Toward a vectorization mechanism in C++

Document number: Draft
Version: 0.4
Date: 2012

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

December 27, 2012

Abstract

This proposal concerns the addition of three helper classes to the standard library in order to provide an easy-to-use vectorization mechanism. The goal is to reduce the current complexity of implementing new containers with optimized vector 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. Finally, this proposal should be able to fill the current lack of vector operations support 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 vectorized 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 mathematical 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 proprosed tool is not to diagonalize 100×100 matrices, or to solve sets of hundreds of equations. Consequently, we also avoid the never ending debate of sparse versus non-sparse mathematical containers. To summarize, the goal is to provide a generic mechanism that simplifies and unifies the design of new containers that need optimized standard operations.

This mechanism will reduce the long step of design, implementation and optimization of basic things like a `Vector3D` to the simple inheritance from a vectorizer class. A resulting example of use is provided in listing 1. Finally, these tools will have a wide range of use and will allow library designers, software developers, engineers and scientists to implement their own optimized vector containers without having to implement them from scratch.

Listing 1: Basic example of the vectorization syntax

```
1  template <typename T>
2  class MyVector3D
3  : public std::static_vectorizer<T, 3, size_t, MyVector3D>
4  {/* Creates a static size vector */};
5
6  template <typename T, size_t N>
7  class MyVectorN
8  : public std::static_vectorizer<T, N, size_t, MyVectorN, N>
9  {/* Creates a static size vector */};
10
11 template <typename T, size_t N, size_t M>
12 class MyMatrixNxM
13 : public std::static_vectorizer<T, N*M, size_t, MyMatrixNxM, N, M>
14 {/* Creates a static size matrix */};
15
16 template <typename T>
17 class MyVector
18 : public std::dynamic_vectorizer<T, size_t, MyVector>
19 {/* Creates a dynamic size vector */};
20
21 template <typename T>
22 class MyMatrix
23 : public std::dynamic_vectorizer<T, size_t, MyMatrix>
24 {/* Creates a dynamic size matrix */};
```

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

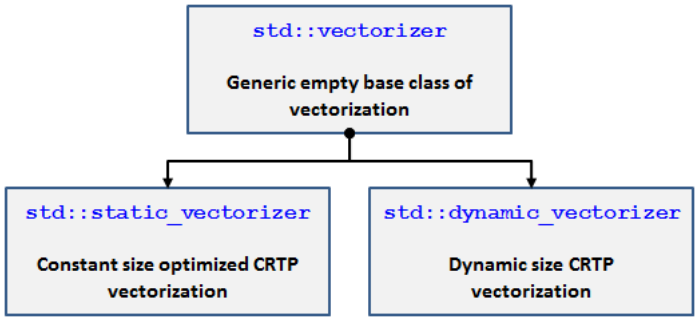
Subclause	Header(s)
IV	`<vectorizer>`

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 `vectorizer` 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` class that provides a generic tool to simplify the creation of new iterators. The goal here is to design helper classes that simplifies the creation of new vector 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 from one of the `vectorizer` classes and overloading some members as the subscript operator. 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 vectorized classes. This ease of modification is a great advantage of the mechanism described in the following paragraphs and in the technical section.

Figure 1: Inheritance relation between the three classes of the `<vectorizer>` header.



The design of the `<vectorizer>` header consists of three classes as presented in figure 1:

- `vectorizer` is the base class, mainly used for type traits.
- `static_vectorizer` is the provided tool to vectorize constant size optimized containers with the template shape `Crtp<typename, Kind...>` where `typename` represents the data type and where `Kind` is the integral type of extra arguments as a list of dimensions.
- `dynamic_vectorizer` is the provided tool to vectorize dynamic size containers with the template shape `Crtp<typename, Kind...>` where `typename` represents the data type and where `Kind` is the integral type of extra arguments as a rank.

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 and no data member: the actual data contents is owned by the derived classes and is accessed by the `vectorizer` classes thanks to the CRTP.

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 a container with the compatible template parameters, and inherit from it. Second, one cannot vectorize a container of vectorized containers due to the use of type traits to distinguish between vector/vector and vector/scalar operations. This is not a limiting factor for the creation of multidimensional containers if the

[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.

subscript operator allow to iterate over the whole contents and not only over a single dimension. These limitations, however, are a small price to pay to use generic tools that can automatically generate all the standard vector operators whatever the container and its contents are.

