# Vector Introduction

Vector is a template based container that behaves just like a Dynamic Array.

It can expands its memory at run time and always store elements in contiguous memory location just like Array.

We can store any type of element in vector by specifying the type as template argument.

Lets see an example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | #include <iostream>  #include <vector>  int main()  {     // This is a vector of int      std::vector<int> vecOfInts;        // While Adding it automatically adjusts it's size      for(int i = 0; i < 10; i++)          vecOfInts.push\_back(i);        std::vector<int>::iterator it = vecOfInts.begin();      while(it != vecOfInts.end())      {          std::cout<<\*it<<" , ";          it++;      }        std::cout<<std::endl;        for(int i = 0; i < vecOfInts.size(); i++)          std::cout<<vecOfInts[i]<<" , ";        std::cout<<std::endl;        return 0;  } |

**Important Points about std::vector :**

*1.) Ordered Collection:*  
In std::vector all elements will remain in same order in which they are inserted.

*2.) Provides random access:*  
Indexing is very fast in std::vector using opeartor [], just like arrays.

*3.) Performance:*  
It Performs better if insertion and deletion is in end only and gives worst performance if insertion/deletion is at middle or at starting of vector.

*4.) Contains Copy:*  
It always stores copy of the object not the same reference. So, if you are adding objects of user defined classes then you should define copy constructor and assignment operator in you class.

**Why Should I use std::vector ?**

std::vector give same kind of fast performance in indexing and iterations as arrays. But it doesn’t have a Fixed Size limitaton like Arrays. You don’t 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.

# 5 Different ways to initialize a vector in c++

In this article we will discuss different ways to initialize a vector in C++.

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

|  |  |
| --- | --- |
| 1 | std::vector<int> vecOfInts; |

But we generally want to initialize a vector with huge values and calling push\_back() that many times to add element in vector is not an efficient and intelligent solution. So, let’s discuss how to initialize a vector in different ways,

## Initializing a vector with default value of elements

Vector provides a constructor that accepts the size as an argument and initialize the vector with that many objects of default value i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // Initialize vector with 5 integers  // Default value of all 5 ints will be 0.  std::vector<int> vecOfInts(5);    for(int x : vecOfInts)  std::cout<<x<<std::endl; |

**Output:**

|  |  |
| --- | --- |
| 1  2  3  4  5 | 0  0  0  0  0 |

## Initialize a vector by filling similar copy of an element

Many times we want to initialize a vector with an element of particular value instead of default value. For that vector provides an overloaded constructor i.e.

|  |  |
| --- | --- |
| 1 | vector (size\_type n, const value\_type& val, const allocator\_type& alloc = allocator\_type()); |

It accepts the size of vector and an element as an argument. Then it initializes the vector with n elements of value val.

Lets see an example that how to initialize a vector of std::string to 5 string objects with value “Hi”.

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Initialize vector to 5 string objects with value "Hi"  std::vector<std::string> vecOfStr(5, "Hi");    for(std::string str : vecOfStr)  std::cout<<str<<std::endl; |

**Output:**

|  |  |
| --- | --- |
| 1  2  3  4  5 | Hi  Hi  Hi  Hi  Hi |

## Initialize a vector with an array

In above two examples we saw how to initialize a vector with same kind of value either default value or a particular value. But what if we want to initialize a vector with an array of elements. For that vector provides an over loaded constructor i.e.

|  |  |
| --- | --- |
| 1 | vector (InputIterator first, InputIterator last, const allocator\_type& alloc = allocator\_type()); |

It accepts a range as an argument i.e. two iterators and initializes the vector with elements in range (first, last] i.e. from first till last -1.

We will use the same overloaded constructor to initialize a vector of string from an array of strings i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // Create an array of string objects  std::string arr[] = {"first", "sec", "third", "fourth"};    // Initialize vector with a string array  std::vector<std::string> vecOfStr(arr, arr + sizeof(arr)/sizeof(std::string));    for(std::string str : vecOfStr)  std::cout<<str<<std::endl; |

**Output:**

|  |  |
| --- | --- |
| 1  2  3  4 | first  sec  third  fourth |

## Initialize a vector with std::list

We will use the same overloaded constructor of std::vector to initialize a vector with range i.e.

|  |  |
| --- | --- |
| 1 | vector (InputIterator first, InputIterator last, const allocator\_type& alloc = allocator\_type()); |

This time range will be of std::list’s iterator i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // Create an std::list of 5 string objects  std::list<std::string> listOfStr;  listOfStr.push\_back("first");  listOfStr.push\_back("sec");  listOfStr.push\_back("third");  listOfStr.push\_back("fouth");    // Initialize a vector with std::list  std::vector<std::string> vecOfStr(listOfStr.begin(), listOfStr.end());    for(std::string str : vecOfStr)  std::cout<<str<<std::endl; |

**Output:**

|  |  |
| --- | --- |
| 1  2  3  4 | first  sec  third  fourth |

## Initializing a vector with another vector

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

|  |  |
| --- | --- |
| 1 | vector (const vector& x); |

Lets how to initialize a vector of string with another vector of same type i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | std::vector<std::string> vecOfStr;  vecOfStr.push\_back("first");  vecOfStr.push\_back("sec");  vecOfStr.push\_back("third");    // Initialize a vector with other string object  std::vector<std::string> vecOfStr3(vecOfStr); |

Complete code with all 5 different ways to initialize a vector is as follows,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | #include <iostream>  #include <vector>  #include <iterator>  #include <list>  #include <string>    void example1() {  // Initialize vector with 5 integers  // Default value of all 5 ints will be 0.  std::vector<int> vecOfInts(5);    for (int x : vecOfInts)  std::cout << x << std::endl;    }    void example2() {  // Initialize vector to 5 string objects with value "Hi"  std::vector<std::string> vecOfStr(5, "Hi");    for (std::string str : vecOfStr)  std::cout << str << std::endl;    }    void example3() {  // Create an array of string objects  std::string arr[] = { "first", "sec", "third", "fourth" };    // Initialize vector with a string array  std::vector<std::string> vecOfStr(arr,  arr + sizeof(arr) / sizeof(std::string));    for (std::string str : vecOfStr)  std::cout << str << std::endl;    }    void example4() {  // Create an std::list of 5 string objects  std::list<std::string> listOfStr;  listOfStr.push\_back("first");  listOfStr.push\_back("sec");  listOfStr.push\_back("third");  listOfStr.push\_back("fouth");    // Initialize a vector with std::list  std::vector<std::string> vecOfStr(listOfStr.begin(), listOfStr.end());    for (std::string str : vecOfStr)  std::cout << str << std::endl;    // Initialize a vector with other string object  std::vector<std::string> vecOfStr3(vecOfStr);    for (std::string str : vecOfStr3)  std::cout << str << std::endl;    }    int main() {    example1();  example2();  example3();  example4();  return 0;  } |

