Container class templates extension for C++ vector class library

Agner Fog

Libre Planet

Free Software

Chapter 1

Introduction

A container class template is a piece of C++ code that is useful for allocating memory space for a list of objects. It is similar to an array but with additional functionality and security.

C++ programmers routinely use the C++ standard containers (previously known as the standard template library) for this purpose. Unfortunately, the standard C++ container templates can be quite inefficient. They are optimized for generality and flexibility, while efficiency has been sacrificed. Many of the standard C++ containers are implemented as linked lists that allocate memory in a lot of separate small pieces. This is inefficient because of a lot of heap management overhead, memory fragmentation, and poor caching. Many C++ programmers are routinely implementing a matrix as a nested container (vector of vectors) which is even more inefficient.

The container class templates provided here are intended to fill the need for more efficient containers with contiguous memory storage. Some of these containers are tailor-made to fit the classes defined in the vector classes library (VCL). Containers for other types of objects are also included.

Overview of container class templates:

Table 1.1: Container class templates

Template	**Description**	**Header file**
ContainerG<type>	Linear array of any type of objects, dynamic size	general_containers.h
ContainerV<vector_type, size>	Linear array of vectors, fixed size or dynamic size	vector_containers.h
MatrixV<vector_type, rows, columns>	Matrix. Rows are stored as VCL vectors	matrixv.h

Chapter 2

Description of container templates

2.1 ContainerG

Declaration
`template <typename T> class ContainerG;`

This container class template makes a linear array with dynamic size. `ContainerG` is independent of the vector class library and can be used for most kinds of objects.

The type `T` can be a simple type such as `int` or `float`, or a composite type such as a `struct`, `class`, or `union`. The container will likely not work if the type `T` has a non-default constructor, destructor, copy constructor, or move constructor. The type `T` cannot be another container, but it can be a `struct` containing a fixed-size array.

```
// Example:
#include <stdio.h>
#include "general_containers.h"

// Function for error reporting
void error_reporter() {
    fprintf(stderr, "\nError: index out of range\n");
}

int main () {
    // Declare a container for float elements
    ContainerG<float> my_array;

    // Register error_reporter function to report any errors
    my_array.set_error_handler(error_reporter);

    // Set the size of the array
    my_array.set_size(10);

    // Put data into a C-style array
    const int listsize = 8;
    float list[listsize];
    for (int i = 0; i < listsize; i++) list[i] = float(i);

    // Load 8 elements into my_array
    my_array.load(listsize, list);
    // Print contents of my_array
    for (int i = 0; i < my_array.size(); i++) {
        printf(" %.2f", my_array[i]);
    }

    // Increase the size of the array
    my_array.set_size(12);

    // Change last element (index goes from 0 to size()-1 )
    my_array[my_array.size()-1] = 88;

    // Print contents again
    printf("\n\n");
    for (int i = 0; i < my_array.size(); i++) {
        printf(" %.2f", my_array[i]);
    }
}

/*  Output:
 0.00 1.00 2.00 3.00 4.00 5.00 6.00 7.00 0.00 0.00

 0.00 1.00 2.00 3.00 4.00 5.00 6.00 7.00 0.00 0.00 0.00 88.00
*/
```

Member functions:

void set_size(int size)
Sets the size of the container, i.e. the number of elements it can contain.
The size can be changed at any time. Increasing the size will make the code allocate a new internal memory buffer if the current memory buffer is not big enough. All contents will be copied to the new memory buffer and the old buffer will be deleted.
The code may allocate a bigger memory block than requested in order to avoid frequent re-allocations if the size is increased in small amounts. The code does not re-allocate memory if the size is decreased to a non-zero value.
Calling `set_size` with a size of zero will delete the allocated memory and set everything to the initial condition. This may free memory for other purposes, but it is not needed because the container has a destructor that will free the allocated memory anyway.

int size()
Returns the current size of the container, i.e. the number of elements it can contain.

int allocated_size()
Gives the size of the internal buffer, which may be bigger than specified by the last call to `set_size`. `allocated_size()` is the maximum size that can be set without reallocation of the internal memory.

T & operator [] (int index)
The operator `[]` works like an array index. This makes it possible to read or write a single element in the array. The code checks that the index is within the range $0 \leq$ `index` $<$ `size()`.

void load(int n, T * p)
Loads `n` objects from an array `p`.

