

ERC CISST Software

Quick start for cisstVector

Anton Deguet

Date: 2008/02/06 20:37:30

Contents

1 Introduction	1
2 Manipulating fixed size vectors and frames	2
3 Manipulating dynamic vectors and matrices	3
4 Container slices and vector references	5
5 Manipulating multidimensional arrays	8
6 Using the C++ Standard Template Library	10
7 Using cisstCommon	10
8 Using cisstNumerical	12
9 Writing your own functions	12
9.1 Using dynamic containers	13
9.2 Using fixed size containers	14

1 Introduction

These examples provide a quick introduction to the features of cisstVector. The code is part of the CVS module `cisst/examples/vectorTutorial`. To compile your own code, remember to include `cisstVector.h`.

2 Manipulating fixed size vectors and frames

```

1 void ExampleFrame(void) {
2     // create 3 points
3     vct3 point000(0.0, 0.0, 0.0);
4     vct3 point100(3.0, 0.0, 0.0);
5     vct3 point110(2.0, 3.2, 0.0);
6
7     // create a normalized vector along the axis X
8     // using methods
9     vct3 axisX;
10    axisX.DifferenceOf(point100, point000);
11    axisX.Divide(axisX.Norm());
12
13    // create a normalized vector along the axis Z
14    // using operators '-' for difference,
15    // '%' for cross product, and '/=' for in-place
16    // elementwise division.
17    vct3 tmp = point110 - point000;
18    vct3 axisZ = axisX % tmp;
19    axisZ /= axisZ.Norm();
20
21    // Using named methods instead of operators
22    vct3 axisY;
23    axisY.CrossProductOf(axisZ, axisX);
24    axisY.NormalizedSelf();
25
26    /* three ways to display the result: */
27    // 1. Just output a vector
28    cout << "X:␣" << axisX << endl;
29    // 2. Output vector component by index
30    cout << "Y:␣" << axisY[0]
31         << "␣" << axisY[1]
32         << "␣" << axisY[2] << endl;
33    // 3. Output vector component by "name"
34    cout << "Z:␣" << axisZ.X()
35         << "␣" << axisZ.Y()
36         << "␣" << axisZ.Z() << endl;
37    /**/
38
39    // create a rotation along axis "tmp"
40    tmp.Assign(1.3, -0.3, 1.7);
41    tmp.NormalizedSelf();
42    vctMatRot3 rotation(vctAxAnRot3(tmp, 3.1415 / 2.0));
43
44    /* two ways to apply the rotation
45       to vectors: */
46    vct3 newAxisX, newAxisY, newAxisZ;
47    // 1. Using operator '*'
48    newAxisX = rotation * axisX;
49    // 2. Using named method ApplyTo

```

```

50     rotation.ApplyTo(axisY, newAxisY);
51     rotation.ApplyTo(axisZ, newAxisZ);
52
53     /* verify that the transformed vectors are still
54        an orthogonal basis. Compute dot products
55        in three ways. */
56     // 1. Using operator * on two vectors
57     double dotXY = newAxisX * newAxisY;
58     // 2. Using global function vctDotProduct
59     double dotYZ = vctDotProduct(newAxisY, newAxisZ);
60     // 3. Using named method DotProduct
61     double dotZX = newAxisZ.DotProduct(newAxisX);
62
63     cout << "Dot products:_" << dotXY << "_"
64           << dotYZ << "_" << dotZX << endl;
65     /**/
66
67     // create a rigid transformation frame from
68     // a rotation matrix and a translation vector
69     vct3 translation(0.0, 3.2, 4.5);
70     vctFrm3 frame(rotation, translation);
71     // Apply the frame to a vector
72     vct3 frameOnX = frame * axisX;
73     cout << "Image_of_X:_" << frameOnX << endl;
74
75     // inverse of the frame
76     vctFrm3 inverse;
77     inverse.InverseOf(frame);
78     // The product of a frame and its inverse
79     // should be the identity (eye for rotation,
80     // zero for translation).
81     cout << "frame*_inverse:_" << endl << frame * inverse
82           << endl;
83     // Compare this with the actual identity frame
84     cout << "Identity_frame:_" << endl
85           << vctFrm3::Identity() << endl;
86 }

```

In the example, we used some fixed size vector of 3 doubles (`vct3`) and some of the methods and operators available for this class (`Norm()`, `DifferenceOf`, `CrossProductOf`, operators `-`, `%`, etc.).

