Vectors in C++ STL

Vectors in C++ are similar to dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.

The major problem with arrays in C++ is the ability to resize. No matter, if you have allocated an array using pointers, you need to know the capacity of that array beforehand, and this creates either wastage of some space or shortage of space once the array gets completely filled.

Vectors in C++ comes with a lot of advantages which are discussed below:

- **Dynamic Size**: Vectors are dynamic in nature, so we can insert as many elements as we want at anytime.
- **Rich Library Functions**: Vectors in C++ comes with a lot of built-in functions to perform some basic operations like insert, delete, find, sort, etc. There are a lot of other functions also which we will see in upcoming lesson.
- **No need to pass size**: One more advantage of vectors is, we don't need to pass an extra variable named size to the functions whenever we are passing a vector to the functions. Size of a vector can be calculated easily as vector_name.size().
- **Can be returned from a function**: Arrays created locally inside a function, cannot be returned from that function. The below code snippet is incorrect:

  ```
  int func()
  {
      int arr[100];

      return arr; // This is not valid, and will generate an error
  }
  ```

  However, vectors can be returned normally like any other variable from a function. Below code snippet is valid:

  ```
  int func()
  {
      vector vec;

      return vec; // This is valid
  }
  ```

- **Default values**: Intially, vectors are intialized with default values. If a vector of type int is created, all of the elements are intially intialized to 0, if a vector of boolean type is created, all elements will be false initlially.
- **Copying Vectors**: Vectors can be copied directly using the **"="** operator like any other variable. Below code, copies content of vector v2 to v1.

  ```
  v1 = v2;
  ```

Note: The above-listed advantages are some of the most popular advantages of the vectors because of which using vectors save a lot of extra time required to write programs. There are many other advantages of vectors which we will see in upcoming lessons.

Vector Declaration and Traversal

Vectors in C++ can be declared in different ways as explained below:

- Declaring vectors without size:

  ```
  vector vector_name;

  This creates an empty vector named vector_name
  of type data_type.
  ```

- Declaring vectors with size:

  ```
  vector vector_name(N);

  The above syntax created a vector named
  vector_name of type data_type of size N
  intially filled with default values.
  ```

  We can also fill the entire vector with an element of our choice at the time of declaration as shown below:

  ```
  vector vector_name(N, val);

  The above syntax created a vector named
  vector_name of type data_type of size N
  intially filled with value val.
  ```

Traversing vector using loop: Vectors can be traversed using loops by first calculating the size of the vector using the size() functions and then running a loop from first to the last index and accessing elements at every index.

> Accessing Vector Elements:
> Just like arrays, elements in vectors are stored at contiguous memory locations and **can be accessed directly using indexes**. An index in vectors ranges from 0 to size-1. For Example, vec[4] gives us the 5th element present in the vector.

Below is a sample program to traverse vectors using for loop:

```cpp

#include<iostream>
#include<vector>

using namespace std;

int main()
{
    vector<int> v;

    v.push_back(10);
    v.push_back(5);
    v.push_back(20);

    for(int i=0; i<v.size(); i++)
        cout<<v[i]<<" ";

    return 0;
}

```

Run

Output:

```
10 5 20
```

Traversing vectors using modified for loop: If we want to traverse the complete vector and print all of the values without modifying them, we can do this simply as:

```
for(data_type x : v)
    cout<< x;

data_type is the type of data stored in the vector
x is the loop variable
v is the actual vector
```

Below is the sample program to illustrate this:

```cpp

#include<iostream>
#include<vector>

using namespace std;

int main()
{
    vector<int> v;

    v.push_back(10);
    v.push_back(5);
    v.push_back(20);

    for(int x:v)
        cout<<x<<" ";

    return 0;
}

```

Run

Output:

```
10 5 20
```

Traversing Vectors using Iterators: Iterators are used to point at the memory addresses of STL containers. We can use iterators along with for loops to traverse and access elements in the vector. The begin() and end() function of iterators are used to traverse a vector. The begin() function returns a pointer pointing to the first element of the vector and end() function returns the pointer pointing to the location just after the last element.

- Create an iterator of Vector type. This can be done as:
  ```
  vector :: iterator itr_name;

  The data_type must be the same as that of the vector.
  ```
- Start iterating over vector using begin() and end() functions.
- Make iterator to point to first element from location returned by begin() function.
- Keep incrementing the iterator until it reaches the last element, i.e. element just before the location returned by end() function.

Note: Instead of declaring the iterator, we can also use the auto keyword instead.

