C++ STL Intermediate

- Viraj Chandra

Goal

To understand:

- Continued
  - Set
  - Unordered Set
  - Map
  - Unordered Map
- String
- Custom Sort

Set

Set in C++ is a container that stores unique, ordered elements.

For sets to work for some data type, the data type must have **inbuilt comparators implemented.** Features include:

- Stores Unique Elements
- Ordered Elements
- Efficient Lookup – Searching for an element takes O(log n).

1, 2, 3

insert(3)

1, 2, 3, 3 ✗

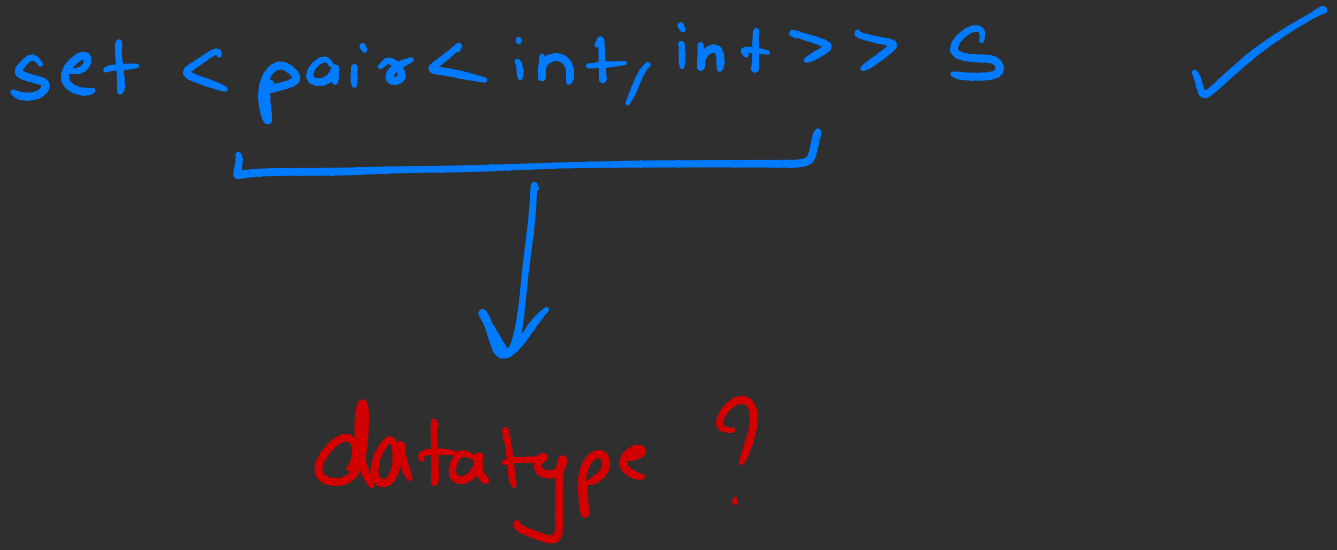
1, 2, 3 ✓

1, 2, 4

insert(3)

1, 2, 4, 3 ✗

1, 2, 3, 4 ✓



Set (Tree)

s[i] x
v[i] ✓

Operation	Syntax	Time Complexity
Declare	set<int> s;	O(1)
Insert Element	s.insert(x);	O(log n)
Remove Element	s.erase(x);	O(log n)
Find Element	s.find(x);	O(log n)
Count Element	s.count(x);	O(log n)
Size of Set	s.size();	O(1)
Check Empty	s.empty();	O(1)
Iterate Over Set	for(auto x : s) cout << x;	O(n)
Clear Set	s.clear();	O(n)

```
for(auto e:vec)
    cout << e << "  ";
```

✓

```
for(int i:0 . . .  i+=2)
    v[i]
```

✓

lowerbound (s.begin() ... , t) ;

Lowerbound on set? Tricky ?

Set is typically implemented as a self-balancing tree.