`n` is the size of the array `p` or the maximum number of elements to load. If `n` is bigger than the size of the `ContainerG` then it is reduced to the size of the container.

void store(int n, T * p)
Stores `n` objects to an array `p`.
`n` is the size of the array `p` or the maximum number of elements to store. If `n` is bigger than the size of the `ContainerG` then it is reduced to the size of the container.

set_error_handler(void (*err)(void))
Saves a pointer to an error handling function. This function will be called in case an index is out of range. The error handling function should issue an error message in a way that is appropriate for the actual user interface. The program will crash in case of an index out of range if no error handler is set.

T * get_buf()
Returns a pointer to the internal buffer. It is important to remember that any pointer or reference to elements in the container will be invalid after the size has been increased.

2.1.1 Recycling of memory

It may be more efficient to reuse a container for a new purpose during the course of the program than to delete each container when it is no longer needed and create a new one. This will improve memory caching.

The container may be resized for every new use. It can be useful to specify an estimated maximum size before first use of the container, and then reduce and increase the size as required during the course of the program.

The memory is zeroed at the first call to `set_size`, but it is not necessarily cleared when the container is later resized.

2.2 ContainerV

Declaration
`template <typename V, int n> class ContainerV`;

This container is designed for VCL vectors only. Vectors of all integer and floating point types are allowed, but not boolean types. Access is provided to each vector as well as individual vector elements.

```
// Example:
#include <stdio.h>
#include <vectorclass.h>
#include <vector_containers.h>

// Make container of four vectors of 8 float values each
ContainerV<Vec8f, 4> c;
// Array of floats
float list[32] = {0,1,2,3,4,5,6};
// Load array into container
c.load(32, list);
```

```
// Change one vector in container
c.set_vector(Vec8f(16,17,18,19,20,21,22,23), 2);
// Change one vector element in container
c.set_element(-99, 5);
// Loop through vectors:
for (int i = 0; i < c.n_vectors(); i++) {
    // Loop through elements of each vector:
    for (int j = 0; j < c.get_vector(0).size(); j++) {
        // Print value:
        printf(" %6.2f", c.get_vector(i)[j]);
    }
    // Next vector on new line:
    printf("\n");
}

/* Output:
   0.00   1.00   2.00   3.00   4.00 -99.00   6.00   0.00
   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
  16.00  17.00  18.00  19.00  20.00  21.00  22.00  23.00
   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
*/
```

Defined type etype
The template defines **etype** as the type of the vector elements. For example, if the container is based on vectors of type **Vec8f**, then **etype** is the type **float**.

Member functions:

int n_vectors()
Returns the number of vectors.

int n_elements()
Returns the number of vector elements.

int elementtype()
Returns the **elementtype()** of the underlying vector class + 0x1000.

vector_type get_vector(int index)
Returns one vector from the position indicated by index. (The first vector has index 0).

set_vector(vector_type x, int index)
Replaces one vector at the position indicated by index.

etype get_element(int index)
Returns a single vector element. The index runs consecutively through all vectors in the container, from 0 to (number of vectors) * (elements per vector) - 1.

set_element(etype x, int index)
Replaces a single vector element. The index runs consecutively through all vectors in the container, from 0 to (number of vectors) * (elements per vector) - 1.

load(int n, void * p)
Loads values from an array into the container. p points to an array of type `etype`. n is the array size or the maximum number of vector elements to load. If n is not a multiple of the vector size then the last vector will be partially filled. If n is bigger than the container size, then it is limited to the container size. If n is smaller than the container size, then the remaining full vectors are unchanged.

store(int n, void * p)
Stores values from the container into an array. p points to an array of type `etype`. n is the array size or the maximum number of vector elements to store. n does not have to be a multiple of the vector size. If n is bigger than the container size, then it is limited to the container size.

vector_type get_buf()
Returns a pointer to the internal buffer. Note that this pointer will be invalid if the size of the container is later increased (see below).

zero()
Sets all vectors and all vector elements in the container to zero. Does not change the size of the container.

set_error_handler(void (*err)(void))
Saves a pointer to an error handling function. This function will be called in case an index is out of range. The error handling function should issue an error message in a way that is appropriate for the actual user interface. The program will crash in case of an index out of range if no error handler is set.

```
// Example:
#include <stdio.h>
#include <vectorclass.h>
#include <vector_containers.h>

void error_reporter() {
    fprintf(stderr, "\nError: index out of range\n");
}

// Make container of four vectors of 8 float values each
ContainerV<Vec8f, 4> c;

// Set the error handler
c.set_error_handler(error_reporter);
```

