

Containers

In programming, a **container** refers to a data structure that holds a collection of elements or objects of a certain type. It provides operations to add, remove, and access the elements stored within it.

Containers can be found in many programming languages and are used in a wide range of applications. They are particularly useful when dealing with large amounts of data that need to be organized and manipulated efficiently.

In C++, the **Standard Template Library** (STL) provides a collection of container classes that are widely used. These include **vectors**, **arrays**, **lists**, **maps**, and **sets**, among others. Each container class provides a specific set of operations and is designed to be used in different situations depending on the specific requirements of the program.

Vector

In C++, a **vector** is a dynamic array that can resize itself automatically when elements are added or removed. It is a **container** that allows you to store and manipulate a sequence of elements of the same data type.

Vectors provide the following benefits over traditional arrays:

1. They can grow or shrink in size dynamically as elements are added or removed.
2. They provide bounds checking to ensure that you don't access elements that are out of range.
3. They can be easily resized and copied.

NOTE

- **vector** can be found in <vector> header file
- Key methods of vector are:
push_back(), pop_back(), size(), front(), back()

Declaration of a vector container

1-Dimensional Vector

E.g `vector<int> myvector;`

E.g `vector<int> myvector(size, val);`

// val : value supplied to all cells, size: Fixed size of vector

E.g `vector<int> myvector{1,2,3,4,5};`

E.g `vector<string> myvector{"Apple", "Banana", "Orange", "Grapes", "Strawberry"};`

2-Dimensional Vector

E.g. `vector<vector<int>> myvector;`

E.g. `vector<vector<int>> myvector {
 {1,2,3}, {4,5,6}, {7,8,9}
 };`

E.g. `vector<vector<int>> matrix(m, vector<int>(n, val));`

// val: value supplied to all cells, m: Rows in vector, n: Columns in vector

Traversing in a vector E.g 1

```
for(int i=0; i<myvector.size();i++){  
    cout<<myvector[i];  
}
```

Traversing in a vector E.g 2

```
for(auto x : myvector){  
    cout<<x;  
}
```

Useful Methods related to vector

size()- Returns the Number of Elements in vector

For 2D Vectors

`myvector.size()` - Returns No. of Rows

`myvector[0].size()` - Returns No. of Columns

max_size() - Returns the Number of Elements a vector can hold
capacity() - Returns the size of the vector which is currently allocated to it, Expressed as a number of Elements

push_back() - Push New Supplied Element into the vector/**string** from the back

pop_back() - Remove Element from the vector/**string** from the back

front() - Get the first Element from the vector/**string**

back() - Get the End Element from the vector/**string**

begin() - Returns an Iterator Pointing to the first Element of the vector

end() - Returns an Iterator Pointing to The Theoretical Element Present Next to the Last Element of the Vector

NOTE

- end() Iterator does not point to any Element Present in the vector

find() - Finds the Element in the given range. Returns Iterator to the First Element if Found, Otherwise Returns the last.

Sample:

```
vector<int> v{1,2,3,4,5};
```

```
vector<int>:: iterator it;// An Iterator stores addresses of vector elements and does not work with primitive types
```

```
int search = 4; // Element to be searched in vector
```

```
it = find(v.begin(), v.end(), search);
```

```
it == v.end() - If element is not found in vector/string
```

Output:

```
*it - 4 // On Successful Search, find will return the address of the first Element Found which actual value can be seen using * before iterator just like a pointer.
```

```
it - v.begin() - 3 // Difference between the Pointers
```

Find Method Key Findings

- We can **subtract (-)** iterators to find the distance between 2 iterators produced by the find method.

E.g

```
vector<int> cont = {1, 2, 2, 3, 4, 5};  
vector<int>::vector it1 = find(v.begin(), v.end(), 2);  
vector<int>::vector it2 = find(v.begin(), v.end(), 3);  
// Here 1st occurrence of 2 is at index 1 and 3 is at index 4  
cout << (it - it2); // -2  
cout << (it2 - it1); // 2  
cout << abs(it1 - it2); // 2
```

sort()- Sorts the vector either in Ascending or Descending Order.

