Standard Template Library

1. Vector

In a simple static array, we can not redefine its size. So it is optimal to make static arrays only where the size of the array doesn't change according to the user's needs. Also, the size should be defined before the compilation because stack memory is assigned at the time of compilation of the program. For example,

- **int arr[20];** **Here, the stack memory that the arr requires is fixed**

- **int n;**
  cin >> n;
  int arr[n];
  In this case some random amount of stack memory is assigned to arr but that might be short than what the user might require.

The above problem can be solved using Vectors. **A Vector is a dynamic array that has the ability to modify its shape whenever we perform insertion or deletion in it.** Just like an array, Vector elements are placed in contiguous storage so that they can be accessed using iterators.

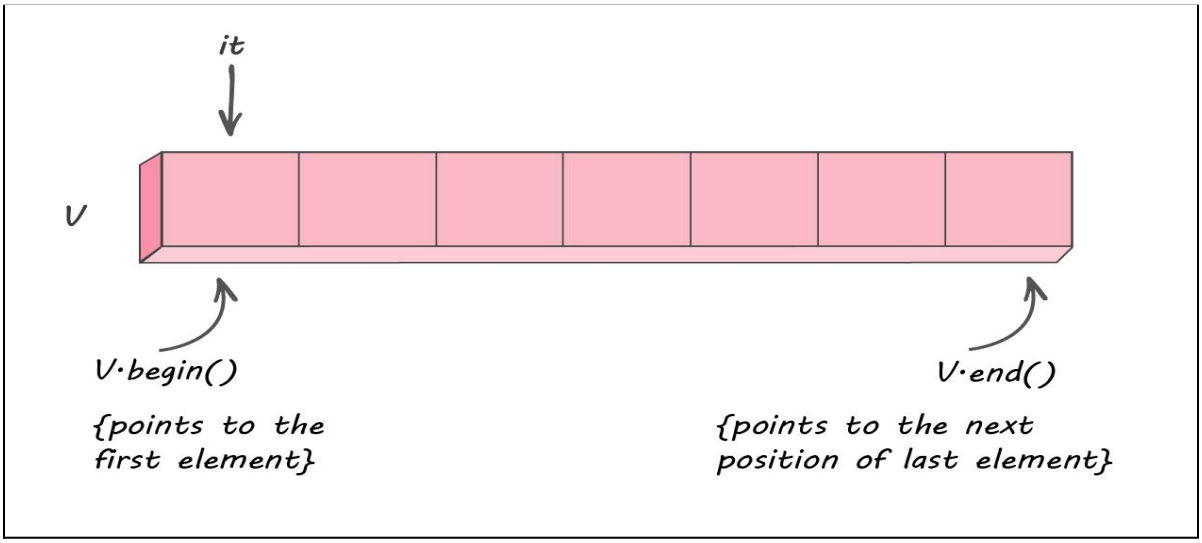
Vector is already present in the standard template library of C++ and can be included in any file using **#include<vector>**

Any new element can be inserted at the end of the vector using the **push_back()** function, which takes differential time because there might be

resizing the vector. While removing the last element takes constant time because no resizing happens.

```
#include<iostream>
#include<vector>

using namespace std;

int main(){

    // while making a vector we need not give the size of the vector
    vector<int> V ;

    // we insert the element at the last index of the vector and
    // the vector automatically reassigns its size
    for(int i=0; i<5; i++){
        V.push_back(i+1);
    }

    return 0;
}
```

There are predefined functions like **begin()**, **end()**, **size()**, etc., that can be used along with a vector.



If we need to traverse the entire vector we can do it using creating an iterator:

```
    // creating an iterator and traversing the vector using it
    vector <int> :: iterator it;
    for(it = v.begin(); it != v.end(); it++){
        cout << *it << endl;
    }
```

2. Stacks

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end (top) and an element is removed from that end only. Stack uses an encapsulated object of either vector or deque (by default) or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

Various functionalities of stacks are explained in the code below:

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

int main() {
        stack<char>s;

        s.push('a');
        s.push('b');
        s.push('c');

        cout<<s.top()<<"\n";
```

```
        s.pop();

        cout<<s.top()<<"\n";
        cout<<s.size()<<"\n";

        if(!s.empty()) cout<<"true"<<"\n";
        else cout<<"false"<<"\n";

        while(!s.empty()){
                s.pop();
        }

        if(!s.empty()) cout<<"true"<<"\n";
        else cout<<"false"<<"\n";
}
```

3. Queues and Deques

Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front. Queues use an encapsulated object of deque or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

Double ended queues are sequence containers with the feature of expansion and contraction on both the ends. They are similar to vectors, but are more efficient in case of insertion and deletion of elements. Unlike vectors, contiguous storage allocation may not be guaranteed. Double Ended Queues are basically an implementation of the data structure double ended queue. A queue data structure allows insertion only at the end and deletion from the front. This is like a queue in real life, wherein people are removed from the front and added at the

back. Double ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.

The functions for deque are the same as vectors, with an addition of push and pop operations for both front and back.

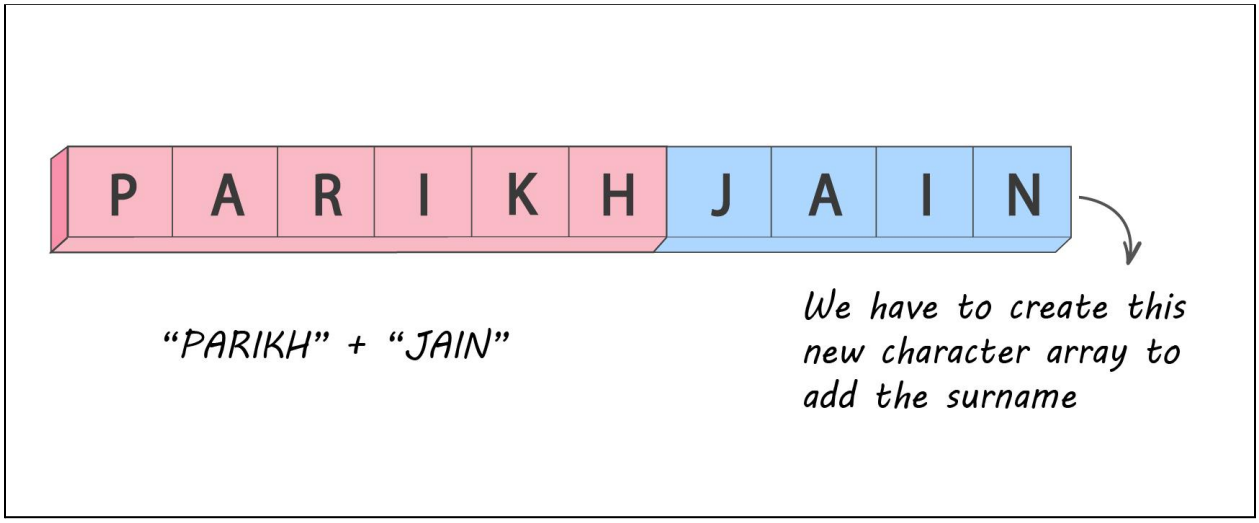
Various functionalities of queues and deques are explained in the code below:

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

int main() {
	queue<int>q;

	q.push(1);
	q.push(2);
	q.push(3);

	cout<<q.front()<<"\n";
	cout<<q.back()<<"\n";

	q.pop();

	cout<<q.front()<<"\n";
	cout<<q.back()<<"\n";

	cout<<"size of queue is: "<<q.size()<<"\n";


	deque<int>dq;

	dq.push_back(1);
	dq.push_front(2);
	dq.push_front(3);

	cout<<dq.front()<<"\n";
	cout<<dq.back()<<"\n";
```

```
        dq.pop_front();
        dq.pop_back();

        cout<<dq.front()<<"\n";
        cout<<dq.back()<<"\n";

        return 0;
}
```

4. Strings

A character array is a static array, and more memory can not be used at run time. Let us suppose we made a character array “PARIKH”. Now we need to add the surname but the length/size of the character array is static, so we have to make a new character array “JAIN” and then use some iteration method to get the full name.



The above problem can be made way easier with a String. A String is a class that defines objects that are represented as a stream of characters.