- s.lower_bound will traverse the tree knowing the properties of the tree structure.
  - **Time Complexity - O(logN)**
- lower_bound(s.begin(), s.end()) needs to run something akin to a binary search over the data. Much slower.
  - **Time Complexity - O(N)**

Set - Code Example

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

int main() {
    set<int> s;

    // Inserting elements
    s.insert(5);
    s.insert(2);
    s.insert(8);
    s.insert(2); // Duplicate, won't be added

    // Checking presence
    if (s.find(5) != s.end())
        cout << "5 is present" << endl;

    // Removing element
    s.erase(2);

    // Iterating over set
    for (int x : s)
        cout << x << " "; // Output: 5 8 (sorted order)

    return 0;
}
```

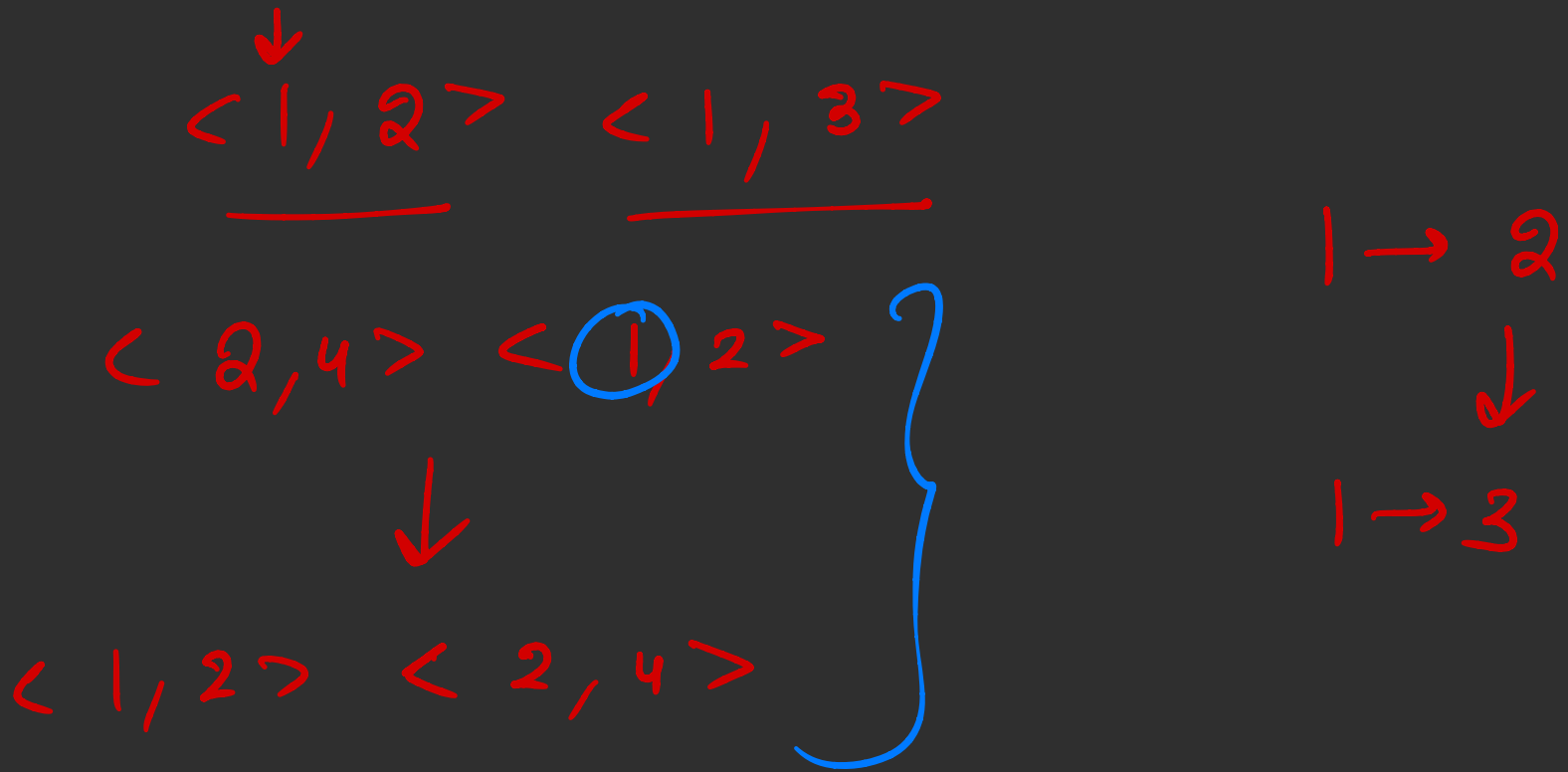
Map

< key, value >

Map in C++ is a key-value pair container that stores elements in sorted order based on the key.

For maps to work for some data type, the data type must have **inbuilt comparators implemented.** Features include:

- Stores Unique Keys – Each key must be unique.
- Ordered Storage – Keys are stored in sorted order (ascending by default).
- Efficient Lookups – Searching for a key takes O(log n).