Assume the Sample vector be v :

*The syntax for **Ascending** order sort: sort(v.begin(), v.end())*

*The syntax for **Descending** order sort: sort(v.begin(), v.end(), greater<T>())*

NOTE: Both vector and string can be sorted with sort()

insert() : The insert() function is used to insert an element at a specific position (**iterator required**) in a vector/**string**. It shifts all the elements to the right of the position by one position to make room for the new element, and inserts the new element at the specified position. However, if you insert an element at the beginning of the vector or at the end, you do not need to shift any elements, since there are no elements on the left or right to be shifted.

E.g : Assume a vector/**string** v, **iterator** for position, **value** to insert

Syntax: v.insert(v.begin() + index, value)

reverse() : the reverse() function from the algorithm header is used to reverse the characters in the string s. The begin() and end() member functions of the string class are used to get the iterators to the start and end of the string. Once the reverse() function is called with these iterators as arguments, the string s is reversed in-place.

E.g : `string s = "hello world";`
`reverse(s.begin(), s.end());`

erase() : To erase an element from a vector/**string** in C++, you can use the erase method, which **takes an iterator to the element to be erased** and not an index.

Here's an example that erases an element from a vector :

```
vector<int> v = {1, 2, 3, 4, 5};  
v.erase(v.begin() + 2); // 1 2 4 5  
string str = "abcde";  
str.erase(v.begin() + 2); // a b d e
```

NOTE

- Avoid using it for large string/vector as memory runtime error may occur

clear(): The clear() function is used to remove all the elements of the vector container, thus making it size 0.

Syntax:

`vector_name.clear()`

Parameters: No parameters are passed.

Result: All the elements of the vector are removed (or **destroyed**).

Input:

```
myvector= {1, 2, 3, 4, 5};  
myvector.clear();
```

Output:

```
myvector= { }
```

numeric_limits<T>::max() : This is a template function in the C++ standard library that returns the maximum value of a given numeric type T.

This function is defined in the limits header file and can be used for various data types, including integers, floating-point numbers, and even some user-defined types.

For instance, if we call `numeric_limits<int>::max()`, it will return the largest value that can be represented by an int on the current platform. Similarly, calling `numeric_limits<double>::max()` will return the maximum finite value that a double can represent.

The `max()` function is particularly useful when we need to compare values of a numeric type, or when we want to initialize a variable with the largest possible value. By using `numeric_limits<T>::max()` we can avoid hardcoding the maximum value for a given type, which can make our code more portable and less error-prone.

count(): In C++ STL, `count()` is a function that is used to count the number of occurrences of a specific element in a given range of elements.

Here are some key points to keep in mind about the `count()` method:

1. `count()` is defined in the **<algorithm>** header file in C++ STL.
2. It takes two iterators as its arguments: the first iterator represents the start of the range, and the second iterator represents the end of the range.
3. It also takes the value of the element to be counted as a third argument.
4. The function returns the number of occurrences of the element in the given range.
5. The range of elements must be sorted in order for `count()` to work properly. If the range is not sorted, the behavior of `count()` is undefined.

6. The type of the value being counted must be comparable using the == operator, or an equivalent custom comparison function must be provided.

Here's an example usage of count():

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

```
using namespace std;
int main() {
// Usage of count() on Containers : Vector, Arrays
    vector<int> v = {1, 2, 3, 4, 2, 2, 5};
    int count = count(v.begin(), v.end(), 2);
    cout << "Number of occurrences of 2: " << count << endl; // 3

// Usage of count() on string
    string str = "the quick brown fox jumps over the lazy dog";
    char ch = 't';
    int count = count(str.begin(), str.end(), ch);
    cout << "Number of occurrences of \" " << ch << "\": "
    << count << endl; // 2

    return 0;
}
```

Count Method Key Findings:

- We can only use the count method to count **characters** and **digits**, and not **string** to find substrings.
- The Following is the wrong use-case of **count** method:
string str = "the quick brown fox jumps over the lazy dog";
string substr = "the";
int count = count(str.begin(), str.end(), substr); // Error

String Transformation

Consider a **string** `s` to be transformed -

String Lowercase to Uppercase :

```
transform(s.begin(), s.end(), s.begin(), ::toupper);
```

String Uppercase to Lowercase :

```
transform(s.begin(), s.end(), s.begin(), ::tolower);
```

NOTE

For Character (**char**) Transformation :

toupper(char) - Turn char to upper-case letter

tolower(char) - Turn char to lower-case letter

Swapping two values

Consider a string `s1 = "ABC";`

Sample Swapping E.g :

```
swap(s1[x],s1[y]);      // STL way of Swapping Two Values
```

NOTE

- The Swapping Method Described Above can be used for Integer Array as well
- Swap method comes under `<utility>` header file

Converting string to integer

To convert a string to a digit in C++ STL, you can use the `stoi()` function.

