Introduction on C++ vector

VE281 - Data Structures and Algorithms, Xiaofeng Gao, TA: Qingmin Liu, Autumn 2019

1 std::vector

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with amortized constant time complexity.

Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than lists and forward_lists.

2 Container properties

2.1 Sequence

Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

2.2 Dynamic array

Allows direct access to any element in the sequence, even through pointer arithmetics, and provides relatively fast addition/removal of elements at the end of the sequence.

2.3 Allocator-aware

The container uses an allocator object to dynamically handle its storage needs.

3 Template parameters

3.1 T

Type of the elements.

Only if T is guaranteed to not throw while moving, implementations can optimize to move elements instead of copying them during reallocations.

Aliased as member type vector::value_type.

3.2 Alloc

Type of the allocator object used to define the storage allocation model. By default, the allocator class template is used, which defines the simplest memory allocation model and is value-independent.

Aliased as member type vector::allocator_type.

4 Member types (C++ 2011 Standard)

Table 1: Member types

member type	definition
value_type	The first template parameter (T)
allocator_type	The second template parameter (Alloc)
reference	value_type
const_reference	const value_type
pointer	allocator_traits<allocator_type>::pointer
const_pointer	allocator_traits<allocator_type>::const_pointer
iterator	a random access iterator to value_type
const_iterator	a random access iterator to const value_type
reverse_iterator	reverse_iterator<iterator>
const_reverse_iterator	reverse_iterator<const_iterator>

5 Member functions

- **Iterators:**
  - **begin** Return iterator to beginning (public member function)
  - **end** Return iterator to end (public member function)
  - **rbegin** Return reverse iterator to reverse beginning (public member function)
  - **rend** Return reverse iterator to reverse end (public member function)
  - **cbegin** Return const_iterator to beginning (public member function)
  - **cend** Return const_iterator to end (public member function)
- **Capacity:**
  - **size** Return size (public member function)
  - **max_size** Return maximum size (public member function)
  - **resize** Change size (public member function)
  - **capacity** Return size of allocated storage capacity (public member function)
  - **empty** Test whether vector is empty (public member function)
  - **reserve** Request a change in capacity (public member function)

  - **shrink_to_fit** Shrink to fit (public member function)

- **Element access:**
  - **operator[]** Access element (public member function)
  - **at** Access element (public member function)
  - **front** Access first element (public member function)
  - **back** Access last element (public member function)
  - **data** Access data (public member function)

- **Modifiers:**
  - **assign** Assign vector content (public member function)
  - **push_back** Add element at the end (public member function)
  - **pop_back** Delete last element (public member function)
  - **insert** Insert elements (public member function)
  - **erase** Erase elements (public member function)
  - **swap** Swap content (public member function)
  - **clear** Clear content (public member function)
  - **emplace** Construct and insert element (public member function)
  - **emplace_back** Construct and insert element at the end (public member function)

- **Allocator:**
  - **get_allocator** Get allocator (public member function)

6 Examples

```cpp
#include <vector>    // When using it, you need to include a header file "vector"

// Declaration and initialization
vector<int> x;       // define an empty vector
vector<int> x[5];    // define a vector with length 5 but without initial value
vector<int> x(5, 2);     // define a vector [2,2,2,2,2]
vector<int> x={0,1,2,3};     // define a vector [0,1,2,3]

// Common operation
x.push_back(4);      // insert "4" at the end of x
x.pop_back();        // delete the last element of x
x[i] = 100;          // change the ith element of x to 100
x.begin();           // return an iterator pointing to the first element of x
x.end();             // return an iterator pointing to the last element of x
x.insert(x.begin()+1, 100);      // insert an element with a value of 100 at the position of the first element of x
```

```
x.size();           // return the number of elements in x
reverse(x.begin(), x.end());     // reverse x
sort(x.begin(), x.end());        // sort the elements of x from smallest
    to largest

// 2D vector
// Get an array of 5 rows and 3 columns.
// The two-dimensional array implemented by vector can change the
   number of rows and columns with resize()
int i,j;
vector<vector<int>> array(5);
for (i = 0; i < array.size(); i++)
    array[i].resize(3);

for(i = 0; i < array.size(); i++)
{
    for (j = 0; j < array[0].size();j++)
    {
        array[i][j] = (i+1)*(j+1);
    }
}

// reverse 2D vector
// using iterator
void reverse_with_iterator(vector<vector<int>> vec)
{
    if (vec.empty())
    {
        cout << "Empty!" << endl;
        return;
    }

    vector<int >::iterator it;
    vector<vector<int >>::iterator iter;
    vector<int> vec_tmp;

    for(iter = vec.begin(); iter != vec.end(); iter++)
    {
        vec_tmp = *iter;
        for(it = vec_tmp.begin(); it != vec_tmp.end(); it++)
            cout << *it << endl;
    }
}

// using index
void reverse_with_index(vector<vector<int>> vec)
{
    if (vec.empty())
    {
        cout << "Empty!" << endl;
```

```
            return;
        }

        int i,j;
        for (i = 0; i < vec.size(); i++)
        {
            for(j = 0; j < vec[0].size(); j++)
                cout << vec[i][j] << endl;
        }
}
```