Simplifying and unifying the design of user-defined mathematical containers

Document number: Draft
Version: 0.1
Date: 2012

Vincent Reverdy (vince.rev@gmail.com)
Laboratory Universe and Theories, Observatory of Paris,
5 place Jules Janssen, 92195 Meudon, France

December 29, 2012

Abstract

This proposal concerns the addition of three helper classes to the standard library in order to simplify and unify the creation of new mathematical containers. The goal is to reduce the current complexity of implementing new containers with optimized operations using generic tools based on the Curiously Recurring Template Pattern (CRTP) technique. It will also provide a common base for all the constant size vectors and matrices of graphics and scientific librairies (`Vector2D, Vector3D, Matrix3D, MatrixNxM, Tensor3D`...). This proposal should be able to fill the current lack of simple mathematical containers of the C++ and bring a standardized answer to a lot of technical and scientific basic problems.

Contents

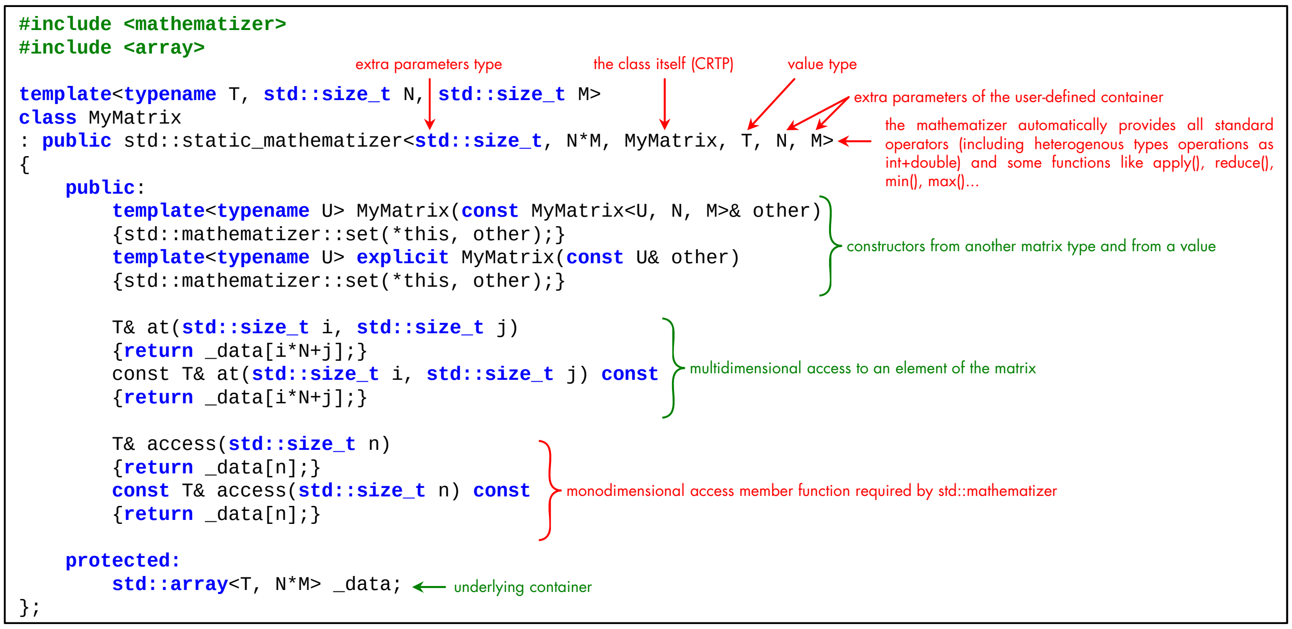
I Motivation

This proposal comes from a simple observation: an impressive number of colleagues working in a wide variety of engineering and scientific fields are concerned by the lack of standardized tool to design basic vectors and matrices and many of them emphasize the advantage of languages like FORTRAN on this specific point. This observation can be easily confirmed by looking to widely used graphics and scientific libraries and frameworks: oftenly, classes such as `Vector2D`, `Vector3D`, `Tensor3D` or `Matrix4x4` are reimplemented from scratch. Some examples are given in table 1. This results in a huge development effort to design and optimize standard operations on simple things like a 3D vector. So the open question addressed here is : what is the most generic base of all mathematical containers that could be standardized in order to provide an elegant solution to this problem ?