2.2.1 Dynamic size

The `ContainerV` template has a dynamic size if, and only if, the initial size is 0.

```
// Example:
#include <stdio.h>
#include <vectorclass.h>
#include <vector_containers.h>

// Make dynamic container of a variable number of vectors
ContainerV<Vec8f, 0> c;
// Set the size
```

```
c.set_nvectors(6);
// Change one vector in container
c.set_vector(Vec8f(16,17,18,19,20,21,22,23), 2);
// Loop through vectors:
for (int i = 0; i < c.n_vectors(); i++) {
    // Loop through elements of each vector:
    for (int j = 0; j < c.get_vector(0).size(); j++) {
        // Print value:
        printf(" %6.2f", c.get_vector(i)[j]);
    }
    // Next vector on new line:
    printf("\n");
}

/* Output:
    0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
    0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
   16.00  17.00  18.00  19.00  20.00  21.00  22.00  23.00
    0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
    0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
    0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
*/
```

Additional member functions for dynamic size
The following member functions are only available for ContainerV if the initial size is 0:

set_nvectors(int size)
Changes the size of the container. On the first call to this function, the code allocates a memory block big enough to contain the specified number of vectors. If a later call to `set_nvectors` increases the size, then it will allocate a new bigger memory block if necessary, copy all data to the new memory block, and delete the old memory block. The code may allocate a bigger memory block than requested in order to avoid frequent re-allocations if the size is increased in small amounts. The code does not re-allocate memory if the size is decreased to a non-zero value. Calling `set_nvectors` with a size of zero will delete the allocated memory and set everything to the initial condition. This may free memory for other purposes, but it is not needed because the container has a destructor that frees the allocated memory anyway.

set_nelements(int n)
This function is similar to `set_nvectors`, but makes it possible to set a size that is not a multiple of the vector size. The size is rounded up to a multiple of the vector size in order to determine the amount of memory to allocate, while the last vector will be only partially used. `n_vectors()` will report the number of vectors used, including any partially used vector, while `n_elements()` will return the value set by `set_nelements(n)`.

int allocated_size()
This function returns the actual amount of memory allocated, including any unused memory. The unit is the vector size, similar to `set_nvectors`.

2.2.2 Recycling of memory

It may be more efficient to reuse a container for a new purpose during the course of the program than to delete each container when it is no longer needed and create a new one. This will improve memory caching.

The container may be resized for every new use. It can be useful to specify an estimated maximum size before first use of the container, and then reduce and increase the size as required during the course of the program.

The memory is zeroed at the first call to `set_nvectors`, but it is not necessarily cleared when the container is later resized. A container with fixed size contains random data initially.

2.3 MatrixV

MatrixV is a container for representing numerical data as a matrix. Each row in the matrix is stored as one or more VCL vectors. This is useful for representing numerical data as a matrix. MatrixV can also be used for storing a list of vectors where each vector is represented as a row, and the number of columns corresponds to the number of elements in each vector.

MatrixV is optimized for accessing the matrix as rows rather than columns.

Declaration
`template <typename V, int rows, int columns> class MatrixV;`

The vector type `V` can be any floating point or integer vector class. Boolean vectors are not supported. (It is more efficient to pack boolean vectors into integer bitfields).

If the number of columns is larger than the number of elements in the vector class `V` then the template will use multiple vectors for each row. If the number of columns is less than the number of elements in the vector class `V` then the template will use the smallest possible vector class that fits the number of columns. Any extra elements in `V` will be unused. The template will not store multiple rows in one vector, but start each row with a new vector. It is possible to specify `V` as the largest possible vector with the desired element type and leave it to the template to find the smallest vector of the same element type that fits the number of columns. The MatrixV template will not use vectors larger than `V`.

Defined type `row_vector_type`
The template defines `row_vector_type` as the vector class used for storing rows. This may be the specified vector class `V` or a smaller vector class with the same element type.

Defined type `etype`
The template defines `etype` as the type of the vector elements. For example, if the container is based on vectors of type `Vec8f`, then `etype` is the type `float`.

Member functions:

int nrows();
Returns the number of rows in the matrix.

int ncolumns();
Returns the number of columns in the matrix.

int vectors_per_row();

Returns the number of vectors of class `row_vector_type` that are used for each row. Any partially used vector is included.

int full_vectors_per_row();
Returns the number of fully-used vectors of class `row_vector_type` that are used for each row. Any partially used vector is not included.

int partial_vector_elements();
If the number of columns is not divisible by the number of elements in vector class `row_vector_type` then the last vector in each row will be partially used. This function returns the number of used elements in a partially-used vector. The function returns 0 if there are no partially used vectors.

