**Vector**

**What:**

Its a template based container and behaves like dynamic array. It can expand memory dynamically at run time and store elements in contiguous memory location just like array. We can store the <type> of element in vector by specfying the <type> in template argument.

1. Ordered collection.

All the elements remain in the same order in which they are inserted.

2. Provides random access.

Indexing is very fast in std::vector using operator [], just like array.

3. Performance.

It performs better when insertion and deletion is performed in the end only.

4. Contains copy.

It always stores copy of the object not the same reference. So, if you are adding objects of user defined classes the you should define copy constructor and assignment opeartor in you class.

**Why:**

std::vector give same kind of fast performance in indexing and iterations as arrays. But it dont have a Fixed Size limitaton like Arrays. You dont need to provide the fixed size for std::vector in advance. Just start inserting elements in std::vector and it will automatically expand its size.

**How to initialize:**

Creating a vector object without any initialization will create an empty vector with no elements.

Ex:

std::vector<int> v;

To initialize with default values:

1. Vector provides a constructor that accepts size as argument and initializes the vector with that many object of default value.

Ex:

std::vector<int> v(5);

//So when we print this vector it contains 5 elements of value 0.

2. Vector provides a constructor that accepts size and value as argument and initializes the vector with that value.

Ex:

std::vector<int> v(5, 10);

//So here vector contains 5 elements of 10 value by default.

To initialize a vector with array:

vector contains a overloaded constructor that can be used to assign a array to vector.

Ex:

int arr[] = {1, 2, 3, 4, 5};

std::vector<int> v (arr, arr + sizeof(arr)/sizeof(int));

for(int x: v)

std::cout << v << “ ” << std::endl;

To initialize a vector with list:

Overloaded constructor is used to initialize a vector with range.

Ex:

#include<iostream>

#include<list>

#include<vector>

int main()

{

std::list<std::string> liststr;

liststr.push\_back("first");

liststr.push\_back("sec");

liststr.push\_back("third");

liststr.push\_back("fourth");

std::vector<std::string> v(liststr.begin(), liststr.end());

for(std::string str: v)

std::cout << str << std::endl;

return 0;

}

To initialize a vector with another vector:

Vector provides a constructor that receives other vector as argument and initializes current vector with the copy of all elements of provided vector.

**Syntax:**

std::vector<type> vector2;

vector2.push\_back(data);

vector2.push\_back(data);

vector2.push\_back(data);

std::vector<type> vector1(vector2);

**How does vector work internally:**

std::vector allocates a memory on heap and store all its elements in contiguous memory location.  
  
  
But what if memory it allocated initially is completely filled?  
For example, let’s create a vector of ints i.e. std::vector<int> . Now suppose it’s initial capacity is to store 10 elements, but in our application we want to store 15 elements in it. Then what will happen when we will insert 11th element?

When std::vector’s internal memory completely finishes then it increases the size of its memory. To do that it performs following steps,

1.) It will allocate a bigger chunk of memory on heap i.e. almost double the size of previously allocated.  
2.) Then it copies all the elements from old memory location to new one. Yes it copies them, so in case our elements are user defined objects then their copy constructor will be called. Which makes this step quite heavy in terms of speed.  
3.) Then after successful copying it deletes the old memory.

You can check the current capacity of vector i.e. how much elements it can store in current allocated memory using capacity() member function.  
To check the count of currently stored elements in std::vector one can use size() member function.

**Importance of constructor for user defined objects:**

For User Defined classes if Copy Constructor and Assignment Operator are public then only one can insert it’s object in std::vector.

This is because of two reasons,

* All STL contains always stores the copy of inserted objects not the actual one. So, whenever we insert any element or object in container then it’s copy constructor is called to create a copy and then this copy is inserted in the container.
* While insertion in std::vector it might be possible that storage relocation takes place internally due to insufficient space. In such cases assignment operator will be called on objects inside the container to copy them from one location to another.

**Note:**

If any of the type constructor is defined and default constructor is not defined, then it will give compiler error when vector is trying to use default constructor.

To overcome the above problem we use ***reserve()*** member function to initialize the size of the vector. Reverve() increases the capacity not the size of the vector.

**How to use vector effeciently:**

**We can use vector efficiently by taking care of following points,**

1.) Vector will be more efficient if elements are inserted or removed from the back-end only.

As, vector internally stores all the elements in consecutive memory location. Therefore, if an element is added in middle, then vector right shifts all the right side elements of that location by 1. Also, if elements were user defined objects then copy constructors for all those elements are called.

Similarly If element is erased from the middle, then vector left shifts all the right side elements of that location by 1. Also, if elements were user defined objects then copy constructors for all those elements are called.

But if elements are inserted or deleted from the back-end only then this costly shifting will not happen.

2.) Set the storage of vector initially using reserve() member function.

As vector is a kind of container in which user can store unlimited elements. Internally it allocates storage to store the elements but during insertion if new memory requirement surpasses the current capacity then it allocates a bigger chunk of storage and copies all the existing elements there. It’s a huge burden for application because if elements in vector are user defined objects then in every new movement to new storage location copy constructor of elements will be called.

We can avoid this if in our application by reserving the vector capacity initially by calling reserve() function. This reserve() function requests the vector capacity to be at least enough to contains n elements. It only increases the vector’s capacity, size remains same.

3.) Instead of adding single element in multiple calls, large set of elements is added in single call

Adding single element can cause,

* Shifting of some elements in vector
* Allocation of new memory and movement of all elements on new location

If we add a single element multiple times than all the above things can happen multiple times. Whereas, if we insert elements in together i.e. in a set than this shifting and copying can happen only once. vector can check if it has the capacity to store n elements or not or it needs to shift some elements by n location.

**Iterator invalidation:**

An Iterator becomes invalidate when the container it points to changes its shape internally i.e. move elements from one location to another and the initial iterator still points to old invalid location.

Iterator invalidation in vector happens when,

* An element is inserted to vector at any location
* An element is deleted from vector.

Example:

Suppose an iterator ‘it’ points to a location x in the vector. Now suppose some deletion happens on that vector, due to which it move its elements from one location to another. Now if initial iterator ‘it’ still points to old location then it becomes invalidated. For example, in the below code we are deleting an element from vector using erase function. This erase function invalidates the current pointer. So if after calling the erase() function , if one uses the same invalidated iterator then it can result in undefined behavior.

To overcome iterator invalidation always pin point the iterator location after erasing the data from it.

**Cause of iterator invalidation**

// Erase and element with value 5.

auto it = std::find(vecArr.begin(), vecArr.end(), 5);

if(it != vecArr.end())

vecArr.erase(it);

**Overcoming iterator invalidation**

// Erase and element with value 5.

auto it = std::find(vecArr.begin(), vecArr.end(), 5);

if(it != vecArr.end())

it = vecArr.erase(it);

So its better to reinitialize the vector to the beginning to overcome iterator ivalidation.

**Removing the recurrence of elements in vector:**

Use Erase-Remove idiom.

std::remove transforms the given range into a range with all the elements that compare not equal to given element shifted to the start of the container. So, actually dont remove the matched elements.  
It just shifted the non moatched to starting and gives an iterator to new valid end.  
It just requires O(n) complexity.

Output of remove algo will be,

1 2 4 1 7 8 9 ? ? ?

Now use vector’s erase function to delete elements from new end to old end of vector. It requires O(1) time.

**Syntax:**

vec.erase(std::remove(vec.begin(), vec.end(), elem), vec.end());

**Hidden cost of vector**

Avoid passing vector by value to avoid wastage of memory, usage of **reference** need to be done to reduce the memory wastage.