` method implementation.
Hint: take a closer look at the STL conversor `std::ptr_fun`.

11. *(Matrix arithmetic operators)* *
    a) Why arithmetic operators are proposed as non-member function instead of member methods?
    b) Propose a basic implementation for the operators `*` and `~` (*i.e.* nested `for` loops)
    c) * Implement the `+` and `-` operators using the STL `std::transform` algorithm. You'll also consider the STL operator classes `std::plus` and `std::minus`.
    d) ** Explain why the proposed implementation is considered inefficient. How would you improve it? Propose a new implementation.
    Hint: take a closer look at the returned type.

12. *(Matrix arithmetic operators benchmarking)* *** The objective is now to benchmark your implementation for various matrix size and to compare your results with the general implementations proposed in the literature, in particular Eigen and Boost uBLAS library.

    a) Prepare a timer for your application using `boost::timer::cpu_timer` (see Boost CPU timer), which permit to measure both the wallclock and the cpu time.
    b) To be statistically significant, the benchmarked results will need to be repeated over multiple random matrices (see the `-repeat` command-line option).
    Adapt your code to integrate a statistical accumulator `accumulator_set<>` – see Boost Accumulator:

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics/stats.hpp>
#include <boost/accumulators/statistics/mean.hpp>
using namespace boost::accumulators;
typedef accumulator_set<long double, features<tag::mean> > accumulators_t;
```

    c) Benchmark your implementation for the following operations: *addition*, *subtraction*, *transposition* and *multiplication*.
    d) ** Perform the comparison of your implementation with Boost uBLAS library. You'll need to prepare the initialization of uBLAS matrices from a previously generated matrix (of your own implementation), thus to implement the following methods:

```
   namespace ublas = boost::numeric::ublas;
   [...]
   Matrix<T> & Matrix<T>::operator=(const ublas::matrix <T> & M);
   [...]
   bool Matrix<T>::operator==(const ublas::matrix <T> & M) const;
```

    Also, the following wrapper might help you:

```
template <typename T>
class BoostMatrix: public ublas::matrix<T> {
public:
   BoostMatrix(const size_t m=2, const size_t n=2): ublas::matrix<T>(m,n) {}
   BoostMatrix(const Matrix<T> & M);
   virtual ~BoostMatrix() {}
   T det() const;
};
```

    e) *** Perform the comparison of your implementation with Eigen. You'll need to prepare the initialization of Eigen matrices from a previously generated matrix (of your own implementation), thus to implement the following methods:

```
   template <int R, int C>
   Matrix<T>::Matrix(const Eigen::Matrix <T,R,C>& M);
   [...]
```

```
   template <int R, int C>
   bool Matrix<T>::operator==(const Eigen::Matrix <T,R,C> & M) const;
```

Also, the following wrapper might help you:

