Vectors

I bet most of you must have been exposed to arrays and linked lists by now, and you must have experienced the short comings of both the data types.

Arrays are *fast but rigid*, *accessing* elements can be achieved in O(1) time (Random access), but *insertion* or *deletion* are quite tedious and can be O(n) in the worst case, and also arrays space inefficient, as they are fixed and cannot be resized easily (without using realloc). **Linked Lists** are quite space efficient, but are equally messy because of pointer operations. *Insertion* and *deletion* of elements in a linked list is easy, and requires no shifting of previous/next elements, but as random access is not possible in an linked list, *accessing* the i^{th} element can be O(n) in the worst case.

Many of you must have surely wondered if there is any Data Structure that combined the efficiencies and capabilities of both Arrays and Linked Lists, and yes, there is one, that is a **Vector.**

Declaring Vectors:

Vectors have 3 different constructors for initialization:

1. *vector<T> g;*
2. *vector<T> g(n);* `// n is the size, you want the vector to be`
3. *vector<T> g(n, u);* `// this initializes the vector of size n with all elements as u`

Here T refers to the type of the elements in the vector. It can an int, double, char, any object, or even a self-defined data structure.

Accessing elements in a Vector:

Elements of a vector can be accessed randomly, which means, yes, g[i] works, and will fetch you the i^{th} element of the vector. *g[i] = 5* will set the i^{th} element to 5. Any type of arithmetic operation can now be performed on g[i] as we do on regular variable.

Insertion in a Vector:

The *push_back()* method can be used to add a new elements to a vector. It will take the element to the inserted as a parameter, and will add the element to the end of the vector and the size of the vector is incremented by 1.

Directly an element can be scanned into the i^{th} position of a vector *(cin >> g[i]),* as in an array, but this is only applicable with a vector which is initialized with constructor-2, as if the size is not mentioned, scanning an element into a random position, whose bounds aren't specified might create a memory issue, (you get that, right :P).

Deletion in a Vector:

An element in a vector can be *erased* or deleted by using the erase() method. It takes an iterator to the element to be erased as a parameter.

g.begin() returns the iterator to the first element, and the iterator of the i^{th} element can be accessed by (*g.begin() + i*). Similarly, *g.end()* returns the iterator to the last element of the vector. We must be careful as if try to erase an location which is *null,* it will lead to a segmentation fault.

Traversing a Vector:

```
for(int i = 0; i < n; i++)
	cout << g[i];

for(vector<T> :: iterator it = g.begin(); it != g.end(); it++)
	cout << *it;
```

In the above example, *vector<T> :: iterator* can be replaced with the *auto* keyword. Upon using the *auto* keyword, the compiler decides the type of the iterator automatically during run-time.

```
for(auto it = g.begin(); it != g.end(); it++)
	cout << *it;

for(auto it : g)
	cout << it;
```

In the above example, a copy of the element is created and stored in *it.* Changes made to *it* will not be reflected in the original vector. It is generally used to print the container.

Other methods related to a Vector:

1. *front() / back():* returns the first or last element of a vector respectively.
2. *size():* returns the size of the vector.
3. *resize():* resizes the vector to the size specified as the parameter.
4. *pop_back():* pops or removes the last element of the vector. Decrements the size by 1. Similar to g.erase(g.end()).
5. *fill(g.begin(), g.end(), u):* All the elements of the vector are set to 'u'.
6. *iota(g.begin(), g.end(), u):* Elements of the array are filled with sequentially increasing values starting with 'u'. Kind of similar to *range* in python.
7. *find(g.begin(), g.end(), u):* Returns the iterator to the element to be searched, i.e, 'u', else returns g.end(), iterator to the last element.
8. *accumulate(g.begin(), g.end(), 0):* Returns the sum of all the elements of the vector.
9. *insert(it, u):* This method takes an iterator and the element to be inserted as parameters. It inserts the given element at the specified location. Please note that, this insertion maybe O(n) in worst case. Vectors allow easy insertion, which occur by realloc.

General Methods:

1. *sort(g.begin(), g.end())*
2. *lower_bound(g.begin(), g.end(), u)*

3. *upper_bound(g.begin(), g.end(), u)*
4. *reverse(g.begin(), g.end())*

Other generic methods related to vector can be found at:
https://www.geeksforgeeks.org/vector-in-cpp-stl/