We also introduced a rotation matrix and a frame which can be used with the `cisst` fixed size vectors (`vct3`, same as `vctDouble3`). For more information related to transformations, see the [cisstVector User Guide](#)

3 Manipulating dynamic vectors and matrices

```

1 void ExampleDynamic(void) {
2     // define our preferred types

```

```

3     typedef vctDynamicVector<double> VectorType;
4     typedef vctDynamicMatrix<double> MatrixType;
5
6     // The dynamic vector library may throw exceptions,
7     // (derived from std::exception)
8     // so we place the operations in a try-catch block.
9     try {
10        // create an empty vector
11        VectorType vector1;
12        cout << "Size of vector1:" << vector1.size() << endl;
13
14        // resize and fill the vector
15        unsigned int index;
16        vector1.SetSize(5);
17        for (index = 0; index < vector1.size(); index++) {
18            vector1[index] = index;
19        }
20        // look at product of elements
21        cout << "Product of elements is 0?"
22            << vector1.ProductOfElements() << endl;
23
24        // create a matrix initialized with zeros
25        MatrixType matrix1(7, vector1.size());
26        matrix1.SetAll(0.0);
27
28        // set the diagonal to 5.0
29        matrix1.Diagonal().SetAll(5.0);
30
31        // look at the sum/product of elements
32        cout << "Sum of elements is 25?"
33            << matrix1.SumOfElements() << endl;
34
35        // multiply matrix1 by vector 2
36        VectorType vector2(matrix1.rows());
37        vector2.ProductOf(matrix1, vector1);
38
39        // multiply vector1 directly
40        VectorType vector3(vector1.size());
41        vector3.ProductOf(5.0, vector1);
42
43        // resize vector2 while preserving the data
44        vector2.resize(vector3.size());
45
46        // vector2 and vector3 are the same?
47        VectorType difference;
48        difference = vector3 - vector2;
49        difference.AbsSelf();
50        cout << "Maximum difference between v2 and v3:"
51            << difference.MaxElement() << endl;
52        // alternative solution
53        cout << "Maximum difference between v2 and v3:"

```

```

54         << (vector3 - vector2).MaxAbsElement() << endl;
55
56     } // end of try block
57     // catch block
58     catch (std::exception Exception) {
59         cerr << "Exception occurred: " << Exception.what() << endl;
60     }
61 }
62

```

In this example, we created a couple of dynamic vectors as well as a dynamic matrix. Dynamic containers are convenient for large collections of data, or when the number of elements is provided during runtime. Since the allocation is dynamic, it is important to check that the sizes of the operands are compatible. Our library throws exceptions, derived from the Standard Library `std::exception` class, on illegal operation arguments, such as unmatching vectors or out-of-range element access. This is why we use a `try` and `catch` structure.

The space allocated for a vector or a matrix can be changed in two ways. `SetSize` discards any old data and allocated memory in the specified size. `resize` preserves the old data by first allocating new space and then copying the elements from the old space to the new one. The `Diagonal` method is a first example of manipulating *container slices*, or *vector references*. The concept is demonstrated in the next example code.

