Implementing Arrays in C++ using STL

We already have discussed the basic declaration of arrays. Arrays can also be implemented using some built-in classes available in the C++ Standard Template Library.

Some of the most commonly used classes for implementing sequential lists or arrays are:

- Vector
- List

Let's look at each of these classes in details.

Vector

Vector in C++ STL is a class that represents a dynamic array. The advantages of vector over normal arrays are,

- We do not need to pass size as an extra parameter when we pass vector.
- Vectors have many in-built functions for erasing an element, inserting an element etc.
- Vectors support dynamic sizes, we do not have to initially specify the size of a vector. We can also resize a vector.
- There are many other functionalities vector provide.

Vectors are same as 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. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

To use the Vector class, include the below header file in your program:

```
#include< vector >
```

Declaring Vector:

```
vector<Type_of_element >vector_name;

Here,Type_of_element can be any valid C++ data type,
or can be any other container also like Pair, List etc.
```

Some important and commonly used functions of Vector class are:

- **begin( )** – Returns an iterator pointing to the first element in the vector.
- **end( )** – Returns an iterator pointing to the theoretical element that follows the last element in the vector.
- **size( )** – Returns the number of elements in the vector.
- **capacity( )** – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
- **empty( )** – Returns whether the container is empty.
- **push_back( )** – It push the elements into a vector from the back.
- **pop_back( )** – It is used to pop or remove elements from a vector from the back.
- **insert( )** – It inserts new elements before the element at the specified position.
- **erase( )** – It is used to remove elements from a container from the specified position or range.
- **swap( )** – It is used to swap the contents of one vector with another vector of same type and size.
- **clear( )** – It is used to remove all the elements of the vector container.
- **emplace( )** – It extends the container by inserting new element at position.
- **emplace_back( )** – It is used to insert a new element into the vector container, the new element is added to the end of the vector.

Below program illustrate the above methods:

```
// C++ program to illustrate the above functions
#include <iostream>
#include <vector> using namespace std;

int main()
{
```

```
    vector<int> v;

    // Push elements
    for (int i = 1; i <= 5; i++)
        v.push_back(i);

    cout << "Size : " << v.size();

    // checks if the vector is empty or not
    if (v.empty() == false)
        cout << "\nVector is not empty";
    else        cout << "\nVector is empty";

    cout << "\nOutput of begin and end: ";
    for (auto i = v.begin(); i != v.end(); ++i)
        cout << *i << " ";

    // inserts at the beginning
    v.emplace(v.begin(), 5);
    cout << "\nThe first element is: " << v[0];

    // Inserts 20 at the end
    v.emplace_back(20);
    int n = v.size();
    cout << "\nThe last element is: " << v[n - 1];

    // erases the vector
    v.clear();
    cout << "\nVector size after erase(): " << v.size();

    return 0;
}
```

Output:

```
Size : 5
Vector is not empty
Output of begin and end: 1 2 3 4 5
```

```
The first element is: 5
The last element is: 20
Vector size after erase(): 0
```

List

Lists are sequence containers that allow non-contiguous memory allocation. List in C++ STL implements a doubly linked list and not arrays. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about doubly linked lists. For implementing a singly linked list, we can use **forward_list** class in C++ STL.

To use the List class, include the below header file in your program:

```
#include< list >
```

Declaring List:

```
list<Type_of_element >list_name;

Here,Type_of_element can be any valid C++ data type,
or can be any other container also like Pair, List etc.
```

Some important and commonly used functions of List are:

- **front( )** – Returns the value of the first element in the list.
- **back( )** – Returns the value of the last element in the list.
- **push_front(g)** – Adds a new element ‘g’ at the beginning of the list.
- **push_back(g)** – Adds a new element ‘g’ at the end of the list.
- **pop_front( )** – Removes the first element of the list, and reduces the size of the list by 1.
- **pop_back( )** – Removes the last element of the list, and reduces the size of the list by 1.
- **begin( )** and **end( )** – begin() function returns an iterator pointing to the first element of the list.
- **empty( )** – Returns whether the list is empty(1) or not(0).
- **insert( )** – Inserts new elements in the list before the element at a specified position.
- **reverse( )** – Reverses the list.

- **size( )** – Returns the number of elements in the list.
- **sort( )** – Sorts the list in increasing order.

Below program illustrate the above functions:

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

//function for printing the elements in a list
void showlist(list <int> g)
{
    list <int> :: iterator it;
    for(it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}

int main()
{

    list <int> gqlist1, gqlist2;


    for (int i = 0; i < 10; ++i)
    {
        gqlist1.push_back(i * 2);
        gqlist2.push_front(i * 3);
    }
    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);

    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);

    cout << "\ngqlist1.front() : " << gqlist1.front();
    cout << "\ngqlist1.back() : " << gqlist1.back();
```

```
    cout << "\ngqlist1.pop_front() : ";
    gqlist1.pop_front();
    showlist(gqlist1);

    cout << "\ngqlist2.pop_back() : ";
    gqlist2.pop_back();
    showlist(gqlist2);

    cout << "\ngqlist1.reverse() : ";
    gqlist1.reverse();
    showlist(gqlist1);

    cout << "\ngqlist2.sort(): ";
    gqlist2.sort();
    showlist(gqlist2);

    return 0;

}
```

Output :-

```
List 1 (gqlist1) is :   0      2    4     6    8    10
12      14      16   18
List 2 (gqlist2) is :   27   24   21    18   15    12   9
6       3       0

gqlist1.front() : 0
gqlist1.back() : 18
gqlist1.pop_front() :   2       4       6       8       10
12      14      16      18

gqlist2.pop_back() :    27    24      21    18     15
12     9       6       3

gqlist1.reverse() :       18      16    14    12     10
8      6       4        2
```

```
gqlist2.sort():           3      6     9    12    15
18     21      24       27
```