```
map<int, int> mp;

map<int, pair<int,int>> mp;
```

Map

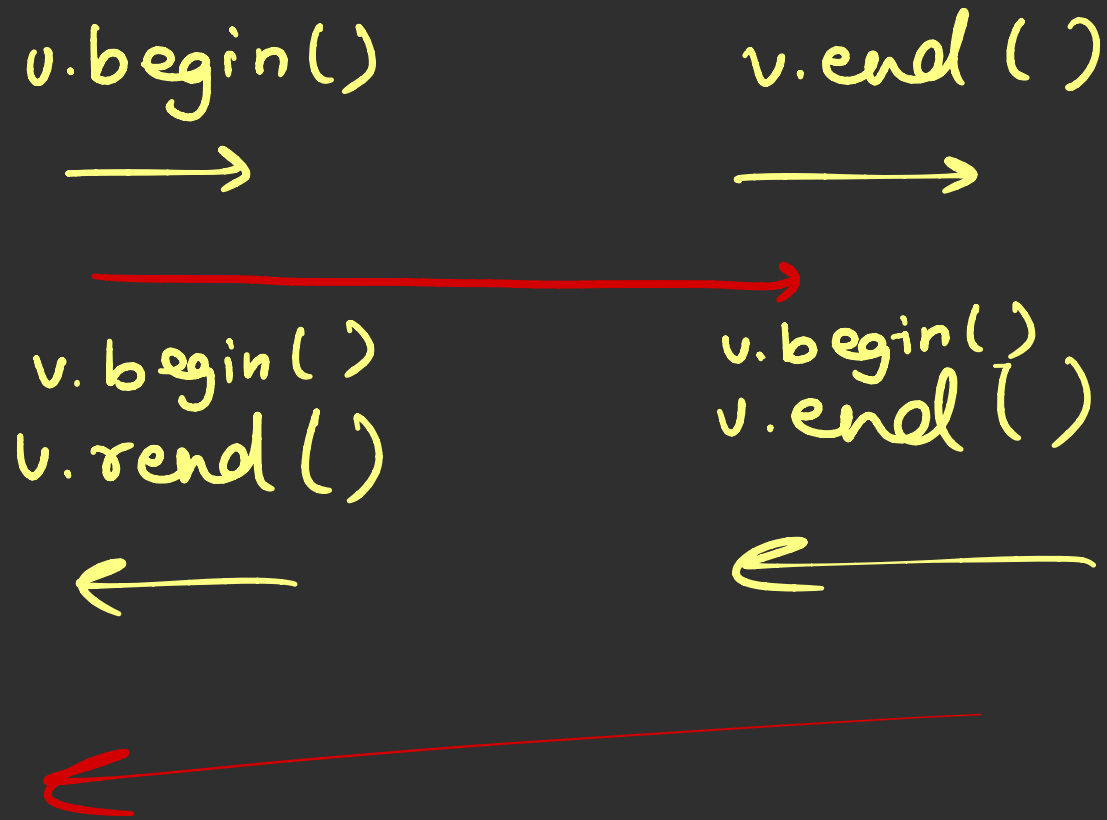
p = { 2, 3 }

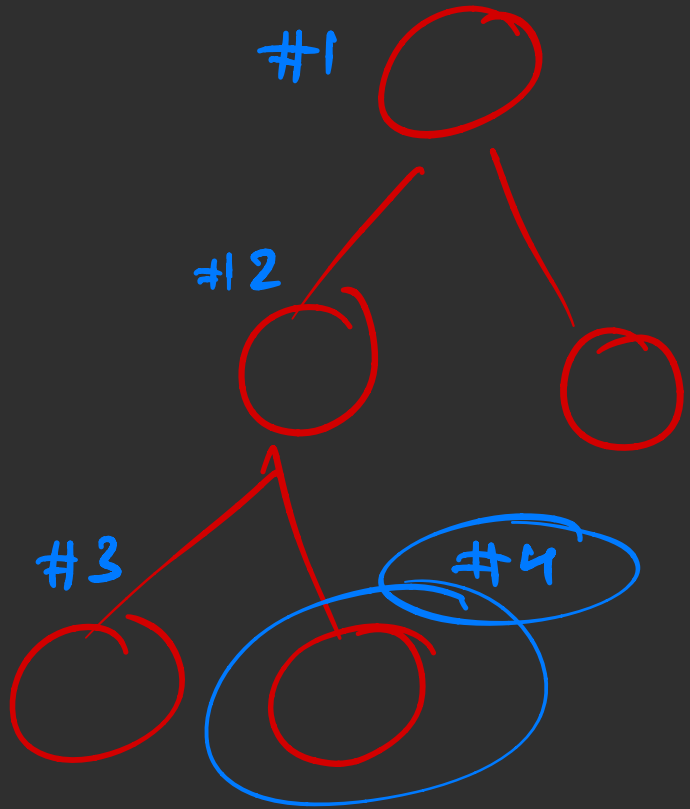
Operation	Syntax	Time Complexity
Insert / Assign	mp[key] = value;	O(log N)
Insert (explicit)	mp.insert({key, value});	O(log N)
Erase by Key	mp.erase(key);	O(log N)
Erase by Iterator	mp.erase(it);	O(1)
Find Element	mp.find(key);	O(log N)
Check if Exists	mp.count(key);	O(log N)
Access Element	mp[key]	O(log N)
Get First Element	mp.begin();	O(1)
Get Last Element	mp.rbegin(); (end)	O(1)
Size of Map	mp.size();	O(1)

① `sort(v.begin(), v.end());`
`reverse(v.begin(), v.end());` } X

② `sort(v.begin(), v.end(), greater<int>());`

③ `sort(v.end(), v.begin());` X
`sort(v.rbegin(), v.rend());` } ✓





Map - Code Example

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

int main() {
    map<int, string> mp;

    // Inserting key-value pairs
    mp[1] = "Alice";
    mp[3] = "Bob";
    mp[2] = "Charlie";

    // Iterating over map (keys are sorted)
    for (auto p : mp)
        cout << p.first << " -> " << p.second << endl;

    // Searching for a key
    if (mp.find(3) != mp.end())
        cout << "Key 3 found!" << endl;

    return 0;
}
```