IV Technical specifications

In all the following, the expression *vectorizer class* refers to one the three classes of the `<vectorizer>` header (see table 3) and the term *vectorized* refers to any user-defined class inheriting from one of these classes.

Table 3: Summary of the new classes

Subclause	Header	Class name
IV.1	`<vectorizer>`	`vectorizer`
IV.2	`<vectorizer>`	`static_vectorizer`
IV.3	`<vectorizer>`	`dynamic_vectorizer`

Optimization through expression template or the use of SIMD instructions are not explicitely required by technical specifications and these decisions are left to implementers.

IV.1 The empty base class `std::vectorizer`

IV.1.1 Summary

The `vectorizer` class is an empty class that defines general-purpose functions for vectorization management. It plays an important role to detect vectorized containers through `is_base_of<vectorizer>` as it does not have any template parameter. Its contents is summarized in table 4. This class is not intended to be derived by the end-user: one should inherit a vectorized container from either `static_vectorizer` or `dynamic_vectorizer`.

Table 4: Summary of `vectorizer` contents

Section	Category	Specifier	Name	Description
IV.1.2	Class	`none`	`vectorizer`	class declaration
	Lifecycle	`public`	`(constructor)`	implements default construction
	Lifecycle	`protected`	`(destructor)`	protects the default destructor
	Lifecycle	`public`	`operator=`	deletes the assignment operator
	Member function	`public static`	`get`	gets vectorized/non-vectorized elements in an unified way
	Member function	`public static`	`set`	sets the contents of a vectorized container
	Member function	`public static`	`is_same_size`	compares the sizes of two containers
	Member function	`public static`	`check_size`	checks the sizes of two containers
	Member function	`public static`	`equal`	compares the contents of two containers (equality)
	Member function	`public static`	`not_equal`	compares the contents of two containers (difference)
	Member function	`public static`	`combine`	combines several vectorized containers through a reduction operation
	Non-member function	`friend`	`swap`	specializes the `swap` algorithm
	Non-member function	`friend`	`begin`	specializes the `begin` function
	Non-member function	`friend`	`end`	specializes the `end` function
	Non-member function	`friend`	`operator<<`	performs stream I/O (output)
	Non-member function	`friend`	`operator>>`	performs stream I/O (input)

IV.1.2 Declaration

The `vectorizer` class should have a declaration equivalent to:

```
1 class vectorizer
```

IV.2 The constant size vectorization tool `std::static_vectorizer`

IV.2.1 Summary

The `static_vectorizer` class is an empty class that vectorizes operations over constant size containers thanks to the CRTP idiom. Its contents is summarized in tables 5, 6 and 7.

Table 5: Summary of `static_vectorizer` declaration, traits and constants

Section	Category	Specifier	Name	Description
IV.2.2	Class	`none`	`static_vectorizer`	declaration
IV.2.3	Member type	`public`	`value_type`	`Type`
IV.2.3	Member type	`public`	`size_type`	Unsigned integral type
IV.2.3	Member type	`public`	`difference_type`	Signed integral type
IV.2.3	Member type	`public`	`reference`	`value_type&`
IV.2.3	Member type	`public`	`const_reference`	`const reference`
IV.2.3	Member type	`public`	`pointer`	`value_type*`
IV.2.3	Member type	`public`	`const_pointer`	`const pointer`
IV.2.3	Member type	`public`	`vectorizer_type`	`static_vectorizer<Type, Size, Kind, Crtp, Params...>`
IV.2.3	Member type	`public`	`vectorized_type`	`Crtp<Type, Params...>`
IV.2.3	Member type	`public`	`params_type`	`Kind`
IV.2.4	Constant	`public static const bool`	`is_static`	`true`
IV.2.4	Constant	`public static const bool`	`is_dynamic`	`false`
IV.2.4	Constant	`public static const bool`	`is_mask`	`is_same<Type, bool>::value`
IV.2.4	Constant	`public static const Kind[]`	`params`	`Params...`

