C++ STL

December 2023

1 Arrays and Vectors

Arrays in C++

Declaration and Initialization:

```cpp
// Declaration of an array
int myArray[5];

// Initialization of an array
int anotherArray[] = {1, 2, 3, 4, 5};
```

Accessing Elements:

```cpp
int value = myArray[2]; // Accessing the third element (index 2) of
     the array
```

Size of Array:

```cpp
int size = sizeof(myArray) / sizeof(myArray[0]); // Size of the
    array
```

Iterating through an Array:

```cpp
for (int i = 0; i < size; ++i) {
    // Access and use myArray[i]
}
```

Vectors in C++ (STL)

Include Header:

```cpp
#include <vector>
using namespace std;
```

Declaration and Initialization:

```
// Declaration of a vector
vector<int> myVector;

// Initialization of a vector
vector<int> anotherVector = {1, 2, 3, 4, 5};
```

Accessing Elements:

```
int value = myVector[2]; // Accessing the third element (index 2) of the vector
int frontValue = myVector.front(); // Accessing the first element
int backValue = myVector.back();   // Accessing the last element
```

Size-related Functions:

```
int size = myVector.size();      // Number of elements in the vector
bool isEmpty = myVector.empty(); // Check if the vector is empty
```

Iterating through a Vector:

```
for (int i = 0; i < myVector.size(); ++i) {
    // Access and use myVector[i]
}

// or using range-based for loop (C++11 and later)
for (int element : myVector) {
    // Access and use element
}
```

Adding Elements:

```
myVector.push_back(6); // Adds an element to the end of the vector
```

Other Vector Operations:

```
myVector.pop_back();  // Removes the last element from the vector
myVector.clear();     // Removes all elements from the vector
myVector.insert(myVector.begin() + 2, 10); // Inserts 10 at position 2
myVector.erase(myVector.begin() + 3);      // Removes element at position 3

// Copy constructor
vector<int> copyVector(anotherVector);

```

```cpp
// Assignment operator
myVector = anotherVector;

// Swapping contents of two vectors
myVector.swap(anotherVector);
// or
swap(myVector, anotherVector);
```

2D Vectors in C++

Declaration and Initialization:

```cpp
// Declaration of a 2D vector
vector<vector<int>> my2DVector;

// Initialization of a 2D vector
vector<vector<int>> another2DVector = {{1, 2, 3}, {4, 5, 6}, {7, 8,
     9}};
```

Accessing Elements:

```cpp
int value = my2DVector[1][2]; // Accessing the element at row 1,
    column 2
```

Size-related Functions:

```cpp
int numRows = my2DVector.size();    // Number of rows in the 2D
    vector
int numCols = my2DVector[0].size(); // Number of columns in the 2D
    vector
```

Iterating through a 2D Vector:

```cpp
for (int i = 0; i < numRows; ++i) {
    for (int j = 0; j < numCols; ++j) {
        // Access and use my2DVector[i][j]
    }
}
```

2 Maps and Unordered Maps

2.1 Maps

Definition

A map is a data structure that stores elements in key-value pairs. It's often referred to as a dictionary in Python.

Characteristics

- Each key in a map is unique.
- In map elements are stored in a sorted order based on their keys.
- It's typically implemented using a self-balancing binary search tree like a Red-Black Tree.

Operations

- **Insertion :** $map[key] = value$; or map.insert(std::make_pair(key, value));
- **Access :** $value = map[key]$;
- **Search :** Check if a key exists using $map.find(key) != map.end()$.
- **Deletion :** Remove a key-value pair with $map.erase(key)$.

Implementation

```cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    // Declare a map
    std::map<int, std::string> map;

    // Insertion
    map[1] = "Alice";
    map.insert(std::make_pair(2, "Bob"));

    // Access
    std::string name1 = map[1];
    std::cout << "Name with key 1: " << name1 << std::endl;

    // Search
    if (map.find(2) != map.end()) {
        std::cout << "Found key 2 with value: " << map[2] << std::
    endl;
    } else {
        std::cout << "Key 2 not found." << std::endl;
    }

    // Deletion
    map.erase(1);

    // Check after deletion
    if (map.find(1) == map.end()) {
```

```
        std::cout << "Key 1 deleted." << std::endl;
    }

    return 0;
}
```

2.2 Unordered maps

Definition

Unordered map stores key-value pairs but in no particular order.

Characteristics

- It's implemented using a hash table, which allows for faster access on average compared to a map.
- Each key is still unique.
- Offers more efficient insertion, search, and deletion compared to a map, except in certain cases of hash collisions.

Operations

- **Insertion :** $umap[key] = value$; or $umap.insert(std :: make_pair(key, value))$;
- **Access :** $value = umap[key]$;
- **Search :** Similar to map, using $umap.find(key) != umap.end()$.
- **Deletion :** Similar to map, using $map.erase(key)$.

