Introduction to STL

theroyakash

Indian Institute of Technology Madras

August 20, 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.

std::lower_bound and std::upper_bound

- We need a non-decreasingly sorted container,
- We want to find out position of the smallest number just $>$ (greater) a given number or position of the smallest number $\geq$ (greater than or equal to) a given number

std::lower_bound and std::upper_bound

- `std::lower_bound` returns iterator to first element in the given range which is equal or greater than the value.
- `std::upper_bound` returns iterator to first element in the given range which is greater than the value.

List container

- List is a doubly linked list, this includes functions such as `push_front`, `push_back`, `insert`, `erase`, etc.

List container

- List is a doubly linked list, this includes functions such as `push_front`, `push_back`, `insert`, `erase`, etc.
- However I'd say not to use `std::list` as a data structure because difficult to manipulate.

Few Example

Following is the usage of `front()`, `back()` on a list. This runs in $O(1)$ time.

Few Example

Following is the usage of `front()`, `back()` on a list. This runs in $O(1)$ time.

```
#include <list>
#include <iostream>

int main() {
    std::list<char> letters {'d', 'm', 'g', 'w', 't', 'f'};

    if (!letters.empty()) {
        std::cout << "The first character is '" << letters.front() << "'.\n"
    ;
        std::cout << "The last character is '" << letters.back() << "'.\n";
    }
}
```

Singly Linked List

- Similarly there is singly linked list called `std::forward_list`.

Singly Linked List

- Similarly there is singly linked list called `std::forward_list`.
- It is not recommended to use these list data structures as you have limited control on the pointers.

Example Usage

Finding Middle of linked list

```
ListNode *middleNode(ListNode *head) {

    if (!head->next) {
        return head;
    }

    ListNode *slowPointer = head;
    ListNode *fastPointer = head;

    while (fastPointer != NULL && fastPointer->next != NULL) {
        slowPointer = slowPointer->next;
        fastPointer = fastPointer->next->next;
    }

    return slowPointer;
}
```

Example Usage on STL List

Finding Middle of STL Forward List

```
template <class T>
T findMiddleElement(forward_list<T> *list) {

    // Using 2 pointer approach
    auto slowPointer = list->begin();
    auto fastPointer = list->begin();
```

Example Usage on STL List

```
    // Update the slowPointer slowly and fastPointer quickly

    while (fastPointer != list->end() &&
           std::next(fastPointer, 1) != list->end()) {

        std::advance(slowPointer, 1);
        std::advance(fastPointer, 2);
    }

    return *slowPointer;

}
```

Map and Set

- Associative containers (Map and Set) are not sequential; they use keys to access data.

Map and Set

- Associative containers (Map and Set) are not sequential; they use keys to access data.
- Both Maps and Sets are internally implemented using trees,

Map and Set

- Associative containers (Map and Set) are not sequential; they use keys to access data.
- Both Maps and Sets are internally implemented using trees,
- Allows for efficient searching, insertion and deletion

Set Example

- Stores unique values

Set Example

- Stores unique values
- Interface functions: `insert`, `erase`, `clear`, `find`, `upper_bound`, `lower_bound`.

Example Usage

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

int main () {
    set<int> marks;
    marks.insert(10); marks.insert(20); marks.insert(30);
    marks.insert(100); marks.insert(100);

    for (auto elem : marks) { cout << elem << " "; }

    return 0;
}
```

Example Usage

Storing from Larger to Smaller

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

int main () {
    set<int, greater<int>> marks;
    marks.insert(10); marks.insert(20); marks.insert(30);
    marks.insert(100); marks.insert(100);

    for (auto elem : marks) { cout << elem << " "; }

    return 0;
}
```

std::multiset

Multiset works same as a set but allows duplicate values.

```cpp
#include <set>

int main () {
    multiset<int, greater<int>> marks;
    marks.insert(100); marks.insert(100); marks.insert(100);
    marks.insert(20); marks.insert(30);

    // 100 100 100 30 20
    for(auto e : marks) cout << e << " ";

    return 0;
}
```

`std::map`

To store Key-Value pairs you have two options `std::map` and `std::unordered_map`.

`std::map`

In `std::map` when you access the elements you get the values in sorted order.

std::map

- Each key is unique

std::map

- Each key is unique
- Internally implemented using `Red-Black Trees`

std::map

- Each key is unique
- Internally implemented using `Red-Black Trees`
- Interface contains functions such as `find`, `count`, `clear`, `erase`, etc,
- Also supports array-like indexing with keys.

std::unordered_map

- Works same as the map, but has better time complexity when access, find, and erase is called.

std::unordered_map

- Works same as the map, but has better time complexity when access, find, and erase is called.
- Each of the operations on `std::map` is $O(\lg n)$ but on `std::unordered_map` each access, find, erase is amortized $O(1)$.

Amortization

- With `C++11` we got hash set and hash map in the form of `std::unordered_set` and `std::unordered_map`

Amortization

- With `C++11` we got hash set and hash map in the form of `std::unordered_set` and `std::unordered_map`
- By amortized $O(1)$ insertion time we mean that each item only runs into $O(1)$ collisions on average,

Amortization

- With `C++11` we got hash set and hash map in the form of `std::unordered_set` and `std::unordered_map`
- By amortized $O(1)$ insertion time we mean that each item only runs into $O(1)$ collisions on average, This means there exists a sequence of elements for which `std::unordered_map` has collisions in every insertion.

Other hash functions

The **builtin** hash function for C++ is not optimal. There exists far better hash functions one such example is `SplitMix64`.

SplitMix64

Example Usage

```
struct SplitMix64 {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().
    time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
```

SplitMix64

Redefinition With `SplitMix64`

Now you define the `unordered_map` as `unordered_map<int, int, SplitMix64> unmap;`