Unordered Set

An unordered_set in C++ is a hash-based container that stores unique elements in an unordered manner.

For sets to work for some data type, the data type must have **inbuilt hash function implemented.** Features include:

- Unique Elements
- Unordered Storage
- Fast Operations – Average O(1) for insert(), erase(), and find().
- Hash Collisions Possible – Can degrade to O(n) in worst cases.

% 1e9+7

a[i] ≈ 10^4 - 10^5

↓

a[i] ≈ 10^{18}

123 % 63 = 60

123 → hashed → 60

<5, 7>

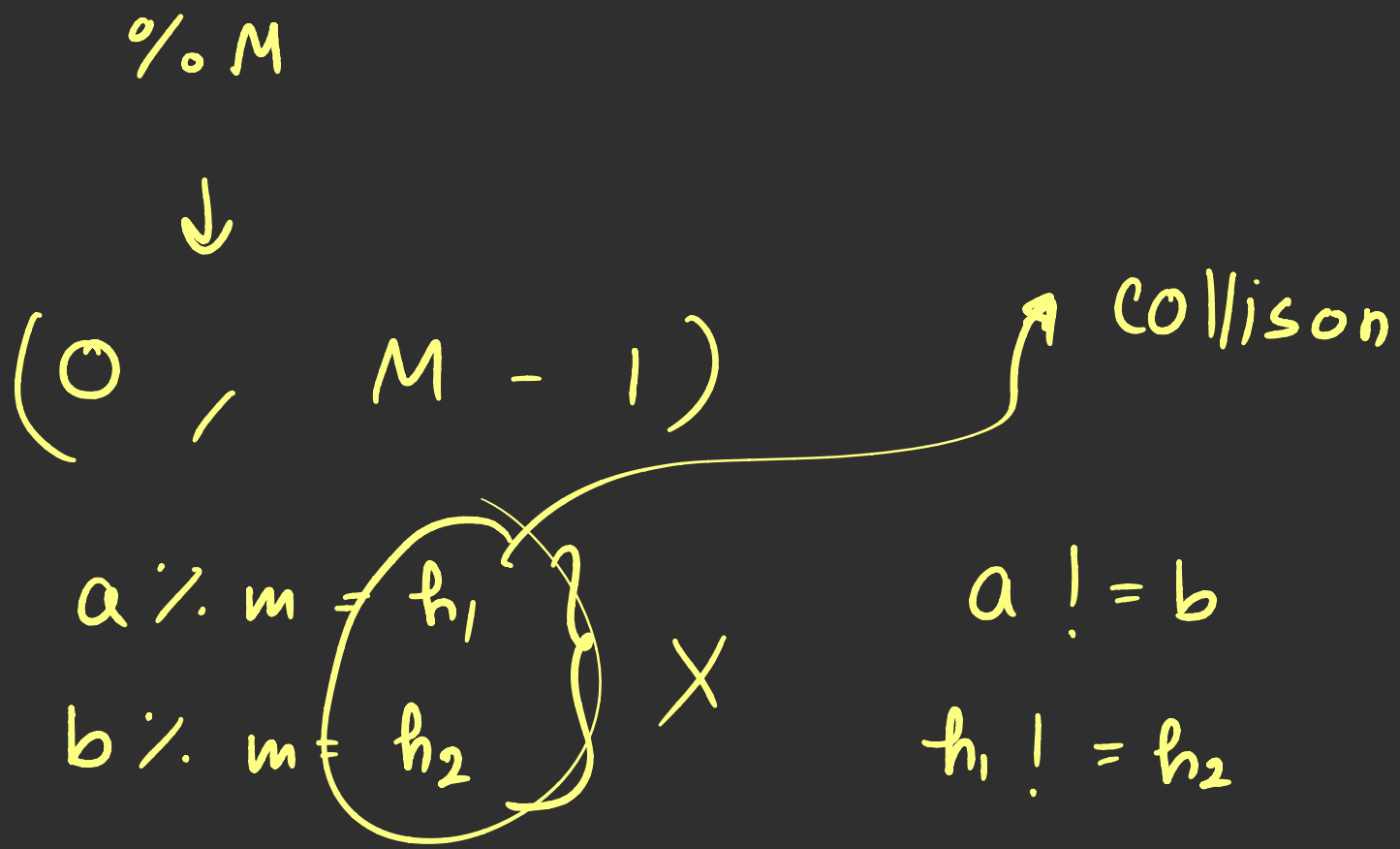
<1, "Viraj">

$10 \rightarrow h_1$

$20 \rightarrow h_2$

$30 \rightarrow h_3$

$30 \rightarrow h_4 = h_3$ X



unordered_set<int> s;

unordered_set<pair<int, int>> s;

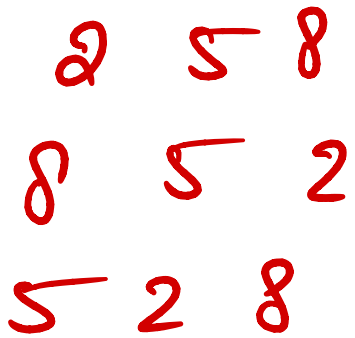
set<pair<int, int>> s;