4 Container slices and vector references

```

1 void ExampleReference(void) {
2     // define our preferred type
3     typedef vctDynamicMatrix<int> MatrixType;
4
5     try {
6         // create a matrix filled with zero
7         MatrixType matrix(8, 6);
8         matrix.SetAll(0);
9
10        // create a reference to column 3 (4th column
11        // from zero-base)
12        MatrixType::ColumnRefType col3 = matrix.Column(3);
13        col3.SetAll(2);
14
15        // create a reference to row 0
16        MatrixType::RowRefType row0 = matrix.Row(0);
17        row0.SetAll(3);
18
19        // create a reference to the last row
20        MatrixType::RowRefType rowLast = matrix[matrix.rows() - 1];
21        rowLast.SetAll(4);
22
23        // create a reference to the diagonal

```

```

24     MatrixType::DiagonalRefType diagonal = matrix.Diagonal();
25     diagonal.SetAll(1);
26
27     // create a sub matrix
28     MatrixType::Submatrix::Type submatrix(matrix,
29                                           /* from */ 3, 1,
30                                           /* size */ 4, 2);
31     submatrix += 6;
32
33     // display the result
34     cout << "Matrix modified by pieces:" << endl
35          << matrix << endl;
36     cout << "Trace:" << diagonal.SumOfElements() << endl;
37
38     } catch (std::exception Exception) {
39         cout << "Exception received:" << Exception.what() << cout;
40     }
41 }

```

This example demonstrates the use of slices through a dynamic matrix. The term “slice” refers to a contiguous subregion of a larger vector or matrix. In the example, we directly address columns, rows, and a diagonal of the large matrix `matrix` through the methods `Column`, `Row` and `Diagonal`. We use the word `Ref` to indicate a vector or matrix object that doesn’t allocate its own memory, but overlays another object’s memory, such as a slice in a matrix. These slices have appropriately named types, which are `ColumnRefType`, `RowRefType`, and `DiagonalRefType`.

Next, we define a submatrix slice, using the type `MatrixType::Submatrix::Type` (the reason for this notation will be given soon). The constructor takes the location of the first element of the submatrix in the large matrix, and the dimensions of the submatrix. As we can see, we can operate on the submatrix just as we do on any matrix.

The next example shows how to use slices with fixed-size vectors and matrices. In the example, we allocate a 4×4 homogeneous transformation matrix, and relate to parts of it as a rotation component and a translation component.

```

1 void ExampleReferenceFixedSize(void) {
2     // define our preferred type
3     typedef vctFixedSizeMatrix<float,4,4> MatrixType;
4
5     try {
6         // create a matrix initialized as identity
7         MatrixType matrix(0.0f);
8         matrix.Diagonal().SetAll(1.0f);
9
10        // create a rotation matrix of 30deg about the
11        // X axis
12        vctAxAnRot3 axRot30(vct3(1.0, 0.0, 0.0), (3.14159265 / 6.0));
13        vctMatRot3 rot30( axRot30 );
14

```

```

15         // Assign the rotation to the upper-left
16         // 3x3 part of our matrix
17         MatrixType::Submatrix<3,3>::Type rotSubmatrix(matrix, 0, 0);
18         rotSubmatrix.Assign( rot30 );
19
20         // Create reference to the last column
21         MatrixType::ColumnRefType lastCol = matrix.Column(3);
22
23         /**/
24         // Create a slice of length 3 of the last column
25         // NOTE: MSVC7 does not allow the following definition:
26         //       MatrixType::ColumnRefType::Subvector<3>::Type translation;
27         // but gcc does.
28         typedef MatrixType::ColumnRefType ColumnType;
29         typedef ColumnType::Subvector<3>::Type TranslationRefType;
30         TranslationRefType translation(lastCol);
31         translation.Assign(5.5f, -3.25f, 217.32f);
32         /**/
33
34         // Display the result
35         cout << "final_matrix:\n";
36         cout << matrix << endl;
37
38     } catch (std::exception Exception) {
39         cout << "Exception_received:" << Exception.what() << cout;
40     }
41 }