Operations

```
#include <iostream>
#include <unordered_map>
#include <string>

int main() {
    // Declare an unordered map
    std::unordered_map<int, std::string> umap;

    // Insertion
    umap[3] = "Charlie";
    umap.insert(std::make_pair(4, "David"));

    // Access
    std::string name2 = umap[3];
    std::cout << "Name with key 3: " << name2 << std::endl;
```

```
16
17      // Search
18      if (umap.find(4) != umap.end()) {
19          std::cout << "Found key 4 with value: " << umap[4] << std::
        endl;
20      } else {
21          std::cout << "Key 4 not found." << std::endl;
22      }
23
24      // Deletion
25      umap.erase(3);
26
27      // Check after deletion
28      if (umap.find(3) == umap.end()) {
29          std::cout << "Key 3 deleted." << std::endl;
30      }
31
32      return 0;
33  }
```

3 Sets and Multisets

The values are stored in a specific sorted order i.e. either ascending or descending. Each element in a set is unique, Incase of a multiset the values can be multiple, it is defined inside the $< set >$ header file.

```
1 #include <set>
2 //set <datatype> setname(Declaration)
3 set <int> st; // defining an empty set
4 set <int> val = {6, 10, 5, 1}; // defining a set with values\\
5 set<int,greater<int>> st;//For descending order
6 //We can also use a custom comparator
```

Frequently used functions for set:

Function	Use	Syntax
insert()/emplace()	It inserts an element in the set if it is not present in the set.	setname.insert()
erase()	It removes the element specified from the set. It will give an error if the element is not present in the set.	setname.erase()
find()	It returns an iterator which points to the element we want to find. If the element is not present it will point after the last element in the set.	auto iterator= setname.find()
count()	For a set it is used to check whether an element is present or not. For a multiset it can be used to check the frequency of the particular value	int variable_name= st.count()
Lower_bound()	It will return iterator to smallest element>=to the value given. If not present it return iterator pointing to the end of the set.	auto iterator= st.lower_bound(value)
Upper_bound()	It will return iterator to smallest element>to the value given If not present it will return iterator pointing to the end of the set.	auto iterator= st.upper_bound(value)

Functions 'begin()', 'end()', 'size()', 'clear()' remain the same as that of vector.

```
set<int> st;//Creating a set of integers
st.insert(1);//{1}
st.insert(7);//{1 7}
st.insert(4);//{1 4 7}
st.insert(3);//{1 3 4 7}
st.insert(1);//{1 3 4 7}

st.erase(4);//{1 3 7}
st.erase(5);// It will give error as 5 is not present
//To tackle this error
if(st.find(5)!=st.end())
    st.erase(5)
auto it=st.find(7);//It returns iterator pointing to 7
auto it=st.find(11);//Since 11 is not present it will point to the
    end of the set(after the last element)

//To erase multiple elements
st.erase(iterator1,iterator2)//Erases elements starting from the
    first iterator till the second iterator
For eg:
auto it1=st.find(1)
auto it2=st.find(7)
st.erase(it1,it2);//{7}

 Now coming on to multiset
multiset<int> mt;
```

```
mt.insert(1);//{1}
mt.insert(7);//{1 7}
mt.insert(4);//{1 4 7}
mt.insert(1);//{1 1 4 7}
mt.insert(1);//{1 1 1 4 7}

int count=mt.count(1)//count=3 over here
// All other things remain the same for multiset.
// If we want to erase all occurrences of a particular value:
mt.erase(1);//{4 7}

// If we want to erase 1 occurrence of a particular value:
mt.erase(mt.find(1));//{1 1 4 7}

// If we want to erase multiple occurrences of a particular value:
mt.erase(mt.find(1), mt.find(1)+2);//{1 4 7}, two occurrences
    removed in this case.
// Printing all the elements of a set
for (auto it = st.begin(); it != st.end(); ++it)
    cout << ' ' << *it;
```

4 Pairs, Tuples

4.1 Pairs

Pairs are a helpful tool in case we want to bind two values together or in other sense a element that is represented by a pair of information.

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    // pair<data_type1, data_type2> pair_name
    pair<int,int> p1 = {1,1};
    pair<int,string> p2 = {1,"Hello"};

    // use of make_pair to assign values;
    pair<int,int> p3;
    p3 = make_pair(2,2);

    // Referencing elements of a pair
    // pair_name.first gives the first element of the pair and
    pair_name.second gives the second element
    auto first_el = p1.first;
    auto second_el = p1.second;

    //output the elements separately
    cout << p2.first << " " << p2.second << endl;
}
```

4.2 Tuples

Tuple is what you would say an extension of pair in case the number of features needed to describe one element is more than 2.
Two basic operation of tuples:

1. get() : uses index and tuple name as an input and help to get and update value in tuples.
2. make_tuple() : used to assign the tuple with values. The values should be in the order they are initialised in tuple.