Table 6: Summary of `static_vectorizer` arithmetic operators and comparators

Section	Category	Specifier	Name	Description

IV.2.5	Member function	`public`	`operator+`
			`operator-`
			`operator~`
			`operator!`
			`operator++`
			`operator++(int)`
			`operator--`
			`operator--(int)`
				applies unary operator to each element of the vectorized container

IV.2.6	Member function	`public`	`operator+=`
			`operator-=`
			`operator*=`
			`operator/=`
			`operator%=`
			`operator&=`
			`operator|=`
			`operator^=`
			`operator<<=`
			`operator>>=`
				applies compound assignment operator to each element of the vectorized container

IV.2.7	Non-member function	`friend`	`operator+`
			`operator-`
			`operator*`
			`operator/`
			`operator%`
			`operator&`
			`operator|`
			`operator^`
			`operator<<`
			`operator>>`
			`operator&&`
			`operator||`
				applies binary operators to each element of two vectorized containers, or a vectorized container and a value

IV.2.8	Non-member function	`friend`	`operator==`
			`operator!=`
			`operator<`
			`operator>`
			`operator<=`
			`operator>=`
				compares two vectorized containers or a vectorized container with a value

Table 7: Summary of `static_vectorizer` methods

Section	Category	Specifier	Name	Description
	Lifecycle	`public`	`(constructor)`	implements construction
	Lifecycle	`protected`	`(destructor)`	protects the default destructor
	Lifecycle	`public`	`operator=`	assigns contents
	Lifecycle	`public`	`assign`	assigns contents
	Member function	`public`	`operator[]`	calls the CRTP subscript operator
	Member function	`public`	`size`	returns the size of the container
	Member function	`public`	`resize`	throws runtime exception (only for compatibility purposes)
	Member function	`public`	`empty`	checks whether the vectorized container is empty
	Member function	`public`	`at`	access specified element with bounds checking
	Member function	`public`	`cat`	circularly access specified element
	Member function	`public`	`front`	access the first element
	Member function	`public`	`back`	access the last element
	Member function	`public`	`index`	gets the index of a reference
	Member function	`public`	`cast`	casts contents to another data type
	Member function	`public`	`swap`	swaps with another container
	Member function	`public`	`fill`	assigns some contents to the container
	Member function	`public`	`change`	assigns some contents to a copy of the container
	Member function	`public`	`modify`	applies a function to every element of the container
	Member function	`public`	`apply`	applies a function to every element of a copy of the container
	Member function	`public`	`min`	returns a reference to the first minimum value
	Member function	`public`	`max`	returns a reference to the first maximum value
	Member function	`public`	`reduce`	applies a reduction operation over the container elements
	Member function	`public static`	`mask`	constructs a boolean mask

IV.2.2 Declaration

The `static_vectorizer` class should have a declaration equivalent to:

```
template <typename Type,
          size_t Size,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params>
class static_vectorizer
: public vectorizer
```

where:

- `Type` is the container data type.
- `Size` is the constant number of elements of the container.
- `Kind` is an integral type (oftenly `size_t`).
- `Crtp` is the derived container type.
- `Params...` is a list of parameters (e.g. a list of dimensions for a constant size multidimensional array).

Furthermore, the vectorized container `Crtp` should fulfill the following requirements:

- it has to inherit from `static_vectorizer<Type, Size, Kind, Crtp, Params...>`.
- its template shape has to be `Crtp<typename, Kind...>` where `Kind` is an integral type.
- its size has to be known at compile-time and passed as the `Size` template parameter.
- it has to provide the subscript `operator[]` in both const/non-const versions.