```

Here, we see that in fixed-size objects, the size of the submatrix has to be given in template parameters, though the location of its first element is still passed as a regular function argument. In C++ it is impossible to have templated typedef statements. Instead, we declare a templated inner class: `MatrixType::Submatrix<3,3>`, and that inner class has an internal typedef of its own type as `Type`. Similarly, for the slice of the first three elements in the last column, we use the type

```
MatrixType::ColumnRefType::Subvector<3>::Type
```

Note that the element type in this example is `float`, while the rotation matrix `rot30` is `double`. We can assign vectors and matrices of different element types to each other, but normally we do not allow other operations between them. Also note that we explicitly define literals of type `float` using the suffix `f`. This may reduce the number of compiler warnings. Also, we consider it safer to use explicit casting of the arguments whenever they are passed in a variable-length list (`va_arg`, or an ellipsis in the function declaration). This has to do with the mechanism used in C and C++ for handling variable-length argument list. So generally, if we have a long vector `v` of type `double`, the following code may generate an error:

```
v.Assign(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Instead, use explicit literal type:

```
v.Assign(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0);
```

For more information regarding the different type of references available, refer to the [cistVector User Guide](#).

5 Manipulating multidimensional arrays

Multidimensional arrays can be used to represent volumes (medical imaging), images with multiple layers (separating RGB channels) or any dataset of high dimension. `vctDynamicArray` can also be used for datasets of dimension 1 or 2 but for these, `vctDynamicVector` and `vctDynamicMatrix` might be a better choice.

Multidimensional arrays can only be found in the dynamic allocation flavor, i.e. there is no `vctFixedSizeNArray`. A multidimensional array type is defined by the type of elements stored as well as by the dimension, i.e. both of these are defined at compilation time and cannot be changed at runtime. As for the vectors and matrices, it is recommended to define an `nArray` type to use in your code:

```
typedef vctDynamicNArray<unsigned short, 3> InputVolumeType;
```

Since the dimension defines the number of indices required to randomly access an element as well as the number of sizes to resize an `nArray`, it is also very convenient to define both an index and size type:

```
typedef InputVolumeType::nsize_type SizeType;
typedef InputVolumeType::nindex_type IndexType;
```

The following code illustrates how to create, fill and operate on `nArrays`:

```
1 void ExampleNArray(void) {
2     // Define a working volume and index types
3     typedef vctDynamicNArray<unsigned short, 3> InputVolumeType;
4     typedef InputVolumeType::nsize_type SizeType;
5     typedef InputVolumeType::nindex_type IndexType;
6
7     // The dynamic vector library may throw exceptions,
8     // (derived from std::exception)
9     // so we place the operations in a try-catch block.
10    try {
11        // Create a volume object and allocate memory
12        SizeType volumeSize(128, 256, 256);
13        InputVolumeType inputVolume(volumeSize);
14        // alternative to set size and allocate
15        inputVolume.SetSize(volumeSize);
16
17        // Fill the memory with data
18        vctRandom(inputVolume, 0, 10);
19    }
```

