**VECTORS IN C++**

**(**<https://www.codesdope.com/cpp-stdvector/>)

Vectors are dynamic array equivalent in C++. They are of great use and notes on the following are given below.

In the last chapter, you saw std::array. In this one, you will be exploring **std::vector**.

Consider a case where you want to store the marks of the students of a class. But you don't know the number of students in the class and have only an approximate idea about the number. You can declare an array by specifying some array length.

Now, if the length of the array which you have specified during its declaration is smaller than the number of students, then the marks of all the students could not be stored. Also, if we declare its length much larger than the number of students, then unnecessarily extra memory will be allocated to the array which is not required.

In such cases, we don't know about the array length untill compile time (when computer compiles the code). This is where we need **std::vector**.

What is std::vector?

Unlike std::array whose length is specified at the time of declaration and remains constant till compile time, we can change the length of std::vector dynamically as the program executes according to our requirement.

**Vectors** are sequence containers which represent arrays which can change in size. Thus, we need not specify its length at the time of declaration and can change it later in the program.

Now let's look at how to use std::vector in place of arrays.

Declaration of std::vector

The declaration syntax of std::vector is the same as that of std::array, with the difference that we don't need to specify the array length along with the data type as shown below.

**std::vector<datatype> array\_name;**

For using std::vector, we need to include the **<vector>** header in our program.

Let's look at the declaration of a vector named **marks** of type int to store the marks of students.

**std::vector<int> marks;**

Now let's see how to initialize a vector.

Initialization of std::vector

The initialization of an std::vector is also the same as that of std::array. We initialize an std::vector by either of the following ways.

**std::vector<int> marks = {50, 45, 47, 65, 80};**

**std::vector<int> marks { {50, 45, 47, 65, 80} };**

We can also assign values to the vector after declaration as shown below.

**std::vector<int> marks;  
marks = {50, 45, 47, 65, 80};**

In the above declarations, we stored the marks of 5 students in a vector named **marks**. Since we did not declare the array length, so the length of 'marks' became equal to the number of values it was initialized with. Now, we may change the number of students i.e. either store the marks of more students or remove the marks of some students.

Length of std::vector

In std::vector also, the **size()** function returns the length (i.e. number of elements in the vector).

Let's see an example of std::vector.

**#include** *<iostream>*

**#include** *<vector>*

**int** main()

{

std::vector<**int**> marks = {50, 45, 47, 65, 80};

marks = {50, 47, 60};

std::cout << "length of array : " << marks.size() << std::endl;

**return** 0;

}

**Output**

In this example, we first initialized the vector **marks** with 5 elements, thus making its length 5. In the second statement, we reassigned **marks** with 3 values, thus making its length 3. After reassigning, the values of marks[0], marks[1] and marks[2] are 50, 47 and 60 respectively. Resizing is not possible in case of arrays.

Passing std::vector to function

Let's look at an example of passing a vector to a function.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**void** printVector(**const** std::vector<**int**> &n)

{

**for** (**int** j = 0; j < n.size(); j++ )

{

cout << "n[" << j << "] = " << n[j] << endl;

}

}

**int** main()

{

vector<**int**> n = {1,2,3,4,5};

printVector(n);

**return** 0;

}

**Output**

void printVector(const std::vector<int> &n) - const is used here to prevent the compiler from making a copy of the vector and this enhances the performance. The passed vector will be n in this function as &n is the parameter of the function 'printArray'.

Member Functions

std::vector also has a number of member functions which makes adding and deleting elements from the vector easier. Let's look at some of these.

at

**at** function is used to access the element at specified position (index). Let's see an example.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> marks = {50, 45, 47, 65, 80};

marks.at(2) = 74;

**for**(**int** i = 0; i < marks.size() ; i++)

{

cout << "marks[" << i << "] = " << marks.at(i) << endl;

}

**return** 0;

}

**Output**

In this example, **marks.at(2) = 74** assigned 74 to the third element (marks[2]) of the vector.

front

The **front** function returns the first element of a vector.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> marks = {50, 45, 47, 65, 80};

cout << marks.front() << endl;

**return** 0;

}

**Output**

back

**back()** function returns the last element of a vector.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> marks = {50, 45, 47, 65, 80};

cout << marks.back() << endl;

**return** 0;

}

**Output**

empty

It checks whether a vector contains any element or not. It returns **1** if the length of a vector is 0 and **0** if it contains some element.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> v1 = {5, 6};

vector<**int**> v2;

cout << v1.empty() << endl;

cout << v2.empty() << endl;