Here's an example:

```
string str = "123";
```

```
int num = stoi(str);
```

NOTE

- `stoi()` method comes under `<string>` header file
- If the input string is not a valid integer, `stoi()` will throw an `invalid_argument` exception.

To handle this, you can use a try-catch block:

```
#include <iostream>
#include <string>
#include <stdexcept>
```

```
using namespace std;
```

```
int main() {
    string str = "abc";
    try {
        int num = stoi(str);
        cout << "String: " << str << endl;
        cout << "Number: " << num << endl;
    } catch (const invalid_argument& e) {
        cout << "Error: " << e.what() << endl;
    }
    return 0;
}
```

Reverse a String

Consider a String s1= "Hello";

*Reversing Method (STL Usage) : **reverse**(s1.begin(), s1.end());*

Reversing Method (Two Pointer Approach) :

```
void reverse(string &s1, int len, int i){
    if(i >= len)
        return;

    swap(s1[i],s1[len]);
    reverse(s1, len-1;i+1);
}
```

Comparing Two Strings

Consider Two String s1 and s2, s1 = "Hello", s2 = "World"

Syntax to Compare s1 and s2:

s1.compare(s2); OR s2.compare(s1);

- If the result is 0, Then strings are Matched

Truncate a String using substr()

string.substr(x, y): The substr() function returns a new string that is a substring of the original string.

Syntax: string substr (size_t pos, size_t len) const;

Parameters:

pos: Position of the first character to be copied.

len: Length of the sub-string.

size_t: It is an unsigned integral type.

Here's an example code that truncates a string into two halves:

```
string str = "Hello, World!";  
int len = str.length();  
int midpoint = len / 2;  
string first_half = str.substr(0, midpoint);  
// here midpoint is length of substring  
string second_half = str.substr(midpoint);
```

Convert integer (int) to string

to_string(int) - This method can be used to convert int to string by passing int to it directly or through a variable.

Verify Whether a Character is Alphanumeric

isalnum() - Method that returns non-zero if the ASCII value passed in has a character equivalent to a letter or number else returns zero.

Prototype: int alnum(int ch);

Header File: ctype (C++), ctype.h (C)

Find Min/Max Element Between Two Values

min(value1, value2) - Returns Element whichever is minimum
max(value1, value2) - Returns Element whichever is maximum

Obtain the Possible min/max value of INT

INT_MIN - Returns minimum integer value
INT_MAX - Returns maximum integer value

istringstream()

istringstream is a class in the C++ Standard Template Library (STL) that allows for easy input parsing and conversion of strings into other data types. It is included in the **<sstream>** header file.

The **istringstream** class takes a string as its input and provides methods to extract individual tokens from the string separated by blank-space. This is useful for **parsing** text files or data from a network connection.

Some of the key features of istringstream include:

1. **>> operator**: The >> operator can be used to extract individual tokens from the string. For example, **iss >> num** will extract the next token from the string and store it in the variable num.
2. **getline method**: The **getline** method can be used to extract an entire line of text from the string.
3. **Conversion functions**: The istringstream class provides several built-in conversion functions, such as **stoi** and **stod**, which can be used to convert strings to **integers** or **doubles**, respectively.

Here's an example of how to use `istringstream` to parse a string:

```
#include <sstream>
#include <string>
#include <iostream>

using namespace std;
int main() {
    std::string input = "42 3.14 hello world";
    std::istringstream iss(input);

    int num;
    double dnum;
    std::string str;
    std::string str2;
    std::string str3;

    iss >> num >> dnum >> str >> str2 >> str3;

    std::cout << "num = " << num << std::endl;    // 42
    std::cout << "dnum = " << dnum << std::endl; // 3.14
    std::cout << "str = " << str << std::endl;      // hello
    std::cout << "str2 = " << str2 << std::endl;    // world
    std::cout << "str3 = " << str3 << std::endl;    //
    return 0;
}
```

Check whether the char (character) is a digit

isdigit(char) - we can use this method to check whether the **char** we have is an equivalent digit or not.