IV.2.3 Typedef

The `static_vectorizer` class defines the following public types:

```
typedef Type value_type;
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef value_type& reference;
typedef const reference const_reference;
typedef value_type* pointer;
typedef const pointer const_pointer;
typedef static_vectorizer<Type, Size, Kind, Crtp, Params...> vectorizer_type;
typedef Crtp<Type, Params...> vectorized_type;
typedef static_vectorizer<bool, Size, Kind, Crtp, Params...>
    vectorizer_mask_type;
typedef Crtp<bool, Params...> vectorized_mask_type;
typedef Kind params_type;
```

IV.2.4 Constants

The `static_vectorizer` class defines the following public constants:

```
static const bool is_static = true;
static const bool is_dynamic = false;
static const bool is_mask = std::is_same<Type, bool>::value;
static const Kind params[sizeof...(Params)] = {Params...};
```

IV.2.5 Unary operators

The unary operators of a `static_vectorizer` are:

```
1  Crtp<Type, Params...> operator+() const;
2  Crtp<Type, Params...> operator-() const;
3  Crtp<Type, Params...> operator~() const;
4  Crtp<bool, Params...> operator!() const;
5  Crtp<Type, Params...>& operator++() const;
6  Crtp<Type, Params...> operator++(int) const;
7  Crtp<Type, Params...>& operator--() const;
8  Crtp<Type, Params...> operator--(int) const;
```

IV.2.6 Compound assignment operators

The compound assignment operators of a `static_vectorizer` are defined in two versions.

The first one is for vectorized right-hand side types and is preceded by:

```
1  template <typename T,
2            class = typename enable_if<is_base_of<vectorizer, T>::value>::type>
```

The compound assignment operators of this first version are:

```
1  Crtp<Type, Params...>& operator+= (const T& rhs) const;
2  Crtp<Type, Params...>& operator-= (const T& rhs) const;
3  Crtp<Type, Params...>& operator*= (const T& rhs) const;
4  Crtp<Type, Params...>& operator/= (const T& rhs) const;
5  Crtp<Type, Params...>& operator%= (const T& rhs) const;
6  Crtp<Type, Params...>& operator&= (const T& rhs) const;
7  Crtp<Type, Params...>& operator|= (const T& rhs) const;
8  Crtp<Type, Params...>& operator^= (const T& rhs) const;
9  Crtp<Type, Params...>& operator<<= (const T& rhs) const;
10 Crtp<Type, Params...>& operator>>= (const T& rhs) const;
```

The second one is for non-vectorized right-hand side types and is preceded by:

```
1  template <typename T,
2            class = typename enable_if<!is_base_of<vectorizer, T>::value>::type>
```

The compound assignment operators of this second version are:

```
1  Crtp<Type, Params...>& operator+= (const T& rhs) const;
2  Crtp<Type, Params...>& operator-= (const T& rhs) const;
3  Crtp<Type, Params...>& operator*= (const T& rhs) const;
4  Crtp<Type, Params...>& operator/= (const T& rhs) const;
5  Crtp<Type, Params...>& operator%= (const T& rhs) const;
6  Crtp<Type, Params...>& operator&= (const T& rhs) const;
7  Crtp<Type, Params...>& operator|= (const T& rhs) const;
8  Crtp<Type, Params...>& operator^= (const T& rhs) const;
9  Crtp<Type, Params...>& operator<<= (const T& rhs) const;
10 Crtp<Type, Params...>& operator>>= (const T& rhs) const;
```

IV.2.7 Binary operators

The binary operators of a `static_vectorizer` are defined in three versions.

The first one is for vectorized left-hand side and right-hand side types and is preceded by:

```
template <typename T1,
          typename T2,
          size_t Size,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params>
```

The binary operators of this first version are:

```
Crtp<typename common_type<T1, T2>::type, Params...> operator+
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator-
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator*
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator/
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator%
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator&
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator|
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator^
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<T1, Params...> operator<<
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<T1, Params...> operator>>
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator&&
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator||
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);
```

The second one is for non-vectorized left-hand side type and vectorized right-hand side type and is preceded by:

```
template <typename T1,
          typename T2,
          size_t Size,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params,
          class = typename enable_if<!is_base_of<vectorizer, T1>::value>::type>
```

The binary operators of this second version are:

```
Crtp<typename common_type<T1, T2>::type, Params...> operator+
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator-
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator*
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator/
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator%
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator&
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator|
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator^
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<T1, Params...> operator<<
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<T1, Params...> operator>>
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator&&
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator||
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);
```

The third one is for vectorized left-hand side type and non-vectorized right-hand side type and is preceded by:

```
template <typename T1,
          typename T2,
          size_t Size,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params
          class = typename enable_if<!is_base_of<vectorizer, T2>::value>::type>
```

The binary operators of this third version are:

```
Crtp<typename common_type<T1, T2>::type, Params...> operator+
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator-
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator*
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator/
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator%
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator&
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator|
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator^
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<T1, Params...> operator<<
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<T1, Params...> operator>>
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<bool, Params...> operator&&
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<bool, Params...> operator||
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);
```

IV.2.8 Comparison operators

The comparison operators of a `static_vectorizer` are defined in three versions.