```
template <typename T>
class EigenMatrix : public Eigen::Matrix<T,
                                Eigen::Dynamic,
                                Eigen::Dynamic,
                                Eigen::RowMajor> {
public:
   typedef Eigen::Matrix<T,
                    Eigen::Dynamic,
                    Eigen::Dynamic,
                    Eigen::RowMajor> eigen_matrix_t;
   EigenMatrix(const size_t m=2, const size_t n=2) : eigen_matrix_t(m,n) {}
   EigenMatrix(const Matrix<T> & M) : eigen_matrix_t(M.rows, M.cols) {
      std::copy(M.begin(), M.end(), this->data());
   }
   EigenMatrix(const EigenMatrix<T> & M) : eigen_matrix_t(M) {}
   virtual ~EigenMatrix() {}
};
```

f) Collect the benchmarking data and use either Gnuplot[1] or R to display the collected data. Example of such plots are proposed in the figure 1.

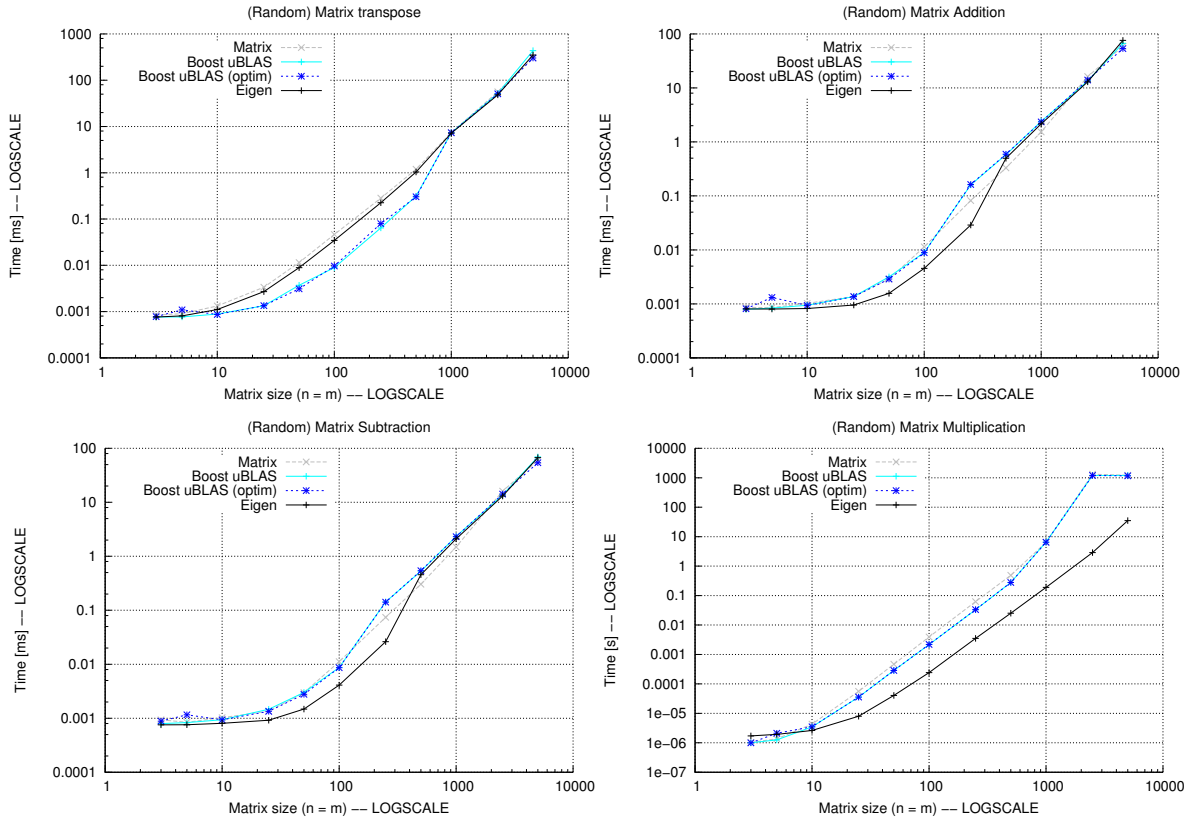

Figure 1: Performance of the `Matrix` implementation of the main operations

[1]See the excellent Gnuplot not so FAQ.

Part III — Square dense matrix in $\mathcal{M}_n$

1. Provide the interface of a class `SquareMatrix` that derives from the previous class `Matrix<T>` and implements... A n-by-n square matrix.
2. Implement the method that computes the co-matrix (also called the minor matrix) $M_{i,j}$:

```
template<class T>
SquareMatrix<T> SquareMatrix<T>::coMatrix(const size_t i, const size_t j) const;
```

   As a reminder, $M_{i,j} \in \mathcal{M}_{n-1}$ consists of the matrix M where the i-th row and the j-th column are deleted, *i.e.*

$$M_{i,j} = \begin{pmatrix} a_{0,0} & \cdots & a_{0,j-1} & a_{0,j+1} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots & \vdots & \cdots & \vdots \\ a_{i-1,0} & \cdots & a_{i-1,j-1} & a_{i-1,j+1} & \cdots & a_{i-1,n-1} \\ a_{i+1,0} & \cdots & a_{i+1,j-1} & a_{i+1,j+1} & \cdots & a_{i+1,n-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,j-1} & a_{m-1,j+1} & \cdots & a_{m-1,n-1} \end{pmatrix}$$

3. Implement the method

```
template<class T>
T SquareMatrix<T>::det() const;
```

   that computes the determinant of the matrix. You shall use the naive approach based on Laplace's formula and the adjugate matrix:

$$det(M) = \sum_{j=0}^{n-1} (-1)^j a_{0,j} M_{0,j}$$

   Note: you can validate your implementation on the following test-cases:

$$\begin{vmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{vmatrix} = 0 \qquad \begin{vmatrix} -2 & 2 & -3 \\ -1 & 1 & 3 \\ 2 & 0 & -1 \end{vmatrix} = 18 \qquad \begin{vmatrix} 3 & 2 & 0 & 1 \\ 4 & 0 & 1 & 2 \\ 3 & 0 & 2 & 1 \\ 9 & 2 & 3 & 1 \end{vmatrix} = 24 \qquad \begin{vmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 0 & 4 & 5 & 6 \\ 2 & 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 6 & 5 \\ 0 & 0 & 0 & 5 & 6 \end{vmatrix} = 99$$

4. *(Benchmark)* *** Complete your benchmarking campaign with an evaluation of the `det()` method, and compare it with Eigen.
   Note: the `determinant()` method implemented in Eigen relies on an LU decomposition for which a true division is required. In particular, using as scalar type `T` an integer (`int`) will lead to wrong results.