# How does std::vector works 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.

Check out this example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | #include <iostream>  #include <vector>    struct Sample  {      Sample(){}      Sample(const Sample & obj)      {          std::cout<<"Sample copy constructor"<<std::endl;      }  };  int main()  {      std::vector<Sample> vecOfInts;        std::cout<<"Capacity :: "<<vecOfInts.capacity()<<std::endl;      std::cout<<"Size :: "<<vecOfInts.size()<<std::endl;      int capcity = vecOfInts.capacity();      for(int i = 0; i < capcity + 1; i++)          vecOfInts.push\_back(Sample());        std::cout<<"Capacity :: "<<vecOfInts.capacity()<<std::endl;          std::cout<<"Size :: "<<vecOfInts.size()<<std::endl;        for(int i = 0; i < capcity + 1; i++)              vecOfInts.push\_back(Sample());        std::cout<<"Capacity :: "<<vecOfInts.capacity()<<std::endl;      std::cout<<"Size :: "<<vecOfInts.size()<<std::endl;        return 0;  } |

# Importance of Constructors while using User Defined Objects with std::vector

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 container 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.

Check out below example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | class Sample  {  // Copy Constructor should not be private if we are inserting it's  // objects in std::vector  Sample(const Sample & obj)  {  std::cout<<"Sample :: Copy Constructor"<<std::endl;  }  public:  Sample()  {  std::cout<<"Sample :: Default Constructor"<<std::endl;  }    Sample & operator=(const Sample & obj)  {  std::cout<<"Sample :: Assignment Operator"<<std::endl;  }  };    int main()  {  std::vector<Sample> vecOfSamples;  Sample obj;  vecOfSamples.push\_back(obj);    return 0;  } |

A part from this, if you are initializing the std::vector with default size by passing size as parameter i.e.

|  |  |
| --- | --- |
| 1 | std::vector<NewSample> vecOfNewSample(2); |

In such scenario 2 objects of NewSample will be created using default constructor. But if default constructor is not available then it will give compile error.

Check out the following example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class NewSample  {  public:  // Remove these comments to make it compilable  NewSample()  {}  NewSample(const NewSample & obj)  {}  NewSample & operator=(const NewSample & obj)  {}  };    int main()  {    std::vector<NewSample> vecOfNewSample(2);  // It will create 2 Objects using default constructor and insert them in vector.    std::cout<<"vecOfNewSample Size = "<<vecOfNewSample.size()<<std::endl;  std::cout<<"vecOfNewSample Capacity = "<<vecOfNewSample.capacity()<<std::endl;    std::vector<NewSample> vecOfNewSample\_2;  vecOfNewSample\_2.reserve(2);  // It will make the capacity of vector to contain 2 objects.  // Although size will remain 2.    std::cout<<"vecOfNewSample\_2 Size = "<<vecOfNewSample\_2.size()<<std::endl;  std::cout<<"vecOfNewSample\_2 Capacity = "<<vecOfNewSample\_2.capacity()<<std::endl;    return 0;  } |

Here we have not Defined Default constructor but defined copy constructor, therefore default constructor will get hidden for this class. Hence, it will give compile error when vector will try to create two objects using default constructor.

This problem will not happen if we use reserver() member function to initialize the size of vector because reserver() just increases the capacity of vector not the size.

Self Example:

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**class MyClass**

**{**

**private:**

**int \*p;**

**public:**

**static int num\_objects;**

**MyClass()**

**{**

**num\_objects++;**

**p = new int;**

**\*p = 10;**

**cout<<"Def Ctor"<<endl;**

**}**

**/\*MyClass(const MyClass &obj)**

**{**

**p = new int;**

**\*(this->p) = \*(obj.p) \* num\_objects;**

**cout<<"Copy Ctor"<<endl;**

**}\*/**

**int getVal(){**

**return \*p;**

**}**

**~MyClass(){**

**cout<<"Dtor"<<endl;**

**delete p;**

**}**

**};**

**int MyClass::num\_objects = 0;**

**int main()**

**{**

**std::vector<MyClass> myVec;**

**std::cout << "capacity of my\_vector = " << myVec.capacity() << std::endl;**

**std::cout << "Number of MyClass objects = " << MyClass::num\_objects << std::endl;**

**myVec.reserve(1);**

**MyClass a1;**

**std::cout << "Push a1"<< std::endl;**

**myVec.push\_back(a1);**

**//myVec.push\_back(MyClass());**

**std::cout << "Value of p = " << myVec[0].getVal()<< std::endl;**

**std::cout << "Vector size = " << myVec.size() << std::endl;**

**std::cout << "capacity of my\_vector = " << myVec.capacity() << std::endl;**

**std::cout << "Number of MyClass objects = " << MyClass::num\_objects << std::endl;**

**myVec.reserve(myVec.capacity()+1);**

**std::cout << "Value of p = " << myVec[0].getVal()<< std::endl;**

**std::cout << "capacity of my\_vector = " << myVec.capacity() << std::endl;**

**std::cout << "Number of MyClass objects = " << MyClass::num\_objects << std::endl;**

**myVec.reserve(myVec.capacity()+1);**

**std::cout << "Value of p = " << myVec[0].getVal()<< std::endl;**

**std::cout << "capacity of my\_vector = " << myVec.capacity() << std::endl;**

**std::cout << "Number of MyClass objects = " << MyClass::num\_objects << std::endl;**

**return 0;**

**}**

Output:

**capacity of my\_vector = 0**

**Number of MyClass objects = 0**

**Def Ctor**

**Push a1**

**Value of p = 10**

**capacity of size = 1**

**capacity of my\_vector = 1**

**Number of MyClass objects = 1**

**Dtor**

**Value of p = 0**

**capacity of my\_vector = 2**

**Number of MyClass objects = 1**

**Dtor**

**free(): double free detected in tcache 2**

**Command terminated by signal 6**

# How to use vector efficiently in C++?

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 contain 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.

# Vector and Iterator Invalidation example

**What is 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.

**Iterator Invalidation Example on Element Deletion in vector:**

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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | #include <iostream>  #include <vector>  #include <iterator>  #include <algorithm>    int main()  {      std::vector<int> vecArr;      for(int i = 1; i <= 10; i++)          vecArr.push\_back(i);        for(auto it = vecArr.begin(); it != vecArr.end(); it++)          std::cout<<(\*it)<<"  ";        std::cout<<std::endl;        // Erase and element with value 5.      auto it = std::find(vecArr.begin(), vecArr.end(), 5);      if(it != vecArr.end())          vecArr.erase(it);        // Now iterator 'it' is invalidated because it still points to      // old location, which has been deleted. So, if you will try to      // do the use the same iterator then it can show undefined      // behavior.        for(; it != vecArr.end(); it++)   // Unpredicted Behavior          std::cout<<(\*it)<<"  ";          // Unpredicted Behavior        return 0;  } |

Now, how to fix this ?