set_error_handler(void (*err)(void));
This function is used for registering a function for reporting errors such as an index out of range. `err` should be a function that reports the error in a way that is appropriate for the actual user interface. The program will crash in case a row or column index is out of range if no error-handling function is registered.

void (*get_error_handler())(void);
Returns a pointer to the function set by `set_error_handler`.

row_vector_type get_row(int r, int i = 0);
Returns row number `r` as a vector of class `row_vector_type`.
Row numbers go from 0 to `nrows()-1`.
If each row contains more than one vector then call `get_row` multiple times with `i` going from 0 to `vectors_per_row() - 1`.

set_row(row_vector_type x, int r, int i = 0);
Sets row number `r` to a vector of class `row_vector_type`.
Row numbers go from 0 to `nrows()-1`.
If each row contains more than one vector then call `set_row` multiple times with `i` going from 0 to `vectors_per_row() - 1`.

etype get_element(int row, int column);
Returns a single element from the matrix.
The row number goes from 0 to `nrows()-1`.
The column number goes from 0 to `ncolumns()-1`.

set_element(etype x, int row, int column);
Changes a single element in the matrix.
The row number goes from 0 to `nrows()-1`.
The column number goes from 0 to `ncolumns()-1`.

load(void * p);
Fills the entire matrix with data from a C-style matrix or linear array pointed to by `p`. The matrix or array must contain (rows * columns) elements.
The elements are retrieved in row-major order in accordance with the C standard.

store(void * p);
A C-style matrix or array pointed to by `p` is filled with all data from the entire matrix. The number of elements stored at `*p` is (rows * columns).

The elements are stored in row-major order in accordance with the C standard.

zero()
Sets all elements in the matrix to zero.

2.3.1 Initializing a MatrixV

A `MatrixV` is not initialized when it is first constructed. The contents of unitialized matrix elements is unpredictable. The matrix can be initialized by the `load` member function or by multiple calls to `set_row`. It is less efficient to set all elements individually with `set_element`.

The internal vectors contain unused vector elements if the number of columns is not divisible by the vector size. It is recommended to call the `zero` member function first if the matrix elements are initialized with `set_element` only, in order to clear any unused vector elements. Otherwise, the unused elements may occur as random values in unused vector elements retrieved by `get_row` or by the pack functions described below.

2.3.2 Pack and unpack functions

Several functions are defined for packing multiple matrix rows into one big vector and for unpacking such a vector into multiple matrix rows. These functions cannot be used if the number of columns is too big for multiple rows to fit into a single large vector.

The pack and unpack functions are useful in cases where matrix elements are accessed in other patterns than rowwise and when permutations are needed, such as matrix transposition and matrix-by-matrix products. The pack and unpack functions support all floating point vector classes and integer vector classes with integer types of at least 16 bits.

Pack functions
```
template <typename M> auto pack2rows(M & matrix, int first_row);
template <typename M> auto pack3rows(M & matrix, int first_row);
template <typename M> auto pack4rows(M & matrix, int first_row);
template <typename M> auto pack5rows(M & matrix, int first_row);
```

These functions will pack n consecutive rows of a `MatrixV` matrix into a single vector, provided that a vector class with sufficient size exists.

For example, a matrix with 5 rows and 3 columns with elements of type `float` can be packed in the following ways. `pack2rows` will pack two consecutive rows into a vector of type `Vec8f` with the elements of two rows in the first 6 vector positions, while the last two vector positions are unused. `pack3rows` will pack three rows into a `Vec16f` with the first 9 elements used and the last 7 elements unused. `pack4rows` will use 12 elements, and `pack5rows` can pack the entire matrix into 15 elements of a `Vec16f` with the last element unused. `first_row` indicates the start row, where row numbers start at 0.

The pack functions are automatically finding a vector size that fits the number of data elements packed. You will get a compilation error if no sufficiently big vector class exists. It is not possible to pack the rows into multiple vectors with a single call to a pack function.

The error handling function set by `set_error_handler` for the matrix will be called in case any of the row indexes is out of range. For example, calling `pack3rows` with `first_row` set to 3 on a matrix with 5 rows will give an error because the last row is out of range. The program will crash if there is no error handling function and a row index is out of range.

Unpack functions

```
template <typename V, typename M> unpack2rows(V rr, M & matrix, int first_row));
template <typename V, typename M> unpack3rows(V rr, M & matrix, int first_row));
template <typename V, typename M> unpack4rows(V rr, M & matrix, int first_row));
template <typename V, typename M> unpack5rows(V rr, M & matrix, int first_row));
```

These functions are doing the opposite of the pack functions. The large vector `rr` is unpacked to fill n consecutive rows of the matrix `M`. The unpack functions can be used for initializing or modifying a matrix.