Below program illustrate this:

```cpp

#include<iostream>
#include<vector>

using namespace std;

int main()
{
    vector<int> v;

    v.push_back(10);
    v.push_back(5);
    v.push_back(20);

    // Declaring Iterator
    vector<int> :: iterator itr;

    for(itr = v.begin(); itr!=v.end(); itr++)
        cout<<*itr<<" ";

    cout<<endl;

    // Traversing again using new iterator
    // using auto keyword
    for(auto itr2 = v.begin(); itr2!=v.end();
                                itr2++)
        cout<<*itr2<<" ";

    return 0;
}
```

Run

Output:

```
10 5 20
10 5 20
```

Initializing Vector using an Array

We can initialize a vector with all elements of any other array by simply passing the address of the first element and the address just after the last element.

Below program illustrate this:

```cpp

#include<iostream>
#include<vector>

```

```cpp
using namespace std;

int main()
{
    int arr[] = {10, 5, 20};
    int n = sizeof(arr)/sizeof(arr[0]);

    vector<int> v(arr, arr+n);

    // Traversing vector using new iterator
    // using auto keyword
    for(auto itr = v.begin(); itr!=v.end();
                                      itr++)
        cout<<*itr<<" ";

    return 0;
}
```

Run

Output:

```
10 5 20
```

Note: We can initialize a vector with any other container also by simply passing the address of the first element of that container and address just after the last element.

How vectors work internally?

Dynammically Allocated Arrays: Arrays whose size can be changed at runtime are called Dynamically Allocated arrays. That is arrays, where memory is dynamically allocated during runtime.

Vectors are implemented internally using Dynamically allocated arrays.

Below is a simple implementation of working of vectors:

- Internally uses dynamically allocated arrays.
- If currently allocated space gets full, do the following:
    1. Create a new layer space of double size.
    2. Copy elements from old space to new space.
    3. Free old space.

  So, the basic implementation is done using dynamically allocated arrays. Whenever the array gets full, the compiler creates a new dynamically allocated array of greater size (Twice or thrice or even 4 times, this is compiler dependent) and copies old elements to this new array and free up space for the old array.

  We can use the capacity() function of Vectors to check the current capacity of the vector anytime. For a better understanding, let's create an empty vector and check it's capacity.

```cpp

#include<iostream>
#include<vector>

using namespace std;

int main()
{
    // Create an empty vector
    vector<int> vec;

    // Display its capacity
    cout<<vec.capacity();

    return 0;
}
```

Run

Output:

```
0
```

You can see that for an empty vector, the online compiler allocates no space and hence the capacity is zero. Let us now add elements to the vector and check capacity.

```

#include<iostream>
#include<vector>

using namespace std;

int main()
{
    // Create an empty vector
    vector<int> vec;

    // For 1 element, it allocates only 1 space
    vec.push_back(10);
    cout<<vec.capacity()<<endl;

    // For a new element, it doubles the space
    // to store 2 elements
    vec.push_back(10);
    cout<<vec.capacity()<<endl;

    // For a new element, it doubles space to 4
    // to store the 3rd element
    vec.push_back(10);
    cout<<vec.capacity();

    return 0;
}

```

Run

Output:

```
1
2
4
```

Note: The above output may vary from compiler to compiler as a different compiler may pre-allocates some extra space while creating vectors.

Time Complexity of Inserting Elements

The **average time complexity** of inserting an element in a vector is **O(1)**. Let us see how:

There are two cases, while inserting elements:

1. If there is space for new element in vector, insert it in O(1) time.
2. If the vector is full, double the size of the internal array and insert new element.

Consider, current vector capacity to be **N**. Insert **N+1** elements to this vector:

- For the first N elements, insert operation takes O(1) time complexity.
- For the (N+1)th element, insert operation takes O(N) time as we need to double the internal array of size N and copy every element to this new array.

  Therefore,

  ```
  Average time complexity = {O(1) + O(1) +.....N-3 times...+ O(1) + O(N)}/(N+1)
                          = {O(N) + O(N)}/(N+1)
                          = {2*O(N)}/(N+1)
                          = O(1).
  ```

Sample Problem: Return a List of integer having value less than K

Problem: Given an array of N integers and a value K. The task is to return a list of integers from the given array whose value is less than K.