For example (consider char and a string)

```
char x = '1';
string str = 'a1';
cout << isdigit(x); // true
cout << isdigit(str[0]); // false
cout << isdigit(str[1]); // true
```

NOTE

- We can only supply **char** variable to `isdigit()`, and not a **string**
- Declared in `<ctype.h>` header file

Check Datatype of a variable

typeid(variable).name() : returns type of the supplied value or variable.

Sample use-case examples:

```
string x = "abc";  
int i = 1;  
char c = 'a';  
cout << typeid(x[0]).name() << endl;    // c  
cout << typeid(x).name() << endl;    // ... string  
cout << typeid(c).name() << endl;    // c  
cout << typeid(i).name() << endl;    // i
```

NOTE

- It can be used in debugging purposes
- Defined in `<typeinfo>` header file

Find the absolute value / mod of a difference

fabs() : `fabs` can be used to calculate absolute value of **floating integer types** and is defined in `<math.h>` header file.

abs() : `abs` can be used to calculate absolute value of **integer types** and is defined in `<stdlib.h>` header file

Key Findings of `abs()`

- `abs` will not work properly/produce errors if you do something like this when computing absolute length of strings.
string x = "abc", y = "defgh";
`abs(x.length() - y.length());` // Error
- Instead, take length outside of `abs` and then directly supply int values, check the next page for suggestions on the same.

Example valid use-case:

<code>int a = 5, b = 7;</code>	<code>string a = "abc", b = "defgh";</code>
<code>abs(a - b); // 2</code>	<code>int s1 = a.length();</code>
<code>abs(b - a); // 2</code>	<code>int s2 = b.length();</code>
	<code>abs(s1 - s2); // 2</code>
	<code>abs(s2 - s1); // 2</code>

String Concatenation

There are many methods to concatenate two strings and here are some of the key methods:

1. + operator

E.g `string a = "ab", b = "cd";`
`cout << a + b; // abcd`

2. append()

E.g `string a = "ab", b="cd"`
`a.append(b); // abcd`

+ Operator Key Findings

- The + Operator will not work if concatenation is done in this fashion
 1. `string s = "" + 'a'; // Error`
 2. `string s = "abc";`
`string x = "" + s.back(); // Error`
- Try Instead,
`string s = "abc";`
`string x = "";`
`x = x + s.back();`
- In above statements we are trying to convert a **char** type variable to **string** type variable ('a' and `s.back()` are both **char** type variable)

Converting a single digit character (char) to its integer (int) equivalent

Suppose we have a **single digit** represented as a **character (0-9)**. When we try to print the character '0', it is actually represented as the **ASCII** value **48**. We want to obtain the integer equivalent of the digit, i.e. 0 for '0', 1 for '1', 2 for '2', and so on up to 9.





To achieve this, we can use a simple trick:

```
char ch = '5'; // any number
int i = ch - '0'; // implicit type casting occurs here
```

NOTE

- Subtracting the ASCII value of '0' from the character gives us the corresponding integer value of the digit. Here, an implicit type-casting occurs, which converts the character to its corresponding integer value.
- It is important to note that **explicit type-casting**, such as **(int)ch**, will not work to convert a single digit represented as a character to its integer value. This is because type-casting converts the ASCII value of the character to an integer, which will not yield the desired result.

Essential ASCII values

- '0' - '9' : 48 - 57 { 48 = '0', 49 = '1', ... 57 = '9' }
- ' ' (blank-space) : 32
- a - z : 97 - 122 { 97 = a, 98 = b, ... 122 = z }
- A - Z : 65 - 90 { 65 = A, 66 = B, ... 90 = Z }
-  : 128 - 255 { 128 =  , 129 =  , ... 255 =  }

NOTE

- ASCII values are in the range **0 - 255**. After 255, values starts repeating i.e { 256 == 0, 257 == 1, ... A == 321, ... a == 353, ... }
- **Printable characters** are represented by ASCII values in the range of **0-127**, while **Non-Printable characters** are represented by ASCII values in the range of **128-255**.