The first one is for vectorized left-hand side and right-hand side types and is preceded by:

```
template <typename T1,
          typename T2,
          size_t Size,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params>
```

The comparison operators of this first version are:

```
Crtp<bool, Params...> operator==
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator!=
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator<
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator>
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator<=
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator>=
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);
```

The second one is for non-vectorized left-hand side type and vectorized right-hand side type and is preceded by:

```
template <typename T1,
          typename T2,
          size_t Size,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params,
          class = typename enable_if<!is_base_of<vectorizer, T1>::value>::type>
```

The comparison operators of this second version are:

```
Crtp<bool, Params...> operator==
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator!=
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator<
```

```
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator>
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator<=
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator>=
(const T1& lhs,
 const static_vectorizer<T2, Size, Kind, Crtp, Params...>& rhs);
```

The third one is for vectorized left-hand side type and non-vectorized right-hand side type and is preceded by:

```
template <typename T1,
          typename T2,
          size_t Size,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params,
          class = typename enable_if<!is_base_of<vectorizer, T2>::value>::type>
```

The comparison operators of this third version are:

```
Crtp<bool, Params...> operator==
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& lhs);

Crtp<bool, Params...> operator!=
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<bool, Params...> operator<
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<bool, Params...> operator>
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<bool, Params...> operator<=
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<bool, Params...> operator>=
(const static_vectorizer<T1, Size, Kind, Crtp, Params...>& lhs,
 const T2& rhs);
```

IV.3 The dynamic size vectorization tool `std::dynamic_vectorizer`

IV.3.1 Summary

The `dynamic_vectorizer` class is an empty class that vectorizes operations over dynamic size containers thanks to the CRTP idiom. Its contents is summarized in tables 8, 9 and 10.

Table 8: Summary of `dynamic_vectorizer` declaration, traits and constants

Section	Category	Specifier	Name	Description
IV.3.2	Class	`none`	`dynamic_vectorizer`	declaration
IV.3.3	Member type	`public`	`value_type`	`Type`
IV.3.3	Member type	`public`	`size_type`	Unsigned integral type
IV.3.3	Member type	`public`	`difference_type`	Signed integral type
IV.3.3	Member type	`public`	`reference`	`value_type&`
IV.3.3	Member type	`public`	`const_reference`	`const reference`
IV.3.3	Member type	`public`	`pointer`	`value_type*`
IV.3.3	Member type	`public`	`const_pointer`	`const pointer`
IV.3.3	Member type	`public`	`vectorizer_type`	`dynamic_vectorizer<Type, Kind, Crtp, Params...>`
IV.3.3	Member type	`public`	`vectorized_type`	`Crtp<Type, Params...>`
IV.3.3	Member type	`public`	`params_type`	`Kind`
IV.3.4	Constant	`public static const bool`	`is_static`	`false`
IV.3.4	Constant	`public static const bool`	`is_dynamic`	`true`
IV.3.4	Constant	`public static const bool`	`is_mask`	`is_same<Type, bool>::value`
IV.3.4	Constant	`public static const Kind[]`	`params`	`Params...`

Table 9: Summary of `dynamic_vectorizer` arithmetic operators and comparators

Section	Category	Specifier	Name	Description

IV.3.5	Member function	`public`	`operator+`
			`operator-`
			`operator~`
			`operator!`
			`operator++`
			`operator++(int)`
			`operator--`
			`operator--(int)`
				applies unary operator to each element of the vectorized container

IV.3.6	Member function	`public`	`operator+=`
			`operator-=`
			`operator*=`
			`operator/=`
			`operator%=`
			`operator&=`
			`operator|=`
			`operator^=`
			`operator<<=`
			`operator>>=`
				applies compound assignment operator to each element of the vectorized container