**return** 0;

}

**Output**

resize

It resizes a vector so that it contains the specified number of elements. Look at the following example.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> v1 = {5, 6};

v1.resize(5);

**for**(**int** i = 0; i < v1.size() ; i++)

{

cout << v1[i] << endl;

}

**return** 0;

}

**Output**

In this example, when we initialized the vector **v1**, it contained 2 elements. Thus its **length was 2** with **v1[0] = 5**and **v1[1] = 6**.  
**v1.resize(5)** resized the vector so that it contains 5 elements. Since we did not assign any value to the rest of the elements of the vector, they got assigned a value 0. So, **v1[2] = 0**, **v1[3] = 0** and **v1[4] = 0**.

max\_size

It returns the maximum number of elements that the vector can hold. This is not the amount of storage space currently allocated to the vector, but the maximum size the vector could reach due to limitations in system implementations.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> v1 = {1, 2, 3, 4, 5};

cout << v1.size() << endl;

cout << v1.max\_size() << endl;

**return** 0;

}

**Output**

assign

It assigns new contents to the vector and replaces its current contents. Let's see an example.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> v;

v.assign(7, 40); *// 7 elements each of value 40*

cout << v.size() << endl;

**return** 0;

}

**Output**

By writing **v.assign(7, 40)**, the vector **v** contains 7 elements each having a value of 40.

We can also assign some or all the elements of an array to a vector using this function. Let's see how.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

**int** arr[] = {1, 2, 3, 4, 7};

vector<**int**> v1;

vector<**int**> v2;

v1.assign(arr, arr+5);

v2.assign(arr, arr+2);

*//printing values of v1*

cout << "elements of v1" << endl;

**for**(**int** i = 0; i < v1.size(); i++)

{

cout << v1[i] << endl;

}

*//printing values of v2*

cout << "elements of v2" << endl;

**for**(**int** i = 0; i < v2.size(); i++)

{

cout << v2[i] << endl;

}

**return** 0;

}

**Output**

In this example, we created an array **arr** and declared two vectors **v1** and **v2**.  
**v1.assign(arr, arr+5) -**Here, **arr** is pointing to **arr[0]** and thus **arr+5** is pointing to **arr[5]**. This statement assigned the values of the elements of the array from arr[0] till arr[5](not including arr[5]) to the vector v1. So, now v1 contains 4 elements of values 1, 2, 3 and 4 respectively.  
**v2.assign(arr, arr+2) -**Similarly, this statement assigned the values of the elements of the array from arr[0] till arr[2](excluding it).

push\_back

This function adds a new element at the end of the vector (at the end of the last element of the vector), thus increasing the size of the vector by one.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> v = {4, 5, 6, 7, 8};

v.push\_back(47);

*//printing values of v*

cout << "elements of v" << endl;

**for**(**int** i = 0; i < v.size(); i++)

{

cout << v[i] << endl;

}

**return** 0;

}

**Output**

**v.push\_back(47)** added an element having value 47 at the end of the vector. Thus, the length of the vector became 6 and the value of the new element v[5] became 47.

pop\_back

This function removes the last element in the vector, thus reducing the size of the vector by one.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> v = {4, 5, 6, 7, 8};

v.pop\_back();

*//printing values of v*

cout << "elements of v" << endl;

**for**(**int** i = 0; i < v.size(); i++)

{

cout << v[i] << endl;

}

**return** 0;

}

**Output**

Here we removed the last element of the vector. thus making the length of the vector 4.

capacity

This function returns the storage space allocated for the vector. In other words, it returns the number of elements which can be stored in the storage space allocated for the vector.

Vector capacity is always greater than or equal to the vector size.

The vector capacity is slightly greater than the vector size because the storage space allocated to a vector is always greater than that occupied by the number of elements in it so that if we add another element in the vector, the extra storage space could adjust it.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> v1;

**for**(**int** i = 0; i < 50; i++)

{

v1.push\_back(1);

}

cout << "size : " << v1.size() << endl;

cout << "max\_size : " << v1.max\_size() << endl;

cout << "capacity : " << v1.capacity() << endl;

**return** 0;

}

**Output**

reserve

This function increases the capacity of the vector if the desired number of elements is greater than the capacity of the vector.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> v1;

**for**(**int** i = 0; i < 50; i++)

{

v1.push\_back(1);

}

v1.reserve(100);

cout << "capacity : " << v1.capacity() << endl;

**return** 0;

}

**Output**

Here we want to store 100 elements in the vector and thus increased the capacity of the vector using the reserve() function.

insert

It inserts a new element in a vector before the element at the specified position. Let's see an example to understand this.