Before going further, it is important to note two key elements. First, even if `std::valarray` provides an optimized container that supports all basic operations, it is not well designed for composition or inheritance. Consequently, it cannot be used as a common base on which one can add new operators or functions. Second, the problem addressed here is not the same as the one addressed by linear algebra libraries. The goal of the proposed tools is not to diagonalize 100×100 matrices, to optimize sparse or non-sparse containers, or to solve sets of hundreds of equations.

The proposal concerns the creation of new containers by the users and is not about the integration of new containers in the standard library. The goal is to provide a generic mechanism that simplifies and unifies the design of mathematical arrays that need the standard operators and some function application members. This mechanism based on the Curiously Recurring Template Pattern (CRTP) technique will reduce the long step of design, implementation and optimization of basic things like a `Vector3D` to the simple inheritance from a class. A resulting example of use for the creation of a constant size matrix is provided in figure 1. Finally, these tools will have a wide range of use and will allow library designers, software developers, engineers and scientists to create their own mathematical containers without having to implement them from scratch.

Figure 1: Creation of a basic constant size matrix using a mathematizer.



```
#include <mathematizer>
#include <array>

template<typename T, std::size_t N, std::size_t M>
class MyMatrix
: public std::static_mathematizer<std::size_t, N*M, MyMatrix, T, N, M>
{
    public:
        template<typename U> MyMatrix(const MyMatrix<U, N, M>& other)
        {std::mathematizer::set(*this, other);}
        template<typename U> explicit MyMatrix(const U& other)
        {std::mathematizer::set(*this, other);}

        T& at(std::size_t i, std::size_t j)
        {return _data[i*N+j];}
        const T& at(std::size_t i, std::size_t j) const
        {return _data[i*N+j];}

        T& access(std::size_t n)
        {return _data[n];}
        const T& access(std::size_t n) const
        {return _data[n];}

    protected:
        std::array<T, N*M> _data;
};
```

extra parameters type
the class itself (CRTP)
value type
extra parameters of the user-defined container
the mathematizer automatically provides all standard operators (including heterogenous types operations as int+double) and some functions like apply(), reduce(), min(), max()...
constructors from another matrix type and from a value
multidimensional access to an element of the matrix
monodimensional access member function required by std::mathematizer
underlying container

Table 1: Examples of basic vectorized containers of some C++ libraries and frameworks.

Library	**Category**	**Example of containers**
Qt	Application/GUI Framework	`QVector2D, QVector3D, QVector4D, QGenericMatrix, QMatrix4x4`...
VTK	Visualization	`vtkVector2, vtkVector3, vtkPoints2D, vtkPoints, vtkMatrix3x3, vtkMatrix4x4, vtkTensor`...
OGRE	3D Engine	`SmallVector, Matrix3, Matrix4`...
Irrlicht	3D Engine	`vector2d, vector3d, CMatrix4`...
OpenSceneGraph	3D Engine	`Vec2d, Vec3d, Vec4d, Matrix2, Matrix3`...
Panda3D	3D Engine	`LVector2d, LVector3d, LVector4d, LMatrix3d, LMatrix4d`...
OpenFOAM	Fluid Dynamics	`scalar, vector, tensor`...
Lorene	Astrophysics	`Scalar, Vector, Tensor`...
ROOT	Particle Physics	`DisplacementVector3D, DisplacementVector2D, PositionVector3D, PositionVector2D, Rotation3D, Vector3D, Point3D`...
And many others...	...	...