String class is present in the standard template library of C++, and can be included in any file using **#include<string>**

```
#include<iostream>
#include<string>

using namespace std;


int main(){

    string s = "PARIKH";
    cout << s << endl; // prints PARIKH
    s = "PARIKH JAIN";
    cout << s << endl; // prints PARIKH JAIN
    return 0;
}
```

Various functionalities of strings are explained in the code below with proper comments:

```
int main(){

    string s = "PARIKH";
    string s1(s, 2, 4);//from index 2 upto 4 characters are copied to s1 from s

    // compare string s and s1
    if(s1.compare(s) == 0){
        cout << s1 << " is equal to " << s << endl;
    } else {
        cout << s1 << " is not equal to " << s << endl;
```

```
    }

    // substring from index 1 in s upto 4 characters is stored in s2
    string s2 = s.substr(1, 4);
    cout<< s2 << endl; // prints ARIK
}
```

5. Pair Class

The Pair Class contains a pair of homogeneous or heterogeneous values. It can be understood as a container in which the pair of values are identified by **“first”** and **“second”** and can be of different or the same data type.

Pair class is present in the standard template library of C++, and can be included in any file using **#include<utility>**

Syntax for creating a pair class:

pair < first_data_type, second data_type > p(first, second);

Example :

pair < int, char > p(1, ‘a’);

To access the elements, we use variable name followed by dot operator followed by the keyword first or second:

cout << p.first << " " << p.second << endl;

```
#include<iostream>
#include<utility>

using namespace std;

int main(){
```

```
    // various ways to initialize a pair

    pair<int, char> p;
    p = make_pair(2, 'b');

    pair<int, int> p1(3, 4);

    pair<int, char> p2(1, 'a');

    // accessing elements inside a pair using first and second

    cout << p.first << " " << p.second << endl; // output is 2 b
    cout << p1.first << " " << p1.second << endl; // output is 3 4
    cout << p2.first << " " << p2.second << endl; // output is 1 a

    return 0;
}
```

6. Set

Set is a container which contains unique elements. The element itself is identified by its value. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.
A set can be used in a problem where we have to know whether the element is present or not or suppose we have an array = [1, 2, 3, 4, 5, 6, 5]. Here the number 5 is occurring two times in the array but when we make a set using this array the occurrence of the number 5 is only once just like all the other numbers / elements.

Set is present in the standard template library of C++, and can be included in any file using **#include<set>**

Some functionalities of Set include:

- **begin()** – Returns an iterator to the first element in the set.
- **end()** – Returns an iterator to the theoretical element that follows the last element in the set.
- **size()** – Returns the number of elements in the set.
- **empty()** – Returns whether the set is empty.
- **find()** - Returns an iterator if the element is present.

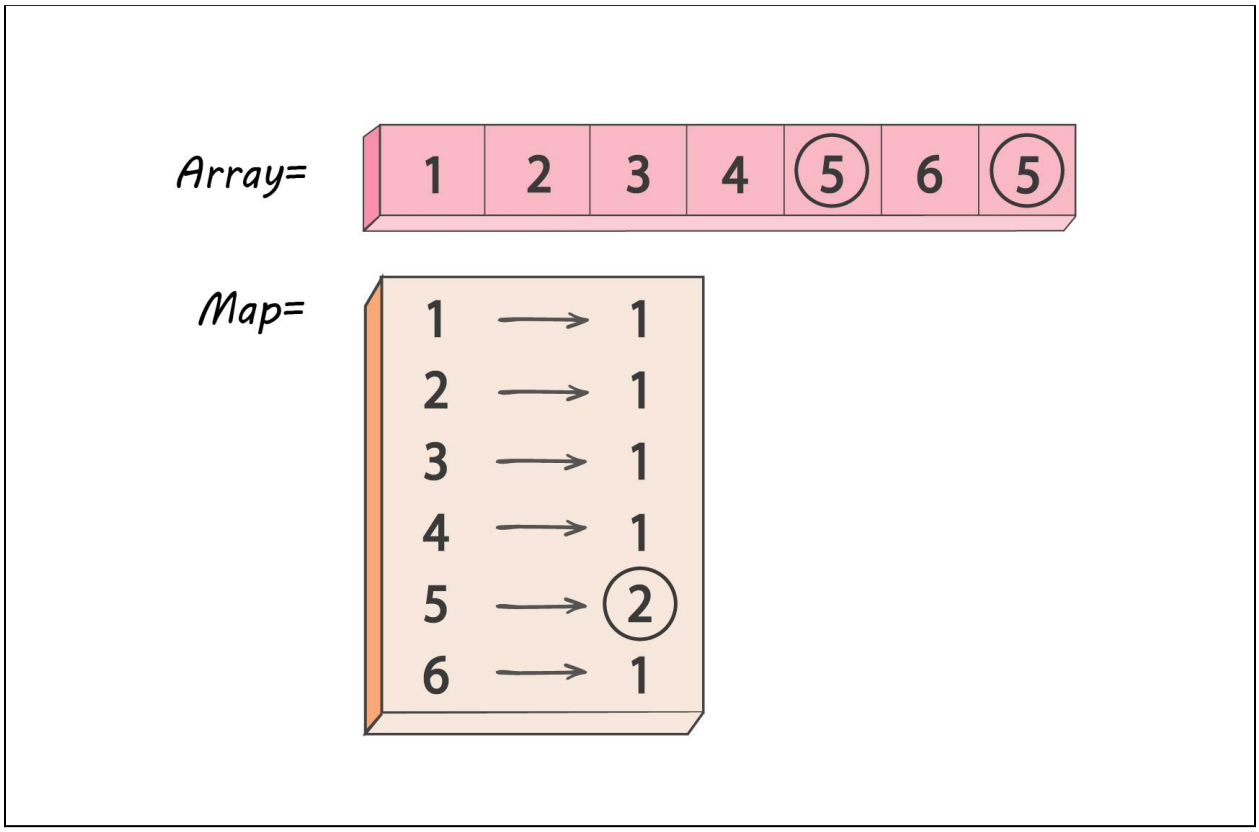
A set is generally implemented using a balanced binary search tree which takes **O(log(N))** time to find an element.