```cpp
// C++ code to demonstrate tuple, get() and make_tuple()
#include<iostream>
#include<tuple> // for tuple
using namespace std;
int main()
{
  // Declaring tuple
  tuple <char, int, float> geek;

  // Assigning values to tuple using make_tuple()
  geek = make_tuple('a', 10, 15.5);

  // Printing initial tuple values using get()
  cout << "The initial values of tuple are : ";
  cout << get<0>(geek) << " " << get<1>(geek);
  cout << " " << get<2>(geek) << endl;

  // Use of get() to change values of tuple
  get<0>(geek) = 'b';
  get<2>(geek) = 20.5;

  // Printing modified tuple values
  cout << "The modified values of tuple are : ";
  cout << get<0>(geek) << " " << get<1>(geek);
  cout << " " << get<2>(geek) << endl;

  return 0;
}
```

5 Common Functions

5.1 Sort

Sorting is one of the fundamental algorithms when it comes to programming. There are various sorting algorithms, but the C++ STL comes in handy by providing an algorithm abstraction. There is a

concept of a custom sorting function for structs, which can be found here. the sort function takes input l and r, it sorts the numbers in range $[l, r)$.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> vec = {1, 4, 3, 2, 5};

    // printing the original vector
    for(int i=0; i<vec.size(); i++)
        cout << vec[i] << " ";
    cout << endl;

    // sorting the vector
    sort(vec.begin(), vec.end());

    // printing the sorted vector, originally ascending
    for(int i=0; i<vec.size(); i++)
        cout << vec[i] << " ";
    cout << endl;
    // 1, 2, 3, 4, 5

    // sorting in descending order
    sort(vec.begin(), vec.end(), greater<int>());

    // sorted in descending order
    for(int i=0; i<vec.size(); i++)
        cout << vec[i] << " ";
    cout << endl;
    // 5, 4, 3, 2, 1

    return 0;
}
```

5.2 Reverse

Reverse is also a fundamental algorithm. the reverse function in STL takes two inputs l and r and reverses the numbers in range $[l, r)$

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> vec = {1, 4, 3, 2, 5};

    // printing the original vector
    for(int i=0; i<vec.size(); i++)
        cout << vec[i] << " ";
    cout << endl;
```

```
12
13      // reversing the vector
14      reverse(vec.begin(), vec.end());
15
16      // printing the reverse vector
17      for(int i=0; i<vec.size(); i++)
18          cout << vec[i] << " ";
19      cout << endl;
20      // 5, 2, 3, 4, 1
21
22      return 0;
23  }
```

5.3 Accumulate

This is like the reduce function in Python. this function is best explained using examples. look at the code to understand.

```
1   #include <bits/stdc++.h>
2   using namespace std;
3
4   int my_func(int x, int y)
5   {
6       return x * y;
7   }
8
9   int main()
10  {
11      vector<int> vec = {1, 4, 3, 2, 5};
12
13      // default function is summation
14      int sum = accumulate(vec.begin(), vec.end(), 0);
15      cout << sum << endl; // 15
16
17      // passing a custom function
18      int product = accumulate(vec.begin(), vec.end(), 1, my_func);
19      cout << product << endl; // 120
20
21      /*
22      what is happening is
23      res = 1
24      for element in the range [1, r)
25          res = my_func(res, element)
26      return res
27      */
28
29      return 0;
30  }
```

5.4 Some Other Functions

There are some other functions like count, partial sum, rotate, and many more, but you can find out about them in the documentation

here. They are functions which are quite simple to implement. We generally only use the sort and reverse functions in the STL.

6 Lower Bound and Upper Bound in C++

6.1 Lower Bound

`lower_bound` finds the position of the first element not less than a given value within a sorted sequence.

6.2 Upper Bound

`upper_bound` finds the position of the first element greater than a given value within a sorted sequence.

6.3 Time Complexity

Both `lower_bound` and `upper_bound` functions have a time complexity of $O(\log n)$, where n is the number of elements in the range. They perform a binary search on the sorted range to locate the desired position efficiently.

6.4 C++ Code Example

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec = {10, 20, 30, 40, 50};

    // Using lower_bound and upper_bound
    auto lb = std::lower_bound(vec.begin(), vec.end(), 25); // Returns iterator to the first element not less than 25
    auto ub = std::upper_bound(vec.begin(), vec.end(), 35); // Returns iterator to the first element greater than 35

    std::cout << "Lower Bound at index: " << std::distance(vec.begin(), lb) << std::endl;
    std::cout << "Upper Bound at index: " << std::distance(vec.begin(), ub) << std::endl;

    return 0;
}
```