Vector to String

```
vector<int> cont = {1, 2, 3, 4, 5};  
string str = string(cont.begin(), cont.end());  
for(auto x : str)  
    cout << x; // 1 2 3 4 5
```

NOTE

- **string(cont.begin(), cont.end())** is a constructor call to create a new **string** object from a **pair of iterators** cont.begin() and cont.end()

Length of a string and Arithmetic Operations

sizeof(str)/sizeof(char) : It is classic way to get the length of a string/array in C/C++, defined in the form of a **character array** char str [] = "i am a string \0"

string.length() : length() method is used to retrieve the current length of the **string** and is defined in **<string>** header (C++ Only)

NOTE

- It is **not advisable** to perform **arithmetic operations** directly when computing the length of a string. Instead, it is recommended to store the length in a variable and then use that variable for any further arithmetic operations.
- For example, rather than computing (string.length() - k), save the length in a variable and then perform the necessary arithmetic using the variable.

Stack

Declaration e.g :

```
stack<int> myStack;  
stack<char> myStack;
```

Push the item to the top of the stack :

```
E.g. myStack.push(5);  
      myStack.push('A');
```

Pop the item from the stack (no-return of pop value) :

```
E.g. myStack.pop();
```

Peek the item at the top of the stack (returns pop value) :

```
E.g. myStack.top();
```

Check Whether the Stack is empty or not :

```
E.g. myStack.empty();
```

- Returns true if Stack is empty else returns false

Check the size of the stack:

```
E.g. myStack.size();
```

- Returns the number of elements present in stack

NOTE

- **size()** may return value of type **size_t** that's why **type-casting** size like **(int)myStack.size()** will be a more safe choice.

Linked Lists

In C++ STL, Linked Lists can be implemented using **List** containers. There are two main types of lists: **std::list** and **std::forward_list**.

List

list is a doubly-linked list that allows bidirectional traversal. It is defined in the <list> header file.

Here's an example code that demonstrates how to use an std::list:

```
#include <iostream>
```

```
#include <list>
```

```
int main() {
```

```
    // Create an empty list
```

```
    std::list<int> myList;
```

```
    // Add some elements to the list
```

```
    myList.push_back(10);
```

```
    myList.push_back(20);
```

```
    myList.push_back(30);
```

```
    // Insert an element at the beginning of the list
```

```
    myList.push_front(5);
```

```
    // Remove the last element from the list
```

```
    myList.pop_back();
```

```
    // Print the elements of the list using an iterator
```

```
    std::cout << "Elements of the list: ";
```

```
    for (auto it = myList.begin(); it != myList.end(); ++it) {
```

```
        std::cout << *it << " ";
```

```
    }
```

```
    std::cout << std::endl;
```

```
    return 0;
}
```

The `std::list` container provides the following operations:

push_front() and push_back(): Adds an element to the front or back of the list, respectively.

pop_front() and pop_back(): Removes the element from the front or back of the list, respectively.

insert(): Inserts an element at a given position in the list.

erase(): Removes an element from a given position in the list.

size(): Returns the number of elements in the list.

empty(): Returns whether the list is empty.

begin() and end(): Returns iterators to the beginning and end of the list, respectively.

Forward List

`forward_list` is a singly-linked list that allows only forward traversal. It is defined in the `<forward_list>` header file.

Here's an example code that demonstrates some of the operations that can be performed on a `std::forward_list`:

```
#include <iostream>
#include <forward_list>
```

```
int main() {
    // Create an empty forward list
    std::forward_list<int> myList;

    // Add some elements to the list
    myList.push_front(30);
    myList.push_front(20);
    myList.push_front(10);
}
```

```
// Print the elements of the list
std::cout << "Elements of the forward list: ";
for (auto it = myList.begin(); it != myList.end(); ++it) {
    std::cout << *it << " ";
}
std::cout << std::endl;

// Insert an element after the second element
auto it = myList.begin();
std::advance(it, 1); // Advance the iterator to the second element
myList.insert_after(it, 25);

// Remove the first element
myList.pop_front();

// Remove all elements with value 20
myList.remove(20);

// Remove consecutive duplicates
myList.unique();

// Sort the elements
myList.sort();

// Print the sorted elements of the list
std::cout << "Sorted elements of the forward list: ";
for (auto it = myList.begin(); it != myList.end(); ++it) {
    std::cout << *it << " ";
}
std::cout << std::endl;

return 0;
}
```

Here are some of the main operations that can be performed on a `std::forward_list`:

push_front(val): Inserts a new element with value `val` at the beginning of the list.

pop_front(): Removes the first element of the list.

insert_after(pos, val): Inserts a new element with value `val` after the element pointed to by `pos`.

erase_after(pos): Removes the element after the element pointed to by `pos`.

remove(val): Removes all elements with value `val` from the list.

unique(): Removes all consecutive duplicate elements from the list.

sort(): Sorts the elements of the list in ascending order.