Examples:

```
Input: arr[] = {17, 10, 20, 13, 7, 9} , K = 15
Output: List = [10, 13, 7, 9]

Input: arr[] = {10, 5, 8, 9, 1, 2} , K = 7
Output: List = [5, 1, 2]
```

Solution: The idea is to use Vector in C++ STL, for creating the output list. We have used Vector to create the output list because Vectors in C++ are implemented using dynamic arrays and can be expanded at runtime.

- Create a Vector to store the output.
- Traverse the input array.
- Push elements with values less than K in the Vector.
- Return the Vector.

Below is the implementation of the above approach:

```cpp

// C++ Program to return a List of integer
// with values less than K

#include<bits/stdc++.h>
using namespace std;

// Function to return a list of integers from
// an array with values less than K
vector<int> fun(int arr[], int n, int k)
{
    // Create a vector for the output list
    vector<int> V;

    // Traverse the array
    for(int i = 0; i < n; i++)
    {
        // Store elements in the output list
        // whose value is less than K
        if (arr[i] < k)
            V.push_back(arr[i]);
    }

    // return the output list
    return V;
}

// Driver Code
int main()
{
```

Run

Output:

```
10 13 7 9
```

Sample Problem : Sorting student data on the basis of Marks scored

Problem: Given two arrays roll_no[] and marks[] of the same size representing the roll numbers and marks scored by the students respectively. The task is to sort this data according to the increasing order of marks. That is, print the roll numbers along with the marks scored in the order of increasing marks.

Examples:

```
Input: roll_no[] = {17, 20, 15, 1, 5}, {80, 75, 93, 78, 84}
Output:
Roll No   Marks
```

```
20        75
1        78
17        80
5        84
15        93
```

Solution: Since for every student, we have information about both roll_no and marks. So, we need to create a vector of pairs to store this detail. A pair will store detail about the roll number and marks of a single student and the vector will store a list of such pairs for every student.

The second thing is to sort this vector according to the second element of pairs, where second element is the marks. We can sort a vector of pairs according to the second element using a comparator function. We will learn this in detail in upcoming tracks.

Below is the implementation of the above approach:

```cpp

// Program to sort students data
// according to marks

#include <bits/stdc++.h>
using namespace std;

// Comparator function to sort according to
// second element
bool sortbysec(const pair<int, int>& p1,
               const pair<int, int> p2)
{
    return p1.second < p2.second;
}

// Function to sort students data
// according to marks
void displaySorted(int roll_no[], int marks[], int n)
{
    // Create a vector of pair to
    // store students data
    vector<pair<int, int> > vp;

    // Traverse the arrays and store elements in vector
    for (int i = 0; i < n; i++) {
        vp.push_back(make_pair(roll_no[i], marks[i]));
    }

    // Sort the vector according to second element
    // using Comparator function
```

Run

Output:

```
Roll No   Marks
20        75
1        78
17        80
5        84
15        93
```

Sample Problem : Keep track of previous indexes after sorting Vector of Pairs

Problem: Given a vector, keep track of the present indexes corresponding to each element and after sorting print element with its previous respective indexes.

Examples:

```
Input:  arr[] = {2, 5, 3, 7, 1}
Output:
{1, 4}
{2, 0}
{3, 2}
{5, 1}
{7, 3}
```

```
Explanation:
Before sorting [index(element)]: [0(2), 1(5), 2(3), 3(7), 4(1)]
After sorting [previous_index(element)]: [4(1), 0(2), 2(3), 1(5), 3(7)]

Input:  arr[] = {4, 5, 10, 8, 3, 11}
Output:
{3, 4}
{4, 0}
{5, 1}
{8, 3}
{10, 2}
{11, 5}
```

Solution: The idea is to store each element with its present index in a vector pair and then sort all the elements of the vector, Finally, print the elements with its index associated with it.

Below is the implementation of the above approach:

```cpp

// C++ implementation to keep track
// of previous indexes
// after sorting a vector

#include <bits/stdc++.h>
using namespace std;

void sortArr(int arr[], int n)
{

    // Vector to store element
    // with respective present index
    vector<pair<int, int> > vp;

    // Inserting element in pair vector
    // to keep track of previous indexes
    for (int i = 0; i < n; ++i) {
        vp.push_back(make_pair(arr[i], i));
    }

    // Sorting pair vector
    sort(vp.begin(), vp.end());

    // Displaying sorted element
    // with previous indexes
    // corresponding to each element
    cout << "Element\t"
         << "index" << endl;
    for (int i = 0; i < vp.size(); i++) {
```

Run

Output:

```
Element     Index
1           4
2           0
3           2
5           1
7           3
```