Introduction to STL and Advanced STL

theroyakash

Indian Institute of Technology Madras

August 16, 2023

Introduction

- C++ Standard Template Library (STL) provides standard ways for storing and processing data.

Introduction

- C++ Standard Template Library (STL) provides standard ways for storing and processing data.
- There are three components of STL.

Introduction

- C++ Standard Template Library (STL) provides standard ways for storing and processing data.
- There are three components of STL. These are **Algorithms**, **Containers** and **Iterators**.

STL Containers

- Containers is used to a collection of data, which can be either built-in types or user-defined class objects.

STL Containers

- Containers is used to a collection of data, which can be either built-in types or user-defined class objects.
- Two main types of containers,

STL Containers

- Containers is used to a collection of data, which can be either built-in types or user-defined class objects.
- Two main types of containers, Sequence container and Associative container,

STL Containers

- Containers is used to a collection of data, which can be either built-in types or user-defined class objects.
- Two main types of containers, Sequence container and Associative container,
- Array, Vectors, Lists are Sequence container, data is stored sequentially.

STL Containers

- Containers is used to a collection of data, which can be either built-in types or user-defined class objects.
- Two main types of containers, Sequence container and Associative container,
- Array, Vectors, Lists are Sequence container, data is stored sequentially.
- There are some special purpose containers that are derived from base containers like Stack, Queue, Priority Queue.

Vector

- **Why Vector?**

Vector

- **Why Vector?**
- If you want to use an array of some specific data structure but you don't know how many of them are there beforehand.

Vector

- **Why Vector?**
- If you want to use an array of some specific data structure but you don't know how many of them are there beforehand.
- You want to use a dynamically sized (resizable) array.

Vector Initialization

Initialization

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

using std::vector;

int main () {
    vector<int> v1; // start empty vector
    vector<int> v2(10, -1); // start 10 sized vector
    // create 10x10 vector filled with -1
    vector<vector<int>> v4(10, vector<int>(10, -1));
    return 0;
}
```

Vector operations

Vector Push and Pop from back

```
int main () {
    std::vector<int> v1; // start empty vector
    v1.push_back(11);
    v1.push_back(12);
    v1.pop_back();
    std::cout << v1.back() << std::endl;
}
```

Other Operations

Vector other important functions

```
int main() {
    std::vector<int> v1 = {1,2,3,4};
    v1.clear(); // clear all elements in O(n)
    size_t size = v1.size(); // current size of vector.
    cout << v1[2]; // get the 3rd element from array.
}
```

Traversing on Vector

Traversing on Vector

```
int main() {
    vector<int> v;
    for (int i = 0; i < v.size(); i++) cout << v[i] << " ";

    for (int data : v) { cout << data << " "; }
    for (auto data : v) { cout << data << " "; }
}
```

Traversing on 2D-Vector

Traversing on 2D-Vector

```
int main() {
    vector<vector<int>> v;
    for (vector<int> row : v) {
        for (int element : row) {
            cout << element << " ";
        }
    }
}
```

Traversing on 2D-Vector

Alternate ways of traversing on 2D-Vector

```
int main() {
    vector<vector<int>> v;
    for (int i = 0; i < v.size(); i++) {
        for (int j = 0; j < v[i].size(); j++) {
            cout << v[i][j];
        }
    }
}
```

Pair

Pair is an important composite data structure, made up of two primitive or composite data type. **To define pair**

```
int main() {
    pair<int, int> p1; // <int, int> pair
    pair<int,pair<int,char>> p2 = {1, {1, 'j'}};
    pair<int, vector<int>> p3 = {1, {1,2,3,4,5}};
    pair<int, pair<int, pair<int, int>>> p4; // unlimited nesting
}
```

Accessing elements from Pair

To access the first element from pair use `p1.first` and to access the second element from pair use `p1.second`.

Access Example

```
int main() {
    pair<int, pair<int, int>> p1 = {1, {2, 3}};
    cout << p1.first << p2.second.first << p2.second.second << endl;
}
```

Sorting + Searching

Before introducing other data structure in the STL library, I'll show you some algorithms and iterator access on vector which is used often.

- Iterators
- Sorting
- Searching

Iterators

- Similar to pointers in C, C++ has Iterators

Iterators

- Similar to pointers in C, C++ has Iterators
- `vector<int>::iterator it = v.begin();` returns a *Pointer* to the first element of vector,

Iterators

- Similar to pointers in C, C++ has Iterators
- `vector<int>::iterator it = v.begin();` returns a *Pointer* to the first element of vector,
- Or you can use `auto it = v.begin();`

Iterators

- Similar to pointers in C, C++ has Iterators
- `vector<int>::iterator it = v.begin();` returns a *Pointer* to the first element of vector,
- Or you can use `auto it = v.begin();`
- For example if you have an array $v = [10, 12, 13, 14]$ then `*it` would return 10,

Iterators

- Similar to pointers in C, C++ has Iterators
- `vector<int>::iterator it = v.begin();` returns a *Pointer* to the first element of vector,
- Or you can use `auto it = v.begin();`
- For example if you have an array $v = [10, 12, 13, 14]$ then `*it` would return 10,
- Similar to pointer you can increase and decrease them, `it++;` and then `*it` would return 12.

Last Iterator

- `vector<int>::iterator it = v.end();` points to a non-existent element sits after the last element.

Last Iterator

- `vector<int>::iterator it = v.end();` points to a non-existent element sits after the last element.
- Hence `*it` would dereference nothing when `it = v.end();`.

Sorting

Sorting and searching is the most common thing you do on a vector.

```
#include <algorithm>

int main () {
    vector<int> v = {4,3,2,1};
    std::sort(v.begin(), v.end());
    for (auto i : v) { cout << i << " "; }
}
```

Custom Sorting

What to do when you have a vector of custom data structure?

Custom Sorting

What to do when you have a vector of custom data structure?
For that we need to design custom comparators.

Custom Comparators Showcase

```
int main() {
    vector<pair<int, int>> v = {{1,2}, {-3,4}, {-12, 12}};
    sort(v.begin(), v.end(), [](const auto &a, const auto &b) {
        return a.first < b.first;
    });
}
```

Custom Comparators Showcase II

Using this you can define your own rule for sorting, for example following shows how to sort in decreasing order.

Custom Comparators Showcase II

Using this you can define your own rule for sorting, for example following shows how to sort in decreasing order.

```
int main() {
    vector<int> v = {1,2,3,4};
    std::sort(v.begin(), v.end(), [](const auto &a, const auto &b) {
        return a > b;
    });
}
```

Custom Comparators Showcase III

Following is an example of sorting a custom class of data.

Custom Comparators Showcase III

Following is an example of sorting a custom class of data.

```
class Job {
public:
    int timestamp; int jobID; vector<int> requests;
};

int main() {
    vector<Job> jobs;
    std::sort(jobs.begin(), jobs.end(), [](const auto &a, const auto &b) {
        return a.timestamp < b.timestamp; // sort according to timestamp
    });
}
```

Note

This comparator should return true for argument (a, b) if and only if a sits left of b in the sorted array.

List container

- A templated doubly linked list, this includes functions such as `push_front` `push_back`, `insert`, `erase`, etc.