**Solution:**

After calling the erase function update the value of iterator **‘it’**i.e.

|  |  |
| --- | --- |
| 1  2  3  4 | // Erase the element with value 5.  auto it = std::find(vecArr.begin(), vecArr.end(), 5);  if(it != vecArr.end())     it = vecArr.erase(it); |

As, erase() function returns an iterator pointing to the new location of the element that followed the last element erased by the same function. Also, if the element deleted was the last element of the container then it returns the end of the container.

**Iterator Invalidation Example on Element Insertion in vector:**

When a new element is inserted in vector then it internally shifts its elements and hence the old iterators become invalidated.

Reasons for element shift are as follows,

* If element is inserted in between then it shift all the right elements by 1.
* If the new size of vector is more than its current capacity, then it relocates a bigger chunk of memory and copies all the elements there.

Therefore, when a new element is inserted in vector then its old iterator can become invalidated. Using this old invalidated iterators can result in undefined behavior i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #include <iostream>  #include <vector>  #include <iterator>  #include <algorithm>    int main()  {      std::vector<int> vecArr;      for(int i = 1; i <= 1; i++)          vecArr.push\_back(i);        auto it = vecArr.begin();      for(; it != vecArr.end(); it++)          std::cout<<(\*it)<<"  ";        std::cout<<std::endl;        it = vecArr.begin();        // Insert an element in position 2,      //vecArr.insert ( it + 2 , 1 , 200 );  vecArr.push\_back(200);        // Now old iterator it has become invalidated      // SO, using it as it is can result in undefined behavior        for(; it != vecArr.end(); it++)   // Undefined Behavior          std::cout<<(\*it)<<"  ";          // Undefined Behavior        return 0;  } |

Now, how to fix this ?

**Solution:**

After calling the insert function update the value of iterator **‘it’**i.e. by re-assigning it.

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Insert an element in position 2,  //vecArr.insert ( it + 2, 1 , 200 );  vecArr.push\_back(200);    // Reinitialize the invalidated iterator to the begining.  it = vecArr.begin(); |

# Remove all occurrences of an element from vector in O(n) complexity

Suppose we have a vector of integers and we want to delete all occurences of a number from it i.e.

Let’s say vector contain following numbers 1,2,5,4,5,1,5,7,8,9.  
Now we want to delete all the occurences of 5 from it, so that vector contents should become – 1 2 4 1 7 8 9 .  
Also the order of elements should be maintained.

There are two ways to do this,

**First Method: A Non efficient way**

**Algo:**  
1.) Iterate through all elements in vector and check for each elements if it matches with required number.  
2.) If it matches then erase that element and go forward.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | std::vector<int>::iterator it = vec.begin();  while(it != vec.end())  {  if(\*it == elem)  {  it = vec.erase(it);  }  else  it++;  } |

Although it seems to be quite efficient but its not. Because erase function deletes the elements and shifts all the elements in right by 1.  
So, it complexity will be O(n^2).

Let’s do it in a more efficient way,

**Second Method : An Efficient Way**

*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-matched 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.

|  |  |
| --- | --- |
| 1 | vec.erase(std::remove(vec.begin(), vec.end(), elem), vec.end()); |

Hence this new algo does the job in O(n) complexity.

Checkout the complete code,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | #include <iostream>  #include <vector>  #include <algorithm>  #include <iterator>  using namespace std;  void removeAllMatchingElements\_nonEfficient(std::vector<int> & vec, int elem)  {  std::vector<int>::iterator it = vec.begin();  while(it != vec.end())  {  if(\*it == elem)  {  it = vec.erase(it);  }  else  it++;  }  }    void displayVector(std::vector<int> & vec)  {  std::vector<int>::iterator it = vec.begin();  while(it != vec.end())  std::cout<<(\*it++)<<" ";  std::cout<<std::endl;  }    void removeAllMatchingElements\_Efficient(std::vector<int> & vec, int elem)  {  //vec.erase(std::remove(vec.begin(), vec.end(), elem), vec.end());  std::vector<int>::iterator it;  //Transforms the range [first,last) into a range with all the elements that compare equal to val removed, and returns an iterator to the new end of that range.  it=std::remove(vec.begin(), vec.end(), elem);  displayVector(vec);  cout<<\*it<<endl;  vec.erase(it, vec.end());  }  int main()  {  int arr[10] = {1,2,5,4,5,1,5,7,8,9};    std::vector<int> vec(arr , arr + 10);  removeAllMatchingElements\_nonEfficient(vec, 5);  displayVector(vec);    std::vector<int> vec2(arr , arr + 10);  removeAllMatchingElements\_Efficient(vec2, 5);  displayVector(vec2);    return 0;  } |

# How to fill a vector with random numbers in C++

Complete example is as follows,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | #include <iostream>  #include <stdlib.h>  #include <vector>  #include <algorithm>    struct RandomGenerator {  int maxValue;  RandomGenerator(int max) :  maxValue(max) {  }    int operator()() {  return rand() % maxValue;  }  };  int main() {  // Initialize a vector with 10 ints of value 0  std::vector<int> vecOfRandomNums(10);    // Generate 10 random numbers by lambda func and fill it in vector  std::generate(vecOfRandomNums.begin(), vecOfRandomNums.end(), []() {  return rand() % 100;  });    std::cout << "Random Number Generated by Lambda Function" << std::endl;  for (int val : vecOfRandomNums)  std::cout << val << std::endl;    // Generate 10 random numbers by a Functor and fill it in vector  std::generate(vecOfRandomNums.begin(), vecOfRandomNums.end(),  RandomGenerator(500));    std::cout << "Random Number Generated by Functor" << std::endl;  for (int val : vecOfRandomNums)  std::cout << val << std::endl;    return 0;  } |

To compile the above code use following command

g++ –std=c++11 sample.cpp

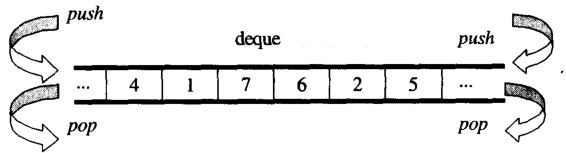
**Output:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | Random Number Generated by Lambda Function  83  86  77  15  93  35  86  92  49  21  Random Number Generated by Functor  362  27  190  59  263  426  40  426  172  236 |

<https://thispointer.com/be-careful-of-hidden-cost-while-using-stdvector-with-user-defined-objects/>

**deque = double ended queue**

I was looking at STL containers and trying to figure what they really are (i.e. the data structure used), and the deque stopped me: I thought at first that it was a double linked list, which would allow insertion and deletion from both ends in constant time, but I am troubled by [the promise made](http://en.cppreference.com/w/cpp/container/deque/operator_at) by the operator [] to be done in constant time. In a linked list, arbitrary access should be O(n), right?



The datas in deque are stored by chuncks of fixed size vector, which are

pointered by a map(which is also a chunk of vector, but its size may change)

Imagine it as a vector of vectors. Only they aren't standard std::vectors.