IV.3.7	Non-member function	`friend`	`operator+`
			`operator-`
			`operator*`
			`operator/`
			`operator%`
			`operator&`
			`operator|`
			`operator^`
			`operator<<`
			`operator>>`
			`operator&&`
			`operator||`
				applies binary operators to each element of two vectorized containers, or a vectorized container and a value

IV.3.8	Non-member function	`friend`	`operator==`
			`operator!=`
			`operator<`
			`operator>`
			`operator<=`
			`operator>=`
				compares two vectorized containers or a vectorized container with a value

Table 10: Summary of `dynamic_vectorizer` methods

Section	Category	Specifier	Name	Description
	Lifecycle	`public`	`(constructor)`	implements construction
	Lifecycle	`protected`	`(destructor)`	protects the default destructor
	Lifecycle	`public`	`operator=`	assigns contents
	Lifecycle	`public`	`assign`	assigns contents
	Member function	`public`	`operator[]`	calls the CRTP subscript operator
	Member function	`public`	`size`	calls the CRTP `size()` function
	Member function	`public`	`resize`	calls the CRTP `resize()` function
	Member function	`public`	`empty`	checks whether the vectorized container is empty
	Member function	`public`	`at`	access specified element with bounds checking
	Member function	`public`	`cat`	circularly access specified element
	Member function	`public`	`front`	access the first element
	Member function	`public`	`back`	access the last element
	Member function	`public`	`index`	gets the index of a reference
	Member function	`public`	`cast`	casts contents to another data type
	Member function	`public`	`swap`	swaps with another container
	Member function	`public`	`fill`	assigns some contents to the container
	Member function	`public`	`change`	assigns some contents to a copy of the container
	Member function	`public`	`modify`	applies a function to every element of the container
	Member function	`public`	`apply`	applies a function to every element of a copy of the container
	Member function	`public`	`min`	returns a reference to the first minimum value
	Member function	`public`	`max`	returns a reference to the first maximum value
	Member function	`public`	`reduce`	applies a reduction operation over the container elements
	Member function	`public static`	`mask`	constructs a boolean mask

IV.3.2 Declaration

The `dynamic_vectorizer` class should have a declaration equivalent to:

```
template <typename Type,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params>
class dynamic_vectorizer
: public vectorizer
```

where:

- `Type` is the container data type.
- `Kind` is an integral type (oftenly `size_t`).
- `Crtp` is the derived container type.
- `Params...` is a list of parameters (e.g. a rank for a dynamic size multidimensional array).

Furthermore, the vectorized container `Crtp` should fulfill the following requirements:

- it has to inherit from `dynamic_vectorizer<Type, Kind, Crtp, Params...>`.
- its template shape has to be `Crtp<typename, Kind...>` where `Kind` is an integral type.
- it has to provide a `size()` method that returns the current size of the container.
- it has to provide a `resize()` method that modifies the size of the container.
- it has to provide the subscript `operator[]` in both const/non-const versions.

IV.3.3 Typedef

The `dynamic_vectorizer` class defines the following public types:

```
typedef Type value_type;
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef value_type& reference;
typedef const reference const_reference;
typedef value_type* pointer;
typedef const pointer const_pointer;
typedef dynamic_vectorizer<Type, Kind, Crtp, Params...> vectorizer_type;
typedef Crtp<Type, Params...> vectorized_type;
typedef dynamic_vectorizer<bool, Kind, Crtp, Params...> vectorizer_mask_type;
typedef Crtp<bool, Params...> vectorized_mask_type;
typedef Kind params_type;
```

IV.3.4 Constants

The `dynamic_vectorizer` class defines the following public constants:

```
static const bool is_static = false;
static const bool is_dynamic = true;
static const bool is_mask = std::is_same<Type, bool>::value;
static const Kind params[sizeof...(Params)] = {Params...};
```

IV.3.5 Unary operators

The unary operators of a `dynamic_vectorizer` are:

```
Crtp<Type, Params...> operator+() const;
Crtp<Type, Params...> operator-() const;
Crtp<Type, Params...> operator~() const;
Crtp<bool, Params...> operator!() const;
Crtp<Type, Params...>& operator++() const;
Crtp<Type, Params...> operator++(int) const;
Crtp<Type, Params...>& operator--() const;
Crtp<Type, Params...> operator--(int) const;
```

IV.3.6 Compound assignment operators

The compound assignment operators of a `dynamic_vectorizer` are defined in two versions.