```

20     // Random access (read) of elements
21     IndexType zyxIndex(15, 120, 240);
22     cout << inputVolume.Element(zyxIndex) << endl
23         << inputVolume.at(zyxIndex) << endl
24         << inputVolume[zyxIndex[0]] [zyxIndex[1]] [zyxIndex[2]]
25         << endl;
26
27     // Define a new volume type
28     typedef vctDynamicNArray<double, 3> NormalizedVolumeType;
29
30     // Type conversions from and existing volume, also defines the size
31     NormalizedVolumeType newVolume(inputVolume);
32     // alternative
33     newVolume.Assign(inputVolume);
34
35     // Find minimum and maximum
36     double minElement, maxElement;
37     newVolume.MinAndMaxElement(minElement, maxElement);
38     cout << "Min_ and _max_" << minElement << "_" << maxElement << endl;
39     // "shift and bias"
40     newVolume.Subtract(minElement); // Also available: operator -=
41     newVolume.Divide(maxElement - minElement); // or operator /=
42
43     // Operations with several operands
44     NormalizedVolumeType noise, corrected;
45     corrected.DifferenceOf(newVolume, noise);
46
47     // Slice overlay: array.Slice(dimension, sliceIndex)
48     newVolume.Slice(0, 0).SumOf(newVolume.Slice(0, 10),
49                                 newVolume.Slice(0, 20));
50
51     // Using a named object for slice overlay
52     typedef NormalizedVolumeType::ConstSliceRefType InputSliceType;
53     InputSliceType input1;
54     input1.SliceOf(newVolume, 1, 15);
55     // alternative
56     input1.SetRef(newVolume.Slice(1, 15));
57
58     // Layout manipulation: permutation of dimensions
59     vctDynamicNArrayRef<double, 3> permutedVolume;
60     permutedVolume.PermutationOf(newVolume, IndexType(1, 0, 2));
61 } // end of try block
62 // catch block
63 catch (std::exception Exception) {
64     cerr << "Exception_ occurred:_" << Exception.what() << endl;
65 }
66 }

```

Multidimensional arrays also provide ways to create slices and other references. It is possible to:

- Use only a sub-array while keeping the dimension, i.e. create a window along each dimension
- Reduce the visible dimension, i.e. only consider n-1 dimensions
- Use the data with a permutation of indices, i.e. permuting the dimensions

In this example, we reduced the dimension using the method `SliceOf`. The type of a slice is defined using the same type of elements and subtracting one from the dimension. To ease the programmer's life, one can use `NormalizedVolumeType::SliceRefType` and `NormalizedVolumeType::ConstSliceRefType`. A more subtle way to slice an `nArray` is to use the square brackets (operator `[]`). This is similar to a matrix operator `[]` as both operators return a reference container with a lower dimension.

Finally, the method `PermutationOf` allows to view the `nArray` from a different “angle” by implicitly re-ordering the dimensions (please note that the data itself is not moved or copied as for all `cisst` “Ref” objects). This can be compared to the method `TransposeOf` for a matrix.

6 Using the C++ Standard Template Library

The different containers of `cisstVector` have been written to be compatible with the STL. They define different iterators as well as the methods required to manipulate these iterators.

```

1 void ExampleSTL(void) {
2     typedef vctFixedSizeVector<double, 6> VectorType;
3
4     VectorType vector1;
5     int value = vector1.size();
6     const VectorType::iterator end = vector1.end();
7     VectorType::iterator iter;
8     // fill with numbers using iterators
9     for (iter = vector1.begin(); iter != end; ++iter) {
10         *iter = value--;
11     }
12     cout << vector1 << endl;
13     // sort using std::sort. cool isn't it?
14     std::sort(vector1.begin(), vector1.end());
15     cout << vector1 << endl;
16 }
```

In this example, we demonstrated the use of an STL generic algorithm (`std::sort`) on a `cisstVector` container.

7 Using `cisstCommon`

This example requires to include `cisstCommon.h`.