The outer vector contains pointers to the inner vectors. When its capacity is changed via reallocation, rather than allocating all of the empty space to the end as std::vector does, it splits the empty space to equal parts at the beginning and the end of the vector. This allows push\_front and push\_back on this vector to both occur in amortized O(1) time.

The inner vector behavior needs to change depending on whether it's at the front or the back of the deque. At the back it can behave as a standard std::vector where it grows at the end, and push\_back occurs in O(1) time. At the front it needs to do the opposite, growing at the beginning with each push\_front. In practice this is easily achieved by adding a pointer to the front element and the direction of growth along with the size. With this simple modification push\_front can also be O(1) time.

Access to any element requires offsetting and dividing to the proper outer vector index which occurs in O(1), and indexing into the inner vector which is also O(1). This assumes that the inner vectors are all fixed size, except for the ones at the beginning or the end of the deque.

# Deque vs vector: What to choose ?

* Vector provides good performance while insertion and deletion at end only and bad performance for insertion and deletion at middle.
* Deque provides same kind of performance as vector for insertion & deletion at end and middle. Apart from that deque provides good performance for insertion and deletion at front also.
* As Vector stores elements contiguously, where as deque internally contains a list of memory chunks which store elements contiguously. Due this basic architectural difference between vector and deque following things happen,
* Performance of addition and deletion at end for vector is better than deque.
* No Iterator invalidation happens in deque for insertion and deletion at front and end because like vectors, deque doesn’t have to shift elements from one memory to another in case current allocated memory is not sufficient to store the newly added element.

* Iterator invalidation happens in deque just like vector, if insertion or deletion takes place in the middle.
* Just like vector, deque also supports random access operations i.e. operator [] and at() function. Although performance of random access in deque will be little slower than vector.

**When to choose deque over vector:**

One should choose deque over vector if he wants to either add or delete from both the ends like implementing a Queue.

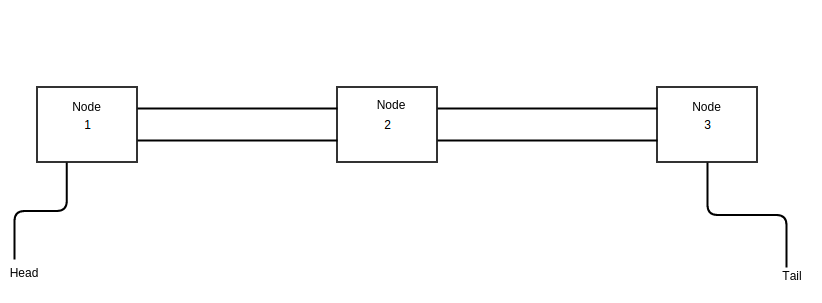
**When to choose vector over deque:**

One should choose vector if insertion or deletions are required mostly in end like implementing a Stack.

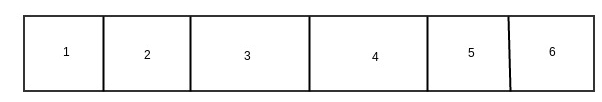
# Difference between Vector and List in C++

Both vector and list are sequential containers of C++ Standard Template Library. But there are many differences between them because of their internal implementation i.e.

**List** stores elements at non contiguous memory location i.e. it internally uses a doubly linked list i.e.

[](https://thispointer.com/wp-content/uploads/2016/05/linkedlist.png)

Whereas, **vector** stores elements at contiguous memory locations like an array i.e.

[](https://thispointer.com/wp-content/uploads/2016/06/array.jpg)

# std::vector vs std::list

### 1.) Insertion and Deletion

Insertion and Deletion in List is very efficient as compared to vector because to insert an element in list at start, end or middle, internally just a couple of pointers are swapped.

Whereas, in vector insertion and deletion at start or middle will make all elements to shift by one. Also, if there is insufficient contiguous memory in vector at the time of insertion, then a new contiguous memory will be allocated and all elements will be copied there.

So, insertion and deletion in list is much efficient than vector in c++.

### 2.) Random Access:

As List is internally implemented as doubly linked list, therefore no random access is possible in List. It means, to access 15th element in list we need to iterate through first 14 elements in list one by one.

Whereas, vector stores elements at contiguous memory locations like an array. Therefore, in vector random access is possible i.e. we can directly access the 15th element in vector using operator [] i.e.

|  |  |
| --- | --- |
| 1  2 | std::vector<int> vec(20);  vec[15] = 10; |

So, we can not use std::list with some of the STL algorithm that needs the random access operators like std::sort.

### 3.) Iterator Invalidation

Deleting or Inserting an element in List does not invalidate any iterator because during insertion and deletion no element is moved from its position only a couple pointers are changed.

Whereas, in vector insertion and deletion can invalidate the iterators. For more details about vector Iterator Invalidation, check this article i.e.

[std::vector and Iterator Invalidation](https://thispointer.com/stdvector-and-iterator-invalidation/)

### 4.) Special Member functions

As std::list do not provide random access, there many STL algorithms that uses Random Access Iterators can not be used with List. Hence std::list provides some extra functions for Sorting, Splicing, Removing elements and identifying unique elements.

Vector provides the random access and hence can be used with STL algorithms that uses Random Access Iterators.

# How to copy all Values from a Map to a Vector in C++

In this article we will discuss how to fetch all values from a map and put them in vector.

Suppose we have a map of words and their frequency count i.e.

|  |  |
| --- | --- |
| 1  2  3  4 | // Map of string & int  // To store the frequency count of words  std::map<std::string, int> wordMap = { { "is", 6 }, { "the", 5 },  { "hat", 9 }, { "at", 6 }, { "of", 2 }, { "hello", 1 } }; |

Now let’s fetch all values from this map in a vector of int i.e.

|  |  |
| --- | --- |
| 1 | 6 , 9 , 1 , 6 , 2 , 5 |

Copy all values from a map to vector using Range Based for Loop

|  |  |
| --- | --- |
| 1  2  3 | /\*\*\* Copy all value fields from map to a vector using Range Based For Loop \*\*\*/  for(auto elem : wordMap)  vecOfValues.push\_back(elem.second); |

Copy all values from a map to vector using for\_each() & Lambda function

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // Vector of int to store values  std::vector<int> vecOfValues;  vecOfValues.reserve(wordMap.size());    /\*\*\* Copy all value fields from map to a vector using Lambda function \*\*\*/  std::for\_each(wordMap.begin(), wordMap.end(),  [&](std::pair<const std::string, int>  & element){  vecOfValues.push\_back(element.second);  }); |

Iterate over each entry in map and call lambda function on each entry. Which will put second field from given pair to the vector.