The first one is for vectorized right-hand side types and is preceded by:

```
template <typename T,
          class = typename enable_if<is_base_of<vectorizer, T>::value>::type>
```

The compound assignment operators of this first version are:

```
Crtp<Type, Params...>& operator+= (const T& rhs) const;
Crtp<Type, Params...>& operator-= (const T& rhs) const;
Crtp<Type, Params...>& operator*= (const T& rhs) const;
Crtp<Type, Params...>& operator/= (const T& rhs) const;
Crtp<Type, Params...>& operator%= (const T& rhs) const;
Crtp<Type, Params...>& operator&= (const T& rhs) const;
Crtp<Type, Params...>& operator|= (const T& rhs) const;
Crtp<Type, Params...>& operator^= (const T& rhs) const;
Crtp<Type, Params...>& operator<<= (const T& rhs) const;
Crtp<Type, Params...>& operator>>= (const T& rhs) const;
```

The second one is for non-vectorized right-hand side types and is preceded by:

```
template <typename T,
          class = typename enable_if<!is_base_of<vectorizer, T>::value>::type>
```

The compound assignment operators of this second version are:

```
Crtp<Type, Params...>& operator+= (const T& rhs) const;
Crtp<Type, Params...>& operator-= (const T& rhs) const;
Crtp<Type, Params...>& operator*= (const T& rhs) const;
Crtp<Type, Params...>& operator/= (const T& rhs) const;
Crtp<Type, Params...>& operator%= (const T& rhs) const;
Crtp<Type, Params...>& operator&= (const T& rhs) const;
Crtp<Type, Params...>& operator|= (const T& rhs) const;
Crtp<Type, Params...>& operator^= (const T& rhs) const;
Crtp<Type, Params...>& operator<<= (const T& rhs) const;
Crtp<Type, Params...>& operator>>= (const T& rhs) const;
```

IV.3.7 Binary operators

The binary operators of a `dynamic_vectorizer` are defined in three versions.

The first one is for vectorized left-hand side and right-hand side types and is preceded by:

```
template <typename T1,
          typename T2,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params>
```

The binary operators of this first version are:

```
Crtp<typename common_type<T1, T2>::type, Params...> operator+
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator-
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator*
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator/
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator%
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator&
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator|
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator^
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<T1, Params...> operator<<
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<T1, Params...> operator>>
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator&&
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator||
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);
```

The second one is for non-vectorized left-hand side type and vectorized right-hand side type and is preceded by:

```
template <typename T1,
          typename T2,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params,
          class = typename enable_if<!is_base_of<vectorizer, T1>::value>::type>
```

The binary operators of this second version are:

```
Crtp<typename common_type<T1, T2>::type, Params...> operator+
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator-
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator*
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator/
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator%
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator&
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator|
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator^
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<T1, Params...> operator<<
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<T1, Params...> operator>>
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator&&
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator||
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);
```

The third one is for vectorized left-hand side type and non-vectorized right-hand side type and is preceded by:

```
template <typename T1,
          typename T2,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params
          class = typename enable_if<!is_base_of<vectorizer, T2>::value>::type>
```

The binary operators of this third version are:

```
Crtp<typename common_type<T1, T2>::type, Params...> operator+
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator-
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator*
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator/
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator%
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator&
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator|
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<typename common_type<T1, T2>::type, Params...> operator^
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<T1, Params...> operator<<
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<T1, Params...> operator>>
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<bool, Params...> operator&&
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);

Crtp<bool, Params...> operator||
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const T2& rhs);
```

IV.3.8 Comparison operators

The comparison operators of a `dynamic_vectorizer` are defined in three versions.