```

1 void ExampleCommon(void) {
2     // fill a vector with random numbers
3     vctFixedSizeVector<double, 8> vector1, vector2;
4     cmnRandomSequence & randomSequence = cmnRandomSequence::GetInstance();
5     randomSequence.ExtractRandomValueArray(-100.0, 100.0,
6                                             vector1.Pointer(), vector1.size());
7
8     // to fill a matrix or vector one can also use vctRandom
9     vctRandom(vector2, -100.0, 100.0);
10
11    // perform some operations such as divide by zero
12    vector2.SetAll(0.0);
13    vector2.Divide(vector2.Norm());
14    unsigned int index;
15    // look for Not A Number
16    for (index = 0; index < vector2.size(); index++) {
17        if (CMN_ISNAN(vector2[index])) {
18            cout << "vector[" << index << "] is NaN" << endl;
19        }
20    }
21
22    // create a rotation based on an a normalized axis
23    vctAxAnRot3 axisAngle(vct3(1.0, 0.0, 0.0), cmnPI / 2.0);
24    vctQuatRot3 rot1(axisAngle);
25
26    // modify this rotation with a new axis, not well normalized
27    vct3 axis(1.0005, 0.0, 0.0);
28    if (axis.Norm() > cmnTypeTraits<double>::Tolerance()) {
29        cout << "Axis is not normalized wrt default tolerance" << endl;
30    }
31    cmnTypeTraits<double>::SetTolerance(0.001);
32    // this method asserts that the axis is normalized
33    axisAngle.Axis() = axis;
34    axisAngle.Angle() = cmnPI / 2.0;
35    rot1.From(axisAngle);
36    cmnTypeTraits<double>::SetTolerance(cmnTypeTraits<double>::DefaultTolerance);
37 }

```

This example illustrates how to use the `cmnRandomSequence` to fill a vector or matrix with random numbers.

The macro `CMN_ISNAN` allows to check if a variable is still a number. It is defined in `cmnPortability.h`.

The default tolerance is used in many methods of `cmnVector` (e.g. `IsNormalized()` for any transformation) and it might be useful to increase it for a given application. This should be used with caution. The class `cmnTypeTraits` contains some useful information per type (double, float, char, int, etc) such as `HasNaN`, `MinNegativeValue`, etc.

8 Using cisstNumerical

In this example, we are showing how to choose the storage order. In C/C++, the usual convention is to use a row major data storage, this means that the elements of a matrix are stored in a single block, row by row. This is the default behavior of `cisstVector`. In some cases, it is necessary to store the data column by column. This is the case whenever one wants to interface with an external library using a column major representation. `cisstNumerical` includes a collections of wrappers around FORTRAN routines which might require a column major storage order (see [cisstNumerical Quickstart](#) and the [cisst cnetlib](#) pages).

```

1 void ExampleNumerical(void) {
2     // here we create the 3x3 matrix (column major!!)
3     const unsigned int rows = 3, cols = 3;
4     vctDynamicMatrix<double> m(rows, cols, VCT_COL_MAJOR);
5     // fill with random numbers
6     vctRandom(m, -10, 10);
7
8     // display the current matrix
9     cout << "Matrix:" << endl << m << endl;
10
11    // create and solve the problem
12    nmrSVDDynamicData svdData(m);
13    nmrSVD(m, svdData);
14    cout << "Singular Values:" << endl << svdData.S() << endl;
15 }

```

This example uses `vctRandom` to fill a 3 by 3 matrix. The matrix is declared using `VCT_COL_MAJOR`. For more information regarding the storage order, see the [cisstVector User Guide](#). To find the singular values of this matrix, we used the function `nmrSVD`. The class `nmrSVDDynamicData` allows to allocate some memory prior to the computation so that one can use the same memory for multiple problems of the same size. This can be particularly useful in a loop.

9 Writing your own functions

These examples are reserved to advanced programmers. They require a fairly good understanding of the C++ templates and the class hierarchy of `cisstVector` (refer to the [cisstVector User Guide](#)). The terms *Const* and *Ref* refer to the `cisstVector` classes (e.g. `vctDynamicConstVectorRef`), not the C++ keyword `const` and symbol `&`.

It is important to understand that the declaration of a new templated function or method is much more complex than the call to this function or method. When you will call the method, the compiler will infer (i.e. deduce) the template parameters based on the type of the objects used as function parameters. This allows to create a very generic method while preserving the ease of use.

9.1 Using dynamic containers

The first four functions show different possible signatures to use for a function parameter. The fifth signature is for a function that takes two matrices containing the same type of elements, but each matrix can be either a *Reference* or not, *Const* or not.

The last example shows how to use a `vctReturnDynamicVector` to avoid a copy of all the elements of a vector when it is returned (this approach is valid for a matrix as well).