A row index out of range will be indicated by a call to the error handling function in the same way as for the pack functions.

2.3.3 Examples

The following examples illustrate how to use the `MatrixV` container, its member functions, and the pack and unpack functions.

```
#include <stdio.h>
#include <vectorclass.h>
#include <matrixv.h>

// Function for reporting an error message
void error_reporter() {
    fprintf(stderr, "\nError: index out of range\n");
}

// Function for printing a whole matrix
template <typename M>
void print_matrix(M & matrix) {
    // row loop
    for (int r = 0; r < matrix.nrows(); r++) {
        // column loop
        for (int c = 0; c < matrix.ncolumns(); c++) {
            // print one element
            printf(" %6.2f", matrix.get_element(r, c));
        }
        printf("\n"); // new line for next row
    }
}

int main() {

    // C-style matrix with 3 rows and 4 columns
    float Amatrix[3][4];

    // Put data into this matrix
    for (int r = 0; r < 3; r++) {
        for (int c = 0; c < 4; c++) {
            Amatrix[r][c] = float(r + 0.1 * c);
        }
    }
```

```
// Vector-based matrix with 3 rows and 4 columns
MatrixV<Vec8f, 3, 4> A;

// Set an error handler
A.set_error_handler(error_reporter);

// Load data from C-style matrix Amatrix into vector-based matrix A
A.load(Amatrix);

// Print matrix A
printf("\n   A:\n");
print_matrix(A);

// Pack matrix A into one big vector (A has 3 rows)
Vec16f Apack = pack3rows(A, 0);

// Transpose matrix A
const int d = V_DC;   // d means don't care
Vec16f Atransposed = permute16<
    0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11, d, d, d, d>(Apack);

// Vector-based matrix for A transposed. Has 4 rows and 3 columns
MatrixV<Vec8f, 4, 3> B;

// Set an error handler
B.set_error_handler(error_reporter);

// Unpack Atransposed vector into matrix B (B has 4 rows)
unpack4rows(Atransposed, B, 0);

// Row operations

// Subtract 2 * row2 from row1 in matrix B
auto row1 = B.get_row(1);  // row 1 as Vec4f
auto row2 = B.get_row(2);  // row 2 as Vec4f
auto new_row1 = row1 - 2.0f * row2; // calculate new row 1
B.set_row(new_row1, 1);    // insert new row 1

// Print matrix B
printf("\n   B:\n");
print_matrix(B);

// Matrix multiplication

// Pack matrix B into one big vector
Vec16f Bpack = pack4rows(B, 0);

// Calculate matrix product A * B
Vec16f AxBpack =

  permute16<0, 0, 0, 4, 4, 4, 8, 8, 8, d, d, d, d, d, d, d>(Apack)
* permute16<0, 1, 2, 0, 1, 2, 0, 1, 2, d, d, d, d, d, d, d>(Bpack)
```

```
    + permute16<1, 1, 1, 5, 5, 5, 9, 9, 9, d, d, d, d, d, d, d>(Apack)
    * permute16<3, 4, 5, 3, 4, 5, 3, 4, 5, d, d, d, d, d, d, d>(Bpack)

    + permute16<2, 2, 2, 6, 6, 6,10,10,10, d, d, d, d, d, d, d>(Apack)
    * permute16<6, 7, 8, 6, 7, 8, 6, 7, 8, d, d, d, d, d, d, d>(Bpack)

    + permute16<3, 3, 3, 7, 7, 7,11,11,11, d, d, d, d, d, d, d>(Apack)
    * permute16<9,10,11, 9,10,11, 9,10,11, d, d, d, d, d, d, d>(Bpack);

    // Product matrix has 3 rows and 3 columns
    MatrixV<Vec8f, 3, 3> AxB;

    // Set an error handler
    AxB.set_error_handler(error_reporter);

    // Unpack product vector into matrix AxB
    unpack3rows(AxBpack, AxB, 0);

    // Print pprocuct matrix AxB
    printf("\n   AxB:\n");
    print_matrix(AxB);
}

/* Output:
   A:
   0.00   0.10   0.20   0.30
   1.00   1.10   1.20   1.30
   2.00   2.10   2.20   2.30

   B:
   0.00   1.00   2.00
  -0.30  -1.30  -2.30
   0.20   1.20   2.20
   0.30   1.30   2.30

   AxB:
   0.10   0.50   0.90
   0.30   2.70   5.10
   0.50   4.90   9.30
*/
```