II Impact on the standard

This proposal is a pure addition to the existing library and consequently it would not affect existing programs. Nevertheless, it requires the addition of a new header shown in table 2.

Table 2: Summary of affected headers

Subclause	**Header(s)**
IV	`<mathematizer>`

Unless otherwise specified, all components described in this proposal would be declared in namespace `std`. Furthermore, unless otherwise specified, all references to components described in the C++ standard library are assumed to be qualified with `std::`. The design described in part III uses only C++11 features and do not require non-standard extension. The technical specifications presented in part IV would require an extensive use of the type traits of the standard library. No current element of the standard library would be modified by or would depend on the tools provided in the `mathematizer` header.

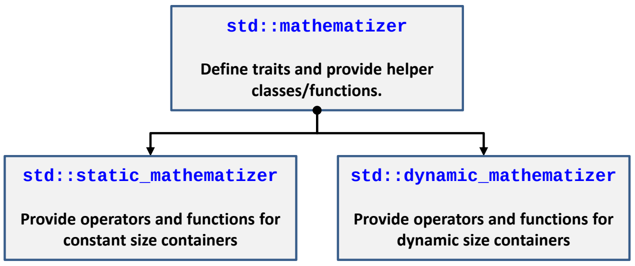
III Design decisions

The design presented in the following is based on original ideas introduced and tested during the development and implementation of the MAGRATHEA framework[1]. The philosophy is quite the same as the one of the `iterator` base class, but instead of providing a generic tool to simplify the creation of new iterators, the goal here is to provide a generic tool that simplifies the creation of new mathematical containers. To do so, the Curiously Recurring Template Pattern (CRTP) idiom is used. The resulting tools allow to provide all the standard arithmetic operators, comparators, element accessors and some modifier functions to a container just by inheriting it from one of the `mathematizer` classes. To keep the design as simple and as generic as possible, no multidimensional operation is provided (as the matrix multiplication) but it will be quite easy for the end user to add all the specific operators he wants to his classes. This ease of modification is a great advantage of the mechanism described in the following paragraphs and in the technical section.

The design of the `<mathematizer>` header consists of three classes as presented in figure 2:

- `mathematizer` is the base class, mainly used for type traits.
- `static_mathematizer` is the provided tool to mathematize constant size containers with the template shape `Container<typename, Integers...>` where `typename` represents the data type and where `Integers` is the integral type of extra parameters.
- `dynamic_mathematizer` is the provided tool to mathematize dynamic size containers with the template shape `Container<typename, Integers...>` where `typename` represents the data type and where `Integers` is the integral type of extra parameters.

Figure 2: Inheritance relation between the three classes of the `<mathematizer>` header.


std::mathematizer
Define traits and provide helper classes/functions.
std::static_mathematizer
Provide operators and functions for constant size containers
std::dynamic_mathematizer
Provide operators and functions for dynamic size containers

These three classes are abstract classes with protected destructors and cannot be used directly. Furthermore, they are empty classes in the sense that they only define methods: the actual data contents is owned by the derived classes and one should only provide an `access()` function to get the elements of the container.

The proposed design has two limitations. First, the inherited containers are required to have a specific template shape. These template parameters are compatible with standardized containers like `vector<typename>` or `array<typename, size_t>`. If one want to use another template shape, it can use alias templates or inherit from a container with the compatible template parameters. Second, one cannot use a mathematizer on a container of mathematized containers due to the use of type traits to distinguish between container/container and container/value operations. This is not a limiting factor for the creation of multidimensional arrays

[1]The MAGRATHEA (Multi-cpu Adaptive Grid Refinement Analysis for THEoretical Astrophysics) framework is under active development by V. Reverdy and is expected to be released publicly as an open source software in 2013.

as the `access()` member should provide an access to all elements regardless of any notion of dimension. These limitations, however, are a small price to pay to use generic tools that can automatically generate all the standard operators whatever the container and its contents are.

IV Technical specifications

V Examples of use

VI Acknowledgements

VII References