Containers

Standard Containers
A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Containers replicate structures very commonly used in programming: dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)...

Many containers have several member functions in common, and share functionalities. The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on the efficiency of some of its members (complexity). This is especially true for sequence containers, which offer different trade-offs in complexity between inserting/removing elements and accessing them.

stack, queue and priority_queue are implemented as *container adaptors*. Container adaptors are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes (such as deque or list) to handle the elements. The underlying container is encapsulated in such a way that its elements are accessed by the members of the *container adaptor* independently of the underlying *container* class used.

Container class templates

Sequence containers:
array Array class (class template)
vector Vector (class template)
deque Double ended queue (class template)
forward_list Forward list (class template)
list List (class template)

Container adaptors:
stack LIFO stack (class template)
queue FIFO queue (class template)
priority_queue Priority queue (class template)

Associative containers:
set Set (class template)
multiset Multiple-key set (class template)
map Map (class template)
multimap Multiple-key map (class template)

std::string class in C++

C++ has in its definition a way to represent **sequence of characters as an object of class**. This class is called std:: string. String class stores the characters as a sequence of bytes with a functionality of allowing **access to single byte character**.

std:: string vs Character Array

- A character array is simply an **array of characters** can terminated by a null character. A string is a **class which defines objects** that be represented as stream of characters.

- Size of the character array has to **allocated statically**, more memory cannot be allocated at run time if required. Unused allocated **memory is wasted** in case of character array. In case of strings, memory is **allocated dynamically**. More memory can be allocated at run time on demand. As no memory is preallocated, **no memory is wasted**.

- There is a **threat of array decay** in case of character array. As strings are represented as objects, **no array decay** occurs.
- **Input Functions**
- **1. getline()** :- This function is used to **store a stream of characters** as entered by the user in the object memory.
- **2. push_back()** :- This function is used to **input** a character at the **end** of the string.
- **3. pop_back()** :- Introduced from C++11(for strings), this function is used to **delete the last character** from the string.

```
// C++ code to demonstrate the working of
// getline(), push_back() and pop_back()
#include<iostream>
#include<string> // for string class
using namespace std;
int main()
{
    // Declaring string
    string str;

    // Taking string input using getline()
    // "geeksforgeek" in givin output
    getline(cin,str);

    // Displaying string
    cout << "The initial string is : ";
    cout << str << endl;

    // Using push_back() to insert a character
    // at end
    // pushes 's' in this case
    str.push_back('s');

    // Displaying string
    cout << "The string after push_back operation is : ";
    cout << str << endl;

    // Using pop_back() to delete a character
    // from end
    // pops 's' in this case
    str.pop_back();

    // Displaying string
    cout << "The string after pop_back operation is : ";
    cout << str << endl;

    return 0;

}
```

Input:

```
geeksforgeek
```

Output:

```
The initial string is : geeksforgeek
The string after push_back operation is : geeksforgeeks
The string after pop_back operation is : geeksforgeek
```

Capacity Functions

3. capacity() :- This function **returns the capacity** allocated to the string, which can be **equal to or more than the size** of the string. Additional space is allocated so that when the new characters are added to the string, the **operations can be done efficiently**.

4. resize() :- This function **changes the size of string**, the size can be increased or decreased.

5.shrink_to_fit() :- This function **decreases the capacity** of the string and makes it equal to its size. This operation is **useful to save additional memory** if we are sure that no further addition of characters have to be made.

Iterator Functions

7. begin() :- This function returns an **iterator** to **beginning** of the string.

8. end() :- This function returns an **iterator** to **end** of the string.

9. rbegin() :- This function returns a **reverse iterator** pointing at the **end** of string.

10. rend() :- This function returns a **reverse iterator** pointing at **beginning** of string.

Manipulating Functions

11. copy("char array", len, pos) :- This function **copies the substring in target character array** mentioned in its arguments. It takes 3 arguments, **target char array, length to be copied and starting position in string to start copying.**

12. swap() :- This function **swaps** one string with other.