**#include** *<iostream>*

**#include** *<vector>*

**#include** *<iterator>*

**using** **namespace** std;

**void** printVector(**const** std::vector<**int**> &n)

{

cout << "Vector is :" << endl;

**for** (**auto** i: n)

{

std::cout << ' ' << i;

}

std::cout << '\n';

}

**int** main()

{

vector<**int**> v = {111,222};

printVector(v);

**auto** it = v.begin();

v.insert(it,000);

printVector(v);

it = v.begin();

v.insert(it,3,333);

printVector(v);

vector<**int**> v1 = {555,555};

it = v.begin();

v.insert(it+4, v1.begin(), v1.end());

printVector(v);

**return** 0;

}

**Output**

**v.insert(it,000);** - We inserted 0 at the beginning of the vector.

**v.insert(it,3,333);** - We inserted 333 thrice at the beginning of the vector.

**v.insert(it+4, v1.begin(), v1.end());** - We inserted the entire vector v1 (v1.begin(), v1.end()) at the 4th position of the vector v.

erase

**erase** function removes either a single element or a range of elements from a vector.

To remove a single element, we need to pass the iterator pointing to the element in the vector which is to be removed. To remove a range of elements, we need to pass iterators specifying the range which we want to delete. Let's see an example.

**#include** *<iostream>*

**#include** *<vector>*

**#include** *<iterator>*

**using** **namespace** std;

**int** main()

{

vector<**int**> v1 = {4, 5, 6, 7, 8};

vector<**int**> v2 = {1, 2, 3, 4, 5};

v1.erase(v1.begin()+4); *// removing a single element at position 4*

v2.erase(v2.begin()+1, v2.begin()+3); *// removing range of elements from position 1 till 2*

*//printing the values of v1*

cout << "Values of v1" << endl;

**for**(**int** i = 0; i < v1.size(); i++)

{

cout << v1[i] << endl;

}

*//printing the values of v2*

cout << "Values of v2" << endl;

**for**(**int** i = 0; i < v2.size(); i++)

{

cout << v2[i] << endl;

}

**return** 0;

}

**Output**

clear

**clear** removes all elements of a vector.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> v1 = {4, 5, 6, 7, 8};

v1.clear();

**for** (**auto** i: v1)

{

std::cout << i << endl;

}

**return** 0;

}

**Output**

swap

This function swaps the contents i.e. it exchanges the value of one vector with that of another.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<**int**> v1 = {1, 2, 3};

vector<**int**> v2 = {4, 5, 6};

v1.swap(v2);

std::cout << "Vector v1" << endl;

**for** (**auto** i: v1)

{

std::cout << i << endl;

}

std::cout << "Vector v2" << endl;

**for** (**auto** i: v2)

{

std::cout << i << endl;

}

**return** 0;

}

**Output**

Multidimensional std::vector

We can also make multidimensional std::vectors. This is similar to std::array. Let's look at the syntax to make a std::vector.

**std::vector<std::vector<int>> v { {{1,2,3}, {4,5,6}} };**

This is similar to std::array. The inner vector (std::vector<int>) is a vector of integers and the outer vector is a vector of such inner vectors (std::vector<vector array>).

Let's look at an example to make a multidimensional vector and access all its elements.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**int** main()

{

vector<vector<**int**>> v {{{1,2,3},{4,5,6},{7,8,9}}};

**for**(**int** i=0; i<3; i++)

{

**for**(**int** j=0; j<3; j++)

{

cout << v[i][j] << "\t";

}

cout << endl;

}

**return** 0;

}

**Output**

Passing a multidimensional std::vector to a function

A multidimensional std::vector is also passed to a function in a similar way a 1D vector is passed. Let's look at an example.

**#include** *<iostream>*

**#include** *<vector>*

**using** **namespace** std;

**void** display(**const** std::vector<std::vector<**int**>> &v)

{

**for**(**int** i=0; i<v.size(); i++)

{

**for**(**int** j = 0; j<v[i].size(); j++)

{

cout << v[i][j] << "\t";

}

cout << endl;

}

}

**int** main()

{

vector<vector<**int**>> v {{{1,2,3},{4,5,6},{7,8,9}}};

display(v);

**return** 0;

}

**MORE Vector Functions –**

A list of functions is given here <https://www.geeksforgeeks.org/vector-in-cpp-stl/>