Unordered Set

Operation	Syntax	Time Complexity
Declare	unordered_set<int> us;	O(1)
Insert Element	us.insert(x);	O(1) (Amortized)
Remove Element	us.erase(x);	O(1) (Amortized)
Find Element	us.find(x);	O(1) (Amortized)
Count Element	us.count(x);	O(1) (Amortized)
Size of Set	us.size();	O(1)
Check Empty	us.empty();	O(1)
Iterate Over Set	for(auto x : us) cout << x;	O(n)
Clear Set	us.clear();	O(n)

Unordered Set - Code Example

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

int main() {
    unordered_set<int> us;

    // Inserting elements
    us.insert(5);
    us.insert(2);
    us.insert(8);
    us.insert(2); // Duplicate, won't be added

    // Checking presence
    if (us.find(5) != us.end())
        cout << "5 is present" << endl;

    // Removing element
    us.erase(2);

    // Iterating over unordered_set
    for (int x : us)
        cout << x << " "; // Output is in any order

    return 0;
}
```



Unordered Map

An unordered map in C++ is an associative container that stores key-value pairs using a hash table.

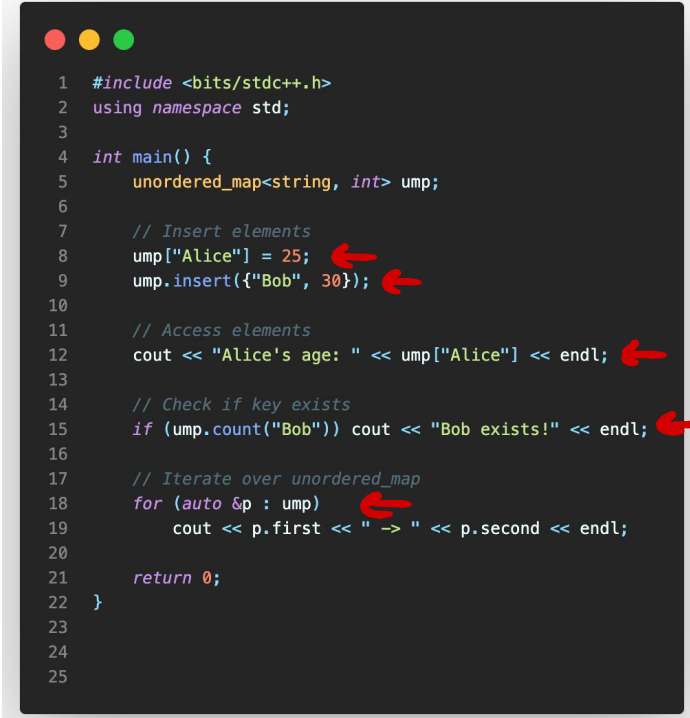
Unlike map, it does not maintain any order of keys and provides average O(1) time complexity. However, in the worst case, when hash collisions occur, these operations may take O(N) time. Features include:

- Stores Unique Elements
- Unordered Elements
- Fast Access

Unordered Map

Operation	Syntax	Time Complexity
Insert / Assign	ump[key] = value;	O(1) (avg), O(N) (worst)
Insert (explicit)	ump.insert({key, value});	O(1) (avg), O(N) (worst)
Erase by Key	ump.erase(key);	O(1) (avg), O(N) (worst)
Erase by Iterator	ump.erase(it);	O(1)
Find Element	ump.find(key);	O(1) (avg), O(N) (worst)
Check if Exists	ump.count(key);	O(1) (avg), O(N) (worst)
Access Element	ump[key]	O(1) (avg), O(N) (worst)
Get First Element	ump.begin();	O(1)
Get Last Element	ump.rbegin();	O(1)
Size of Map	ump.size();	O(1)

Unordered Map - Code Example



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

int main() {
    unordered_map<string, int> ump;

    // Insert elements
    ump["Alice"] = 25;
    ump.insert({"Bob", 30});

    // Access elements
    cout << "Alice's age: " << ump["Alice"] << endl;

    // Check if key exists
    if (ump.count("Bob")) cout << "Bob exists!" << endl;

    // Iterate over unordered_map
    for (auto &p : ump)
        cout << p.first << " -> " << p.second << endl;

    return 0;
}
```

String

A string in C++ is a sequence of characters stored in contiguous memory.

Features include:

- Dynamic size unlike character arrays (char[])
- Supports indexing and iterators
- Supports comparison operators (<, >, ==)
- Compatible with STL functions (sort(), reverse(), find(), etc.).

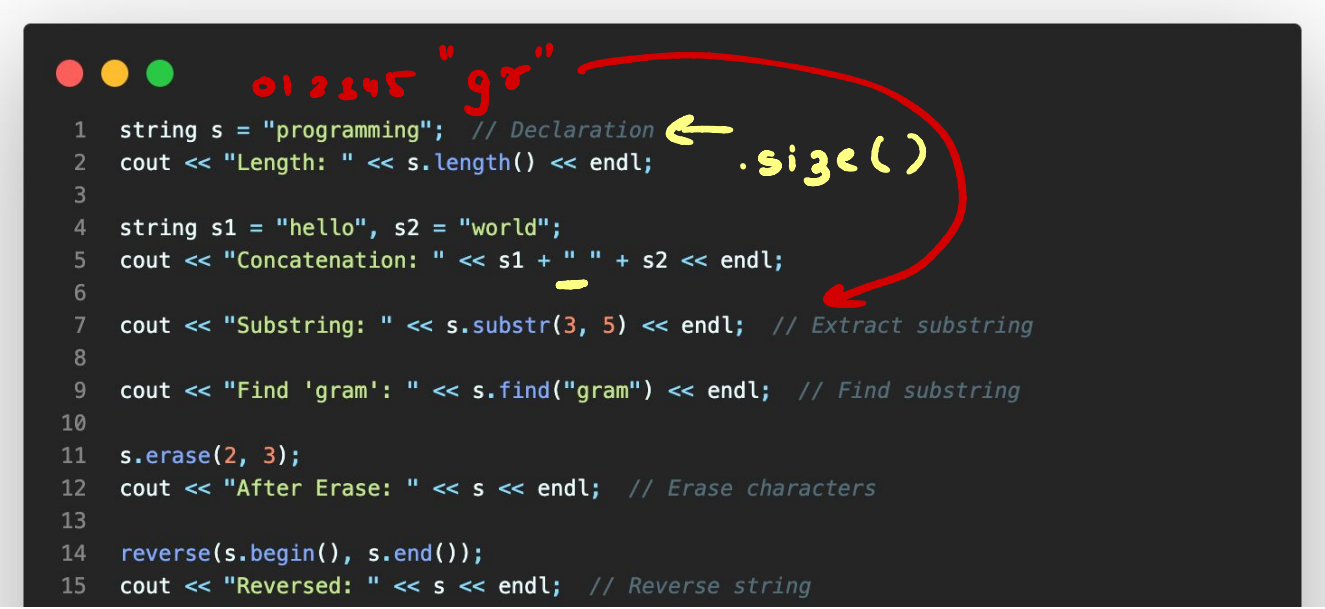
abc abcd

ab bc

Lexicographical Order

↳ Dictionary

String - Code Example



```cpp
string s = "programming";  // Declaration
cout << "Length: " << s.length() << endl;

string s1 = "hello", s2 = "world";
cout << "Concatenation: " << s1 + " " + s2 << endl;

cout << "Substring: " << s.substr(3, 5) << endl;  // Extract substring

cout << "Find 'gram': " << s.find("gram") << endl;  // Find substring

s.erase(2, 3);
cout << "After Erase: " << s << endl;  // Erase characters

reverse(s.begin(), s.end());
cout << "Reversed: " << s << endl;  // Reverse string
```

S.substr (a, b)