NOTE

- Unlike `std::list`, `std::forward_list` does not provide a `push_back()` operation since it can only be traversed forward. However, `std::forward_list` does provide a `before_begin()` function which returns an iterator to the element before the first element of the list. This can be used to insert elements at the end of the list by calling `insert_after()` with the `before_begin()` iterator.
- Both types of lists provide similar operations, but `std::list` has more functionality since it supports bidirectional traversal. However, `std::forward_list` is more memory-efficient since it requires only one pointer per node (compared to two pointers in `std::list`), and it can be faster in some cases since it has fewer memory access operations.

Hash Map Using STL

“ Hash Map is a Data Structure which Provides Facility to Perform Operations like Insertion, Deletion, Traversal, etc in Constant Time “

- Hash Map is Like a table that maps a Unique Key to a Value.
- Hash Map can be created using :
 1. unordered_map<key-datatype, value-datatype>
 2. map<key-datatype, value-datatype>

Simply,

Hash Map == Unordered_map

Balanced BST == map

std::unordered_map and std::map are two commonly used container classes in the C++ Standard Library (STL) for implementing associative arrays.

Map

map is an implementation of a balanced binary search tree, also known as a Red-Black tree, that orders its elements based on their keys. This means that elements in a std::map are **stored in a sorted order based on their keys**, which allows for efficient searching and traversal of the elements. However, the time complexity for insertion and deletion operations is logarithmic, which means that they are **slower compared to the unordered version**.

Unordered_map

unordered_map, on the other hand, is an implementation of a hash table, where the **elements are stored in a random order based on their hash values**. The unordered nature of this container makes it **faster for insertion, deletion, and search operations**, with an average time complexity of $O(1)$. However, the **order of the elements is not guaranteed to be sorted**.

unordered_map<key-datatype, value-datatype>

Creation E.g.

```
unordered_map<string, int> table;
```

Insertion Type 1 E.g.

```
pair<string, int> p = make_pair("C++", 1);  
table.insert(p);
```

Insertion Type 2 E.g.

```
pair<string, int> p("C++", 1);  
table.insert(p);
```

Insertion Type 3 E.g. (Easy way)

```
table["C++"] = 1;
```

NOTE :

- *Re-Inserting to already available key (with/without value) will Overwrite the key value*

Searching Key and Fetching value Type 1 E.g.

```
table["C++"]
```

Searching Key and Fetching value Type 2 E.g.

```
table.at["C++"]
```

Searching Key (IF Unique Key Exist)

```
table.find(key) == table.end()
```

NOTE

- Here **find** method returns an iterator
- **Reminder:** The following snippet can be useful to remove duplicates from an array/string without affecting the relative ordering and maintaining a proper sequence **without** any **sort** method (Hash-map and arrays can be used to achieve this).

Obtain the size of the map E.g.

```
table.size()
```

Check The Presence of the key E.g.

```
table.count("C++")
```

Erase a key from the map E.g.

```
table.erase("C++")
```

Create a map Iterator E.g.

```
unordered_map<string, int> :: iterator it;
```

Traverse a map using Iterator E.g.

```
it = table.begin();
while(it != table.end()){
    cout<<"Key = "<<it->first<<" , Value = "<<it->second;
    it++;
}
```

NOTE :

- **it->first** - Provides Key
- **it->second** - Provides Key's - value

Unordered_map Key Findings:

1. When we declare unordered_map, all **keys** are automatically supplied **value** at beginning based on the **type** of **value** :
 - **char, int** - '\0' (equivalent to 0)
 - **string** - "" (empty string)

So in order to check whether we already got a value in a map or not, we can do something like this (suppose map as mp and key of type **int** being accessed is k):

```
If ( mp[k] == '\0' ) {
    // do something
}
```


- Here the if statement will run if map's specific key is not supplied a value apart from its initialization (beginning) step.
- We don't have to directly compare with zero as it can create conflict between determining actual value and NULL (not initialized) character.