The first one is for vectorized left-hand side and right-hand side types and is preceded by:

```
template <typename T1,
          typename T2,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params>
```

The comparison operators of this first version are:

```
Crtp<bool, Params...> operator==
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator!=
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator<
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator>
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator<=
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator>=
(const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);
```

The second one is for non-vectorized left-hand side type and vectorized right-hand side type and is preceded by:

```
template <typename T1,
          typename T2,
          typename Kind,
          template <typename, Kind...> class Crtp,
          Kind... Params,
          class = typename enable_if<!is_base_of<vectorizer, T1>::value>::type>
```

The comparison operators of this second version are:

```
Crtp<bool, Params...> operator==
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator!=
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);

Crtp<bool, Params...> operator<
(const T1& lhs,
 const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);
```

```
12
13 Crtp<bool, Params...> operator>
14 (const T1& lhs,
15  const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);
16
17 Crtp<bool, Params...> operator<=
18 (const T1& lhs,
19  const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);
20
21 Crtp<bool, Params...> operator>=
22 (const T1& lhs,
23  const dynamic_vectorizer<T2, Kind, Crtp, Params...>& rhs);
```

The third one is for vectorized left-hand side type and non-vectorized right-hand side type and is preceded by:

```
1 template <typename T1,
2           typename T2,
3           typename Kind,
4           template <typename, Kind...> class Crtp,
5           Kind... Params,
6           class = typename enable_if<!is_base_of<vectorizer, T2>::value>::type>
```

The comparison operators of this third version are:

```
1  Crtp<bool, Params...> operator==
2  (const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
3   const T2& lhs);
4
5  Crtp<bool, Params...> operator!=
6  (const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
7   const T2& rhs);
8
9  Crtp<bool, Params...> operator<
10 (const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
11  const T2& rhs);
12
13 Crtp<bool, Params...> operator>
14 (const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
15  const T2& rhs);
16
17 Crtp<bool, Params...> operator<=
18 (const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
19  const T2& rhs);
20
21 Crtp<bool, Params...> operator>=
22 (const dynamic_vectorizer<T1, Kind, Crtp, Params...>& lhs,
23  const T2& rhs);
```

V Examples of use

V.1 A minimal example

Using a `static_vectorizer`, one can easily create 3D vectors and matrices that implement all operators:

```
// Vector 3D implementation
template <typename T>
class Vector3D : public std::static_vectorizer<T, 3, size_t, Vector3D>
{
    public:
        template<typename U> Vector3D (const Vector3D<U>& x)
        {for(size_t i = 0; i < _data.size(); ++i) _data[i] = x[i];}
        explicit Vector3D(const T& x) {_data.fill(x);}

        T& operator[](size_t i) {return _data[i];}
        const T& operator[](size_t i) const {return _data[i];}

    protected: std::array<T, 3> _data;
};

// Matrix 3D implementation
template <typename T>
class Matrix3D : public std::static_vectorizer<T, 9, size_t, Matrix3D>
{
    public:
        template<typename U> Matrix3D (const Matrix3D<U>& x)
        {for(size_t i = 0; i < _data.size(); ++i) _data[i] = x[i];}
        explicit Matrix3D(const T& x) {_data.fill(x);}

        T& operator()(size_t i, size_t j) {return _data[i*3+j];}
        const T& operator()(size_t i, size_t j) const {return _data[i*3+j];}
        T& operator[](size_t i) {return _data[i];}
        const T& operator[](size_t i) const {return _data[i];}

    protected: std::array<T, 9> _data;
};
```

These two new classes can be used in that way:

```
int main(int argc, char* argv[])
{
    // Initialization
    Vector3D<double> v0, v1(4.), v2(8.);
    Vector3D<bool> vb;
    Matrix3D<double> m0, m1(15.), m2(16.);
    Matrix3D<bool> mb;
    std::iota(std::begin(v0), std::end(v0), 23.);
    std::iota(std::begin(m0), std::end(m0), 42.);

    // Basic operations
    v2 = 1+v0+v1*v2/v0-v0-4+5;
    m2 = 1+m0+m1*m2/m0-m0*4+5;
    v0 *= 3;
    m0 *= 3;
    vb = v1 < v2;
    mb = m1 < m2;
    return 0;
}
```

VI Acknowledgements

VII References