11/09/2023

VECTOR STL

* **STL** → STL stands for **Standard Template library.**

STL provides a collection of functions that offer common data structures and algorithms to make programming more efficient and convenient.

It means it provides an implementation for data structures like array. Using this implementation, we need not to create an array by our own so that the user or developer can efficiently write code by focusing more on the logics rather than creating array and defining its size.

This implementation will work same as like arrays.

Talking about arrays, STL provides a way that how to create arrays, how to use arrays etc. and by using this way or implementation or the functions provided by the STL for arrays, we will get the same behaviour as array.

So, we can say that the implementation to create dynamic arrays is written in Standard Template library (STL).

__/__/____

* **VECTOR →** Vector is a dynamic sized array which can grow and shrink its size which makes it versatile and efficient data structure for storing and manipulating sequence of elements.

Vector also stores its elements in continuous memory blocks just like arrays so that we can access the each element in the vector by using iterator or loop.

- **Static Memory allocation in array →**

```
#include <iostream>
using namespace std;

void func (int arr[ ], int n){
    for (int i=0; i<n; i++){
        cout << arr[i] << " ";
    }
}
int main( ) {
    int arr[5]= {1,2,3,4,5};
    int n= 5;
    func (arr, n);
}
```

Output = 1 2 3 4 5

__/__/____

Dynamic memory allocation in array →

```
#include<iostream>
using namespace std;

void func( int arr[], int n){
    for (int i=0; i<n; i++){
        cout << arr[i] << " ";
    }
}
int main( ) {

    int n;
    cin>>n;
    int * arr = new int[n]; //dynamic allocation

    for (int i=0; i<n; i++){
        int data;
        cin>>data;
        arr[i] = data;
    }
    func (arr, n);
}
```

Output = 5

1
2
3
4
5
1 2 3 4 5 → Array printed

__/__/__

- Suppose, we have taken input value 5 to add 5 elements in the array. But, now requirement increases and we want to add some more elements in the array. We can't simply do that.
The solution for this is STL which will define the implementation of Vector Data Structure.

In vector Data Structure, we need not to tell the size of vector, just keep inserting the required number of elements in vector.

→ #include <vector>
To include the implementations in the STL.

→ size( )
It returns the size of the vector.

→ push_back( )
To push the elements in the vector from the back.

→ pop_back( )
To pop or remove the elements from the back.

→ Declaration of vector →

```
vector<return_type> vector_variable_name;
```

Eg - `vector<int> v;`

__/__/____

* **Vector printing →**

```
#include <iostream>
# include <vector>
using namespace std;

void print (vector<int> v){
    int size = v.size();  // to find size of vector
    for (int i=0; i<size; i++){
     cout<< v[i] <<" ";  // to print the vector
    }
 }
int main ( ){

    vector<int>v;

    v.push_back (1);
    v.push_back (2);
    v.push_back (3);
    v.push_back (4);

    print (v);  // function call

 }
```

// To push or insert the elements in vector

Output = 1 2 3 4

__/__/__

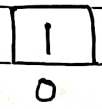
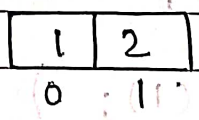
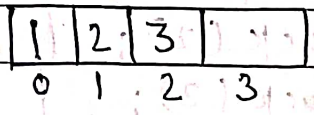
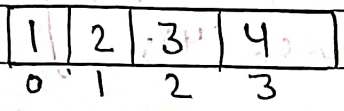
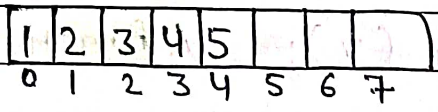
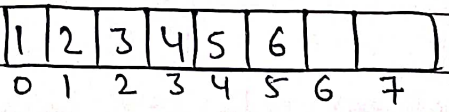
→ **capacity( )**

It returns the storage capacity currently allocated to the vector.

→ **size( )**

It returns the number of elements present in the vector.

* **How vector works behind the scenes?**

		Capacity	size
vector<int>v; → No memory allocation			
v.push_back(1);		1	1
v.push_back(2);		2	2
v.push_back(3);		4	3
v.push_back(4);		4	4
v.push_back(5);		8	5
v.push_back(6);		8	6

and so on.

Here, when the capacity got full, the size is getting doubled.

__/__/___

* To find capacity and size of vector →

```
#include <iostream>
using namespace std;
#include <vector>

int main(){

  vector<int> v;

  while(1){
    int data;
    cin>>data;

    v.push_back(data);

cout<<"capacity"<<v.capacity()<<" "<<"size"<<v.size()
  }
}
```

Output → 1
capacity 1 size 1
2
capacity 2 size 2
3
capacity 4 size 3
4
capacity 4 size 4
5
capacity 8 size 5

__/__/____

* To push_back and pop_back the vector →

```
#include <iostream>
using namespace std;
#include<vector>

void printvec(vector<int>v){
    cout<<"print the vector"<<endl;
     int size = v.size();
    for(int i=0; i<size; i++){
        cout<<v[i]<<" ";
    }
    cout<<endl;
}
int main(){

    vector<int>v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    printvec(v);

    v.pop_back();
    printvec(v);

    v.pop_back();
    printvec(v);
}
```

// To push or insert the elements in vector

// To pop or remove the elements from vector.

Output →

```
Print the vector    1 2 3
Print the vector    1 2
Print the vector    1
```

__/__/____

* To take input from user for vector size and inserting more elements without asking user→

```
#include <iostream>
#include <vector>
using namespace std;

void print(vector<int>v){
    cout<<"print the vector"<<endl;
    int size = v.size();
    for(int i=0; i<size; i++){
        cout<<v[i]<<" ";
    } cout<<endl;
}
int main(){
    vector<int>v;
    int n; cin>>n;
    for(int i=0; i<n; i++){
        int d; cin>>d;
        v.push_back(d);
    }
    print(v);
// for push_back more elements without asking user
    for(int i=0; i<5; i++){
        v.push_back(6);
    }
    print(v);
}
```

Output → 5 (n) , 1 2 3 4 5 (d)

print the vector → 1 2 3 4 5

print the vector → 1 2 3 4 5 6 6 6 6 6

→ Ways to access elements in vector & print them-

```
cout << v[i] << " ";
cout << v.at(i) << " ";
```

Here, v is the vector_name and i is the iteration in the loop.

→ clear( )

This function is used to clear the whole vector.

* Types of vector initialisation →

1. Default with no data, 0 size →

```
vector <int> v;
```

2. Initialisation with n size with specific data →

```
vector<int> v1 (size, specific_data)
Vector<int> v1 (5, 2);
```

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

4. `vector <int> v3{1, 2, 3, 4, 5};`

NOTE → We can always push_back, pop_back the data in vector whether we have given the input n as size as vector is dynamic in nature.

→ **COPY VECTOR →**

```
vector<int> arr1 = {1, 2, 3, 4, 5};
vector <int> arr2 (arr1);
```

If print arr2 →

output = {1, 2, 3, 4, 5}

→ **Front and Back element →**

```
vector<int>v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
```

For front element → `v[0];` or `v.front();`

For back element → `v[v.size()-1];` or `v.back();`

→ **front()** – Used to find the front element.

→ **Back()** – Used to find the back element.

* **FEATURES OF VECTOR →**

1. Continuous memory allocation
2. Dynamic sizing
3. Automatic reallocation → Accomodate new elements
4. Size and capacity
5. Array-like access → Using square brackets ([]) and can also access by at() member function.