Sets

In C++ STL (Standard Template Library), sets are a container that stores unique, ordered elements. A set is a collection of elements that are sorted in a specific order (usually ascending order) and have no repeated elements. Sets are implemented as balanced binary search trees (usually **Red-Black trees**) and have a logarithmic time complexity for insertion, deletion, and searching.

Sets in C++ STL are defined in the `<set>` header file. The most commonly used set in C++ is `std::set`, which is a template class. The type of elements that can be stored in a set is defined by the template parameter.

Here's an example of how to declare and use a set in C++ STL:

```
#include <iostream>
#include <set>

int main() {
    std::set<int> mySet; // declare a set of integers

    mySet.insert(10);
    mySet.insert(20);
    mySet.insert(30); // insert elements into the set

    std::cout << "Size of mySet: " << mySet.size() << std::endl; // output: 3

    if (mySet.find(20) != mySet.end()) {
        std::cout << "20 is in the set." << std::endl; // output: 20 is in the set.
    }

    // Erase an element from the set
    mySet.erase(20);
```

```

// Print all elements in the set
for (const auto& e : mySet) {
    std::cout << e << std::endl;
}

// Accessing set using a for loop
set<int> s = {1, 2, 3, 4};
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << " ";
}
// Output: 1 2 3 4

return 0;
}

```

NOTE

- The `find()` function for sets takes one argument, which is the value to be searched in the set. It returns an iterator to the element if it is found in the set, otherwise it returns an iterator to the `end()` of the set. For instance, let `s` be a set.
Then, **`s.end() == s.find(val)`** if **`val`** is not present in the set.
- The `find()` method of set is not purely same as method being used with vectors that requires 3 arguments i.e start iterator, end iterator and value to be searched
- **set** does not provide subscript operator `[]` to access elements like vector, string. To access its elements **iterator** can be used
- **size()** method can be used to check size of a set like a vector, map.

Unordered_Set

`std::unordered_set` is an unordered associative container implemented as a hash table. It stores unique elements in no particular order, and provides constant $O(1)$ average-case time complexity for most operations, including insertions, deletions, and searches.

Here are some key notes on `std::unordered_set`:

1. Elements are stored based on their hash values, which are calculated using a hash function provided by the user or by the default hash function of the type of the element.
2. Since the order of elements is not fixed, iteration through the elements of an unordered set can be in any order and is not guaranteed to be sorted.
3. `std::unordered_set` allows for duplicate detection of elements, but only stores unique elements. If you attempt to insert a duplicate element, it will not be added to the set.
4. The `load_factor` of an unordered set is the average number of elements per bucket. A low load factor means that the set has many buckets with few elements, while a high load factor means that the set has fewer buckets with more elements. The default load factor is 1.0.
5. `std::unordered_set` provides the same set of functions as `std::set`, including `insert()`, `erase()`, and `find()`, but with constant time complexity on average rather than $\log(n)$. However, the worst-case time complexity of some operations can be $O(n)$ due to hash collisions.
6. `std::unordered_set` can be useful for applications such as fast lookups, removing duplicates, and implementing set operations such as intersection and union. However, it may not be suitable for cases where the order of elements matters, or when you need to iterate through the elements in a specific order.

C++ code to demonstrate the implementation of `std::unordered_set`:

```
#include <iostream>
#include <unordered_set>
#include <string>
```

```
int main() {  
    // Declare an unordered_set with string elements  
    std::unordered_set<std::string> mySet;  
  
    // Insert elements into the set  
    mySet.insert("apple");  
    mySet.insert("banana");  
    mySet.insert("orange");  
  
    // Check if an element is present in the set  
    if (mySet.find("pear") != mySet.end()) {  
        std::cout << "Pear is in the set." << std::endl;  
    } else {  
        std::cout << "Pear is not in the set." << std::endl;  
    }  
  
    // Erase an element from the set  
    mySet.erase("banana");  
  
    // Print all elements in the set  
    for (const auto& e : mySet) {  
        std::cout << e << std::endl;  
    }  
  
    return 0;  
}
```

Priority Queue

A priority queue in C++ STL is a **container adapter** that provides constant time lookup for the highest or lowest element, depending on the priority set. It is implemented using a heap data structure.

Here are some important features of a priority queue in C++ STL:

1. Default priority queue is a max-heap i.e. the largest element is always on the top. We can create a min-heap by passing a comparator function as a template argument.
2. The priority queue can be initialized using a range of elements, an array, or an initializer list.
3. The priority queue provides constant time access to the top element using the **top()** function.
4. The elements in a priority queue are automatically sorted according to the priority set.
5. The **push()** function is used to insert an element into the priority queue.
6. The **pop()** function is used to remove the top element from the priority queue.
7. The **size()** function is used to return the number of elements in the priority queue.
8. The **empty()** function is used to check if the priority queue is empty.

Here is an example of how to use priority queue in C++ STL:

```
int main() {  
    priority_queue<int> pq;  
  
    pq.push(10);  
    pq.push(20);  
    pq.push(30);  
  
    while (!pq.empty()) {  
        cout << pq.top() << " ";  
        pq.pop();  
    }
```

```
}  
    return 0;  
}
```

Priority Queue can be implemented in many Different ways:

- `priority_queue<int> pq;`
- `priority_queue<pair<int, vector<pair<int, int>>>> cont;`

Max-Heap

A heap is a **specialized tree-based data structure** that satisfies the heap property. In a heap, the **parent node is either greater than or equal to** (for max heap) or less than or equal to (for min heap) its children.

Heaps are a powerful data structure that can provide efficient access to the maximum or minimum element of a collection.

A max heap is a binary tree where the value of each parent node is greater than or equal to the values of its children nodes. In a max heap, the node with the highest value is always the root node.

The heap property in a max heap means that every parent node is greater than or equal to its children. The largest value is always at the root of the heap.

Some important operations in a max heap include:

Insertion: Inserting a new node into the heap while maintaining the heap property.

Deletion: Removing the root node while maintaining the heap property.

Peek: Returning the value of the root node without removing it from the heap.

A heap is commonly implemented as an array, where the root is stored at index 0, and the left and right children of a node at index i are stored at

indices $2i+1$ and $2i+2$, respectively. This makes it easy to maintain the heap property and to perform heap operations efficiently.

Heap operations have a time complexity of $O(\log n)$ for both insertion and deletion, where n is the number of nodes in the heap. The peek operation has a time complexity of $O(1)$.

NOTE

Default Implementation of Priority Queue builds Max Heap

- `priority_queue<int> pq`

Min Heap

A min heap is a binary tree where **the parent node is always smaller than or equal to its child nodes**. This means that the minimum element in the heap is always stored at the root of the tree. Min heaps are commonly used in algorithms that require finding the minimum element in a set of elements, such as Dijkstra's shortest path algorithm.

In C++, you can implement a min heap using the `std::priority_queue` container adapter by specifying a **comparison function** that returns true when its first argument should appear after its second argument in the heap ordering. To create a min heap using a priority queue, you can use the following code:

- `priority_queue<int, vector<int>, greater<int>> pq;`

NOTE

- Min heaps do not remove duplicates by default. Means we have 2 same values at the top and another element after top. That's why we need to pop 1 extra element.

In the context of creating a min heap using a `std::priority_queue` in C++, the `vector<int>` template parameter refers to the underlying container used to implement the priority queue.

A priority queue is a container adapter that provides constant time access to the largest (or smallest) element in the container, based on a comparison function. The underlying container stores the elements in the priority queue and provides the basic operations of insertion, removal, and access to elements.

The default underlying container for `std::priority_queue` is `std::vector`. When you specify the template parameters for `std::priority_queue`, the second parameter specifies the container type to use, and the third parameter specifies the comparison function type.

For example, `std::priority_queue<int, vector<int>, greater<int>>` specifies a priority queue of integers, implemented using `std::vector<int>` as the underlying container, and using `std::greater<int>` as the comparison function to create a min heap.

By default, `std::priority_queue` uses `std::less<T>` as the comparison function, which creates a max heap.

Bit Manipulation

C++ function to count the number of 1's bits in an int

Syntax: `__builtin_popcount(x)`

E.g

```
result = __builtin_popcount(5);           // result will be 2
```

Unsigned int 8-bit

`uint8_t` - Equivalent to unsigned char, Consume 1 byte

Unsigned int 16-bit

`uint16_t` - Equivalent to unsigned short, Consume 2 bytes

Unsigned int 32-bit

`uint32_t` - Equivalent to unsigned int, Consume 4 bytes

Unsigned int 64-bit

`uint64_t` - Equivalent to unsigned long long, Consume 8 bytes