Copy all values from a map to vector using transform() and lambda function

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Vector of int to store values  std::vector<int> vecOfValues2;  vecOfValues2.reserve(wordMap.size());    /\*\*\* Copy all value fields from map to a vector using transform() & Lambda function \*\*\*/  std::transform (wordMap.begin(), wordMap.end(),back\_inserter(vecOfValues2), [] (std::pair<std::string, int> const & pair)  {  return pair.second;  }); |

std::transform will iterate over each entry in map and call given lambda function on it. Then pass the result returned by it to the vector’s back\_inserter.

## Copy all values from a map to vector using transform() & function pointer

We can also call the std::transform() with a function pointer i.e. let’s create a template function that returns second value from a given pair i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | /\*  \* Template function that returns 2nd Value from a pair  \*/  template <typename K, typename V>  V getSecond(std::pair<K, V> const & pair)  {  return pair.second;  } |

Now use this template function instead of lambda function in std::transform() to copy all values to vector i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // Vector of int to store values  std::vector<int> vecOfValues3;  vecOfValues3.reserve(wordMap.size());    /\*\*\* Copy all values from a map to vector using transform() & function pointer \*\*\*/  std::transform (wordMap.begin(), wordMap.end(),back\_inserter(vecOfValues3), &getSecond<std::string, int>); |

Complete example is as follows,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | #include <iostream>  #include <map>  #include <string>  #include <vector>  #include <algorithm>  #include <iterator>  #include <utility>  #include <functional>    /\*  \* Template function that returns 2nd Value from a pair  \*/  template <typename K, typename V>  V getSecond(std::pair<K, V> const & pair)  {  return pair.second;  }    int main() {    // Map of string & int  // To store the frequency count of words  std::map<std::string, int> wordMap = { { "is", 6 }, { "the", 5 },  { "hat", 9 }, { "at", 6 }, { "of", 2 }, { "hello", 1 } };    /\*  \* Copy all Values from a Map to a Vector using different techniques  \*/    // Vector of int to store values  std::vector<int> vecOfValues;  vecOfValues.reserve(wordMap.size());    /\*\*\* Copy all value fields from map to a vector using Range Based For Loop \*\*\*/  for(auto elem : wordMap)  vecOfValues.push\_back(elem.second);    // Print contents of vector  std::copy(vecOfValues.begin(), vecOfValues.end(), std::ostream\_iterator<int>(std::cout, " , ") );  std::cout<<std::endl;    // Clear the vector  vecOfValues.clear();    /\*\*\* Copy all value fields from map to a vector using Lambda function \*\*\*/  std::for\_each(wordMap.begin(), wordMap.end(),  [&](std::pair<const std::string, int>  & element){  vecOfValues.push\_back(element.second);  });    // Print contents of vector  std::copy(vecOfValues.begin(), vecOfValues.end(), std::ostream\_iterator<int>(std::cout, " , ") );  std::cout<<std::endl;      // Vector of int to store values  std::vector<int> vecOfValues2;      vecOfValues2.reserve(wordMap.size());    /\*\*\* Copy all value fields from map to a vector using transform() & Lambda function \*\*\*/  std::transform (wordMap.begin(), wordMap.end(),back\_inserter(vecOfValues2), [] (std::pair<std::string, int> const & pair)  {  return pair.second;  });    // Print contents of vector  std::copy(vecOfValues2.begin(), vecOfValues2.end(), std::ostream\_iterator<int>(std::cout, " , ") );  std::cout<<std::endl;    // Vector of int to store values  std::vector<int> vecOfValues3;  vecOfValues3.reserve(wordMap.size());    /\*\*\* Copy all values from a map to vector using transform() & function pointer \*\*\*/  std::transform (wordMap.begin(), wordMap.end(),back\_inserter(vecOfValues3), &getSecond<std::string, int>);      // Print contents of vector  std::copy(vecOfValues3.begin(), vecOfValues3.end(), std::ostream\_iterator<int>(std::cout, " , ") );  std::cout<<std::endl;    return 0;  } |

**Output:**

|  |  |
| --- | --- |
| 1  2  3 | 6 , 9 , 1 , 6 , 2 , 5 ,  6 , 9 , 1 , 6 , 2 , 5 ,  6 , 9 , 1 , 6 , 2 , 5 , |

# How to get element by index in vector | at() vs operator []

In vector elements are indexed from 0 to size() – 1. To access any element in vector by index vector provides two member functions i.e.

* at()
* operator[]

Let’s discuss them in detail,

## Access an element in vector using operator []

std::vector provides [] operator i.e.

|  |  |
| --- | --- |
| 1 | element\_reference operator[] (size\_type n); |

It returns the reference of element in vector at index n.

Suppose we have a vector of int i.e.

|  |  |
| --- | --- |
| 1 | std::vector<int> vecOfNums{ 1, 4, 5, 22, 33, 2, 11, 89, 49 }; |

Let’s access element at index 3 using operator [] i.e.

|  |  |
| --- | --- |
| 1  2 | // Access element at index 3  int & element = vecOfNums[3]; |

### Access and update element in vector using []

As, operator [] returns a reference to the element in vector, so we can change the content of vector too using operator [] i.e.

|  |  |
| --- | --- |
| 1  2 | // Access and change the value of element at index 3  vecOfNums[3] = 10; |

It will update the value of element at index 3. New contents of vector will be,

|  |  |
| --- | --- |
| 1 | 1 , 4 , 5 , 10 , 33 , 2 , 11 , 89 , 49 |

We can also keep the returned reference in a reference variable and use later to modify the vector i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Get the reference of element at index 3  int & elemRef = vecOfNums[3];    // Modifying the vector using reference to element at index 3  elemRef = 22; |

New contents of vector will be,

|  |  |
| --- | --- |
| 1 | 1 , 4 , 5 , 22 , 33 , 2 , 11 , 89 , 49 |

### Accessing out of index element through operator []

While accessing any element through operator [] we need to make sure that given index is in range i.e. less than the size of vector, otherwise it will result in undefined behaviour and can also crash application.

Therefore we should always check the size before accessing element using operator [] i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // Accessing out of range element using [] results in undefined behaviour  int index = 100;  if (index < vecOfNums.size())  {  element = vecOfNums[index];  }  else  std::cout << "index out of bound" << std::endl; |

## Access an element in vector using vector::at()

std::vector provides an another member function at() i.e.

|  |  |
| --- | --- |
| 1 | reference at(size\_type n); |

It returns the reference of element at index n in vector. If index n is out of range i.e. greater then size of vector then it will throw out\_of\_range exception.

Let’s access element at index 3 using at() i.e.

|  |  |
| --- | --- |
| 1  2 | // Access element at index 3 using at()  int & numRef = vecOfNums.at(3); |

As at() returns a reference, so we can modify the value of element too i.e.

|  |  |
| --- | --- |
| 1  2 | // Modifying the element in vector using reference  numRef = 96; |

### Accessing out of range element using at()

vector::at() will throw out\_of\_range exception in case we try to access the out of range element i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // Accessing out of range element using at() will throw out\_of\_range exception  try  {  int index = 100;  element = vecOfNums.at(index);  }  catch (const std::out\_of\_range & ex)  {  std::cout << "out\_of\_range Exception Caught :: " << ex.what() << std::endl;  } |

**Output:**

|  |  |
| --- | --- |
| 1 | out\_of\_range Exception Caught :: vector::\_M\_range\_check: \_\_n (which is 100) >= this->size() (which is 9) |

## vector::operator[] vs vector::at()

Both operator[] & at() provides random access to elements in vector in O(1) Complexity. But in case of out of range access operator[] causes undefined behaviour, whereas at() returns proper out\_of\_range exception. So, at() is more safe to use as compared to operator[].

**Complete example is as follows,**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83 | #include <iostream>  #include <vector>  #include <algorithm>  #include <functional>  #include <string>    template <typename T>  void print(T & vecOfElements, std::string delimeter = " , ")  {  for(auto elem : vecOfElements)  std::cout<<elem<<delimeter;  std::cout << std::endl;  }  int main()  {  std::vector<int> vecOfNums{ 1, 4, 5, 22, 33, 2, 11, 89, 49 };    /\*  Accessing element by index in vector using operator[]  \*/    // Access element at index 3  int & element = vecOfNums[3];    std::cout << "Original vecOfNums = ";  print(vecOfNums);    std::cout << "Element at index 3 is : " << element << std::endl;    // Access and change the value of element at index 3  vecOfNums[3] = 10;    std::cout << "Modified vecOfNums = ";  print(vecOfNums);    // Get the reference of element at index 3  int & elemRef = vecOfNums[3];    // Modifying the vector using reference to element at index 3  elemRef = 22;    std::cout << "Modified vecOfNums = ";  print(vecOfNums);    // Accessing out of range element using [] results in undefined behaviour  int index = 100;  if (index < vecOfNums.size())  {  element = vecOfNums[index];  }  else  std::cout << "index out of bound" << std::endl;    /\*  Accessing element by index in vector using at()  \*/    // Access element at index 3 using at()  int & numRef = vecOfNums.at(3);    std::cout << "Element at index 3 is : " << numRef << std::endl;    // Modifying the element in vector using reference  numRef = 96;    std::cout << "Modified vecOfNums = ";  print(vecOfNums);    // Accessing out of range element using at() will throw out\_of\_range exception  try  {  int index = 100;  element = vecOfNums.at(index);  }  catch (const std::out\_of\_range & ex)  {  std::cout << "out\_of\_range Exception Caught :: " << ex.what() << std::endl;  }      return 0;    } |

**Output:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Original vecOfNums = 1 , 4 , 5 , 22 , 33 , 2 , 11 , 89 , 49 ,  Element at index 3 is : 22  Modified vecOfNums = 1 , 4 , 5 , 10 , 33 , 2 , 11 , 89 , 49 ,  Modified vecOfNums = 1 , 4 , 5 , 22 , 33 , 2 , 11 , 89 , 49 ,  index out of bound  Element at index 3 is : 22  Modified vecOfNums = 1 , 4 , 5 , 96 , 33 , 2 , 11 , 89 , 49 ,  out\_of\_range Exception Caught :: vector::\_M\_range\_check: \_\_n (which is 100) >= this->size() (which is 9) |

# How to insert element in vector at specific position

Vector provides different overloaded version of member function insert() , to insert one or more elements in between existing elements.  
Let’s discuss them in detail,

## Inserting a single element at specific position in vector

We are going to use first overloaded version of Vector’s insert() function i.e.

|  |  |
| --- | --- |
| 1 | iterator insert (const\_iterator pos, const value\_type& val); |

It Inserts a copy of give element **“val”**, before the iterator position “**pos**” and also returns the iterator pointing to new inserted element.

Let’s understand by example,

Suppose we have a vector of int i.e.

|  |  |
| --- | --- |
| 1 | std::vector<int> vecOfNums { 1, 4, 5, 22, 33, 2, 11, 89, 49 }; |

Now we want to insert an element at index position 4th (In vector position index start from 0),

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Create Iterator pointing to 4th Position  auto itPos = vecOfNums.begin() + 4;    // Insert element with value 9 at 4th Position in vector  auto newIt = vecOfNums.insert(itPos, 9); |

Vector’s contents will be now,

|  |  |
| --- | --- |
| 1 | 1 , 4 , 5 , 22 , 9 , 33 , 2 , 11 , 89 , 49 |

Inserting an element in vector will increase the vector size by 1.  
As in vector all elements are stored at continuous memory locations, so inserting an element in between will cause all the elements in right to shift or complete reallocation of all elements.

## Inserting multiple elements or a range at specific position in vector

Some times we encounter a situation where we want to insert multiple elements in vector at specific position. These multiple elements can from another vector , array or any other container.

For this, vector provides an overloaded version of insert() function to insert multiple elements i.e.

|  |  |
| --- | --- |
| 1 | iterator insert (const\_iterator pos, InputIterator first, InputIterator last); |

It inserts the elements in range from **[first, end)** before iterator position**pos** and returns the iterator pointing to position first newly added element.

Let’s understand by an example,

Suppose we have 2 vectors of strings i.e.

|  |  |
| --- | --- |
| 1  2 | std::vector<std::string> vec1 { "at" , "hello", "hi", "there", "where", "now", "is", "that" };  std::vector<std::string> vec2 { "one" , "two", "two" }; |

Now insert all the elements in vec2 at position 3 in vec1 i.e.

|  |  |
| --- | --- |
| 1  2 | // Insert all the elements in vec2 at 3rd position in vec1  auto iter = vec1.insert(vec1.begin() + 3, vec2.begin(), vec2.end()); |

Contents of vec1 will be now,

|  |  |
| --- | --- |
| 1 | at , hello , hi , one , two , two , there , where , now , is , that , |

## Inserting multiple elements using Initializer list

Another overloaded version of vector’s insert() function is as follows,

|  |  |
| --- | --- |
| 1 | iterator insert (const\_iterator position, initializer\_list<value\_type> list); |

It copies all the elements in given initializer list before given iterator position pos and also returns the iterator of first of the newly added elements.

Suppose we have vector of int i.e.

|  |  |
| --- | --- |
| 1 | std::vector<int> vecOfInts { 1, 4, 5, 22, 33, 2, 11, 89, 49 }; |

Let’s add all elements in initialisation list to the existing vector i.e.

|  |  |
| --- | --- |
| 1  2 | // Insert all elements from initialization\_list to vector at 3rd position  auto iter2 = vecOfInts.insert(vecOfInts.begin() + 3, {34,55,66,77}); |

Contents of vecOfInts will be now,

|  |  |
| --- | --- |
| 1 | 1 , 4 , 5 , 34 , 55 , 66 , 77 , 22 , 33 , 2 , 11 , 89 , 49 |

## vector.insert() and Iterator invalidation

Inserting elements in vector will cause existing elements to shift places or sometimes complete reallocation, which will invalidates all the existing iterators.

Complete example is as follows,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63 | #include <iterator>  #include <iostream>  #include <vector>  #include <algorithm>  #include <string>    template <typename T>  void print(T & vecOfElements, std::string delimeter = " , ")  {  for(auto elem : vecOfElements)  std::cout<<elem<<delimeter;  std::cout << std::endl;  }  int main()  {    std::vector<int> vecOfNums { 1, 4, 5, 22, 33, 2, 11, 89, 49 };    /\*  \* Inserting an element at specific position in vector  \*/    // Create Iterator pointing to 4th Position  auto itPos = vecOfNums.begin() + 4;    // Insert element with value 9 at 4th Position in vector  auto newIt = vecOfNums.insert(itPos, 9);    std::cout << "Element added in vector is : " << \*newIt << std::endl;  std::cout << "Modified vecOfNums = ";  print(vecOfNums);    /\*  \* Inserting multiple elements / range at specific position in vector  \*/    std::vector<std::string> vec1 { "at" , "hello", "hi", "there", "where", "now", "is", "that" };  std::vector<std::string> vec2 { "one" , "two", "two" };    // Insert all the elements in vec2 at 3rd position in vec1  auto iter = vec1.insert(vec1.begin() + 3, vec2.begin(), vec2.end());    std::cout << "First of the newly added elements : " << \*iter << std::endl;  std::cout << "Modified vec1 = ";  print(vec1);    /\*  \* Inserting all elements in initialization\_list in another vector  \* at specific position.  \*/    std::vector<int> vecOfInts { 1, 4, 5, 22, 33, 2, 11, 89, 49 };    // Insert all elements from initialization\_list to vector at 3rd position  auto iter2 = vecOfInts.insert(vecOfInts.begin() + 3, {34,55,66,77});    std::cout << "First of the newly added elements : " << \*iter2 << std::endl;  std::cout << "Modified vecOfInts = ";  print(vecOfInts);    return 0;    } |

**Output:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Element added in vector is : 9  Modified vecOfNums = 1 , 4 , 5 , 22 , 9 , 33 , 2 , 11 , 89 , 49 ,  First of the newly added elements : one  Modified vec1 = at , hello , hi , one , two , two , there , where , now , is , that ,  First of the newly added elements : 34  Modified vecOfInts = 1 , 4 , 5 , 34 , 55 , 66 , 77 , 22 , 33 , 2 , 11 , 89 , 49 , |

# How to find duplicates in a vector?

Suppose we have a vector of strings i.e.

|  |  |
| --- | --- |
| 1  2  3 | // Vector of strings  std::vector<std::string> vecOfStings{ "at" , "hello", "hi", "there", "where", "now", "is", \  "that" , "hi" , "where", "at", "no", "yes", "at"}; |

Let’s find duplicate elements from this list and their duplication count. For example in above vector duplicate strings and their duplication count is as follows,

|  |  |
| --- | --- |
| 1  2  3 | at :: 3  hi :: 2  where :: 2 |

Let’s see how to do that,

## Finding duplicates in a vector

Steps are :

* Create a map of <string , int> type to store the frequency count of each string in vector.
* Iterate over all the elements in vector try to insert it in map as key with value as 1.

If string already exists in map then increment its value by 1.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // Create a map to store the frequency of each element in vector  std::map<std::string, int> countMap;    // Iterate over the vector and store the frequency of each element in map  for (auto & elem : vecOfStings)  {  auto result = countMap.insert(std::pair<std::string, int>(elem, 1));  if (result.second == false)  result.first->second++;  } |

Now iterate over the map and print items whose value is greater than 1 i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Iterate over the map  for (auto & elem : countMap)  {  // If frequency count is greater than 1 then its a duplicate element  if (elem.second > 1)  {  std::cout << elem.first << " :: " << elem.second << std::endl;  }  } |

It will print duplicate elements in vector and their duplication count i.e.

|  |  |
| --- | --- |
| 1  2  3 | at :: 3  hi :: 2  where :: 2 |

### Generic function to find duplicates in vector:

Create a Generic function to get the duplicate elements and their duplication count i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | /\*  \* Generic function to find duplicates elements in vector.  \* It adds the duplicate elements and their duplication count in given map countMap  \*/  template <typename T>  void findDuplicates(std::vector<T> & vecOfElements, std::map<T, int> & countMap)  {  // Iterate over the vector and store the frequency of each element in map  for (auto & elem : vecOfElements)  {  auto result = countMap.insert(std::pair<std::string, int>(elem, 1));  if (result.second == false)  result.first->second++;  }    // Remove the elements from Map which has 1 frequency count  for (auto it = countMap.begin() ; it != countMap.end() ;)  {  if (it->second == 1)  it = countMap.erase(it);  else  it++;    }  } |

Let’s use this generic function to find duplicate elements in vector i.e.

|  |  |
| --- | --- |
| 1  2  3  4 | std::map<std::string, int> duplicateElements;    // Get the duplicate elements in vector  findDuplicates(vecOfStings, duplicateElements); |

Now Iterate over this map to print the duplicate elements with count i.e.

|  |  |
| --- | --- |
| 1  2 | for (auto & elem : duplicateElements)  std::cout << elem.first << " :: " << elem.second << std::endl; |

**Output:**

|  |  |
| --- | --- |
| 1  2  3 | at :: 3  hi :: 2  where :: 2 |

**Complete example is as follows,**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89 | #include <iostream>  #include <vector>  #include <map>  #include <algorithm>  #include <functional>  #include <string>    // Print the contents of vector  template <typename T>  void print(T & vecOfElements, std::string delimeter = " , ")  {  for(auto elem : vecOfElements)  std::cout<<elem<<delimeter;  std::cout << std::endl;  }    /\*  \* Generic function to find duplicates elements in vector.  \* It adds the duplicate elements and their duplication count in given map countMap  \*/  template <typename T>  void findDuplicates(std::vector<T> & vecOfElements, std::map<T, int> & countMap)  {  // Iterate over the vector and store the frequency of each element in map  for (auto & elem : vecOfElements)  {  auto result = countMap.insert(std::pair<std::string, int>(elem, 1));  if (result.second == false)  result.first->second++;  }    // Remove the elements from Map which has 1 frequency count  for (auto it = countMap.begin() ; it != countMap.end() ;)  {  if (it->second == 1)  it = countMap.erase(it);  else  it++;    }  }      int main()  {  // Vector of strings  std::vector<std::string> vecOfStings{ "at" , "hello", "hi", "there", "where", "now", "is", \  "that" , "hi" , "where", "at", "no", "yes", "at"};    print(vecOfStings);    // Create a map to store the frequency of each element in vector  std::map<std::string, int> countMap;    // Iterate over the vector and store the frequency of each element in map  for (auto & elem : vecOfStings)  {  auto result = countMap.insert(std::pair<std::string, int>(elem, 1));  if (result.second == false)  result.first->second++;  }    std::cout << "Duplicate elements and their duplication count " << std::endl;    // Iterate over the map  for (auto & elem : countMap)  {  // If frequency count is greater than 1 then its a duplicate element  if (elem.second > 1)  {  std::cout << elem.first << " :: " << elem.second << std::endl;  }  }    /\*  \* Finding duplicates in vector using generic function  \*/    std::map<std::string, int> duplicateElements;    // Get the duplicate elements in vector  findDuplicates(vecOfStings, duplicateElements);    std::cout << "Duplicate elements and their duplication count " << std::endl;  for (auto & elem : duplicateElements)  std::cout << elem.first << " :: " << elem.second << std::endl;    return 0;  } |

**Output:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | at , hello , hi , there , where , now , is , that , hi , where , at , no , yes , at ,  Duplicate elements and their duplication count  at :: 3  hi :: 2  where :: 2  Duplicate elements and their duplication count  at :: 3  hi :: 2  where :: 2 |

# How to find an element in vector and get its index

Suppose we have a vector of int i.e.

|  |  |
| --- | --- |
| 1 | std::vector<int> vecOfNums = { 12, 45, 54, 33, 2, 7, 8, 22, 43, 19 }; |

Now we want to find if number 22 exists in vector? If yes then what’s its index or position in the vector?

std::vector doesn’t provides any direct function to check if an element exists in vector or not. So let’s see how to do that using STL Algorithms.

## Finding an element in vector using STL Algorithm std::find()

Basically we need to iterate over all the elements of vector and check if given elements exists or not.  
This can be done in a single line using std::find i.e.

|  |  |
| --- | --- |
| 1  2 | // Check if element 22 exists in vector  std::vector<int>::iterator it = std::find(vecOfNums.begin(), vecOfNums.end(), 22); |

It accepts a range and an element to search in the given range. If element is found then it returns an iterator to the first element in the given range that’s equal to given element, else it returns an end of the list.

|  |  |
| --- | --- |
| 1  2  3  4 | if (it != vecOfNums.end())  std::cout << "Element Found" << std::endl;  else  std::cout << "Element Not Found" << std::endl; |

If element is found then we can get its index from the iterator i.e.

|  |  |
| --- | --- |
| 1  2 | // Get index of element from iterator  int index = std::distance(vecOfNums.begin(), it); |

But in practical, we will not have vector of integers always. So, let’s create a generic function for this.

## Generic function to find an element in vector of any type

Let’s create a generic function to search an element in any type of vector i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | /\*  Generic function to find an element in vector and also its position.  It returns a pair of bool & int i.e.    bool : Represents if element is present in vector or not.  int : Represents the index of element in vector if its found else -1    \*/  template < typename T>  std::pair<bool, int > findInVector(const std::vector<T>  & vecOfElements, const T  & element)  {  std::pair<bool, int > result;    // Find given element in vector  auto it = std::find(vecOfElements.begin(), vecOfElements.end(), element);    if (it != vecOfElements.end())  {  result.second = distance(vecOfElements.begin(), it);  result.first = true;  }  else  {  result.first = false;  result.second = -1;  }    return result;  } |

This function tells if given element exists in vector and if yes then it also return its position in the vector.

Let’s use this function to find an element in vector i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | std::pair<bool, int> result = findInVector<int>(vecOfNums, 45);    if (result.first)  std::cout << "Element Found at index : " << result.second <<std::endl;  else  std::cout << "Element Not Found" << std::endl; |

## Finding an element by custom comparator using std::find\_if()

Instead of directly searching by value in the vector , we can search by custom logic too.

Like, in a vector of int check if any multiple of 3 exists i.e.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // Check if any multiple of 3  exists in vector using lambda function as comparator    std::vector<int>::iterator it2 = std::find\_if(vecOfNums.begin(), vecOfNums.end(), [](const int & val){  if (val % 3 == 0)  return true;  return false;  });    if (it != vecOfNums.end())  std::cout << "Multiple of 3 Found : " << \*it2 << std::endl;  else  std::cout << "Multiple of 3 Not Found" << std::endl; |

## Finding an element in vector using C++11 Range Based for loop

We can also iterate over the vector using range based for loop and check if element exists or not.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | bool found = false;  // Iterate over all elements in Vector  for (auto & elem : vecOfNums)  {  if (elem == 22)  {  found = true;  break;  }  }  if(found)  std::cout << "Element Found" << std::endl;  else  std::cout << "Element Not Found" << std::endl; |

Complete example is as follows,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105 | #include <iostream>  #include <vector>  #include <algorithm>    /\*  Generic function to find an element in vector and also its position.  It returns a pair of bool & int i.e.    bool : Represents if element is present in vector or not.  int : Represents the index of element in vector if its found else -1    \*/  template < typename T>  std::pair<bool, int > findInVector(const std::vector<T>  & vecOfElements, const T  & element)  {  std::pair<bool, int > result;    // Find given element in vector  auto it = std::find(vecOfElements.begin(), vecOfElements.end(), element);    if (it != vecOfElements.end())  {  result.second = distance(vecOfElements.begin(), it);  result.first = true;  }  else  {  result.first = false;  result.second = -1;  }    return result;  }    int main()  {  std::vector<int> vecOfNums = { 12, 45, 54, 33, 2, 7, 8, 22, 43, 19 };    /\*  Find an element in vector using std::find  \*/    // Check if element 22 exists in vector  std::vector<int>::iterator it = std::find(vecOfNums.begin(), vecOfNums.end(), 22);    if (it != vecOfNums.end())  {  std::cout << "Element Found" << std::endl;    // Get index of element from iterator  int index = std::distance(vecOfNums.begin(), it);  std::cout <<"Index of element in vector : "<<index<<std::endl;  }  else  {  std::cout << "Element Not Found" << std::endl;  }    std::pair<bool, int> result = findInVector<int>(vecOfNums, 45);    if (result.first)  std::cout << "Element Found at index : " << result.second <<std::endl;  else  std::cout << "Element Not Found" << std::endl;      /\*  \* Finding an element by custom comparator  \*/      // Check if any multiple of 3  exists in vector using lambda function as comparator    std::vector<int>::iterator it2 = std::find\_if(vecOfNums.begin(), vecOfNums.end(), [](const int & val){  if (val % 3 == 0)  return true;  return false;  });    if (it != vecOfNums.end())  std::cout << "Multiple of 3 Found : " << \*it2 << std::endl;  else  std::cout << "Multiple of 3 Not Found" << std::endl;    /\*  Find an element in vector using c++11 range based for loop  \*/    bool found = false;  // Iterate over all elements in Vector  for (auto & elem : vecOfNums)  {  if (elem == 22)  {  found = true;  break;  }  }  if(found)  std::cout << "Element Found" << std::endl;  else  std::cout << "Element Not Found" << std::endl;        return 0;  } |