```

1  // take any dynamic matrix as input parameter
2  // Const or not, Reference or not
3  template <class _matrixOwnerType, class _elementType>
4  void
5  FunctionDynamicA(vctDynamicConstMatrixBase<_matrixOwnerType,
6                  _elementType> & matrix)
7  {}
8
9  // take any non-const matrix as input parameter, Reference or not
10 template <class _matrixOwnerType, class _elementType>
11 void
12 FunctionDynamicB(vctDynamicMatrixBase<_matrixOwnerType,
13                 _elementType> & matrix)
14 {}
15
16 // only take a matrix as input parameter, can't use a Reference
17 template <class _elementType>
18 void
19 FunctionDynamicC(vctDynamicMatrix<_elementType> & matrix)
20 {}
21
22 // this shows how to restrict to a given type of elements (double)
23 template <class _matrixOwnerType>
24 void
25 FunctionDynamicD(vctDynamicConstMatrixBase<_matrixOwnerType, double> & matrix)
26 {}
27
28 // take any two dynamic matrices as input parameters
29 // Const or not, Reference or not, same type of elements
30 template <class _matrixOwnerType1, class _matrixOwnerType2, class _elementType>
31 void
32 FunctionDynamicE(vctDynamicConstMatrixBase<_matrixOwnerType1,
33                 _elementType> & matrix1,
34                 vctDynamicConstMatrixBase<_matrixOwnerType2,
35                 _elementType> & matrix2)
36 {}
37
38 // function with a return type
39 template<class _vectorOwnerType, class _elementType>
40 vctReturnDynamicVector<_elementType>
41 FunctionDynamicF(const vctDynamicConstVectorBase<_vectorOwnerType,
42                 _elementType> & inputVector) {

```

```

43     typedef _elementType value_type;
44     vctDynamicVector<value_type> resultStorage(inputVector.size());
45     // ..... do something to resultStorage
46     return vctReturnDynamicVector<value_type>(resultStorage);
47 }

```

9.2 Using fixed size containers

Fixed size containers are similar to the dynamic ones with a major exception, the size(s) and stride(s) must be specified at compilation time. This requirement adds a fair amount of template parameters.

```

1  // Take any fixed size vector as input parameter
2  // Const or not, Reference or not
3  template <unsigned int _size, int _stride, class _elementType,
4           class _dataPtrType>
5  void
6  FunctionFixedSizeA(vctFixedSizeConstVectorBase<_size, _stride, _elementType,
7                 _dataPtrType> & vector)
8  {}
9
10 // take any non-const vector as input parameter, Reference or not
11 template <unsigned int _size, int _stride, class _elementType,
12         class _dataPtrType>
13 void
14 FunctionFixedSizeB(vctFixedSizeVectorBase<_size, _stride, _elementType,
15                 _dataPtrType> & vector)
16 {}
17
18 // only take a vector as input parameter, can't use a Reference
19 template <class _elementType, unsigned int _size>
20 void
21 FunctionFixedSizeC(vctFixedSizeVector<_elementType, _size> & vector)
22 {}
23
24 // this shows how to restrict to a given type of elements (float)
25 template <unsigned int _size, int _stride, class _dataPtrType>
26 void
27 FunctionFixedSizeD(vctFixedSizeConstVectorBase<_size, _stride, float,
28                 _dataPtrType> & vector)
29 {}
30
31 // take any two fixed size vectors as input parameters
32 // Const or not, Reference or not, same size, same type of elements
33 template <unsigned int _size, class _elementType,
34         int _stride1, class _dataPtrType1,
35         int _stride2, class _dataPtrType2>
36 void
37 FunctionFixedSizeE(vctFixedSizeConstVectorBase<_size, _stride1, _elementType,
38                 _dataPtrType1> & vector1,

```

```
39         vctFixedSizeConstVectorBase<_size, _stride2, _elementType,  
40                                     _dataPtrType2> & vector2)  
41     {}  
42 }
```

Since the fixed size containers are designed for fairly small containers (up to approximately 10 elements) and they use stack memory, there is no specific return type (as opposed to `vctReturnDynamicVector` or `vctReturnDynamicMatrix`).