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

int main(){

    // initializing a set
    set<int> s;
    int arr[] = {1,2,3,4,5,6,5};

    // inserting elements into a set
    for(int i=0; i<7; i++){
        s.insert(arr[i]);
    }
    // creating an iterator to traverse through the set
    set<int>::iterator it;

    for(it=s.begin(); it!=s.end(); it++){
        cout << *it << endl;
    }
    // finding elements in set
    if(s.find(7) == s.end()){
        cout << "Element not found" << endl;
    } else {
        cout << "Element found" << endl;
```

```
    }
    return 0;
}
```

7. Map

Consider the previous example used in sets where we made a set using the **array = [1, 2, 3, 4, 5, 6, 5]**. Now if we want to store the frequency of each element, we can not do that by using a set. Therefore, we use a map that stores their frequencies like a **key-value pair in addition to storing unique elements.**



Map is present in the standard template library of C++, and can be included in any file using **#include<map>**

Syntax to make a map:

map < int, int > M;

Some functionalities of Set include:

- **begin()** – Returns an iterator to the first element in the map
- **end()** – Returns an iterator to the theoretical element that follows last element in the map
- **size()** – Returns the number of elements in the map
- **empty()** – Returns whether the map is empty
- **pair insert(keyValue, mapValue)** – Adds a new element to the map

The **keyValue** and **mapValue** can be accessed using **M -> first** and **M -> second.**

A Map is generally implemented using a balanced binary search tree which takes **O(log(N))** time to do operations.

```
#include<iostream>
#include<map>

using namespace std;

int main(){


    int arr[] = {1,2,3,4,5,6,5};
    map<int, int> m;

    for(int i=0; i<7; i++){
        m[arr[i]] = m[arr[i]] + 1;
    }
```

```
    map<int, int>::iterator it;

    for(it=m.begin();it!=m.end();it++){
        cout<<it->first << " :" << it->second<<endl;
    }
    cout<<endl;
    m.erase(1);
    for(it=m.begin();it!=m.end();it++){
        cout<<it->first << " :" << it->second<<endl;
    }
}
```

8. unordered_map

A normal Map is implemented using a balanced binary search tree which takes **O(log(N))** time to perform operations like insert , search etc but an **unordered_map** is implemented using a hash table which takes **O(1)** time in the average case to perform operations like insert, search, etc. However, in the worst case scenario it takes **O(N)** time.
unordered_map is present in the standard template library of C++, and can be included in any file using **#include<unordered_map>**

```
#include<iostream>
#include<unordered_map>

using namespace std;

int main(){


    int arr[] = {1,2,3,4,5,6,5};
    unordered_map<int,int> m;

    for(int i=0;i<7;i++){
```

```
        m[arr[i]] = m[arr[i]]+1;
    }

    unordered_map<int,int>::iterator it;
    for(it=m.begin();it!=m.end();it++){
        cout<<it->first << " :" << it->second<<endl;
    }
    cout<<endl;
    m.erase(1);
    for(it=m.begin();it!=m.end();it++){
        cout<<it->first << " :" << it->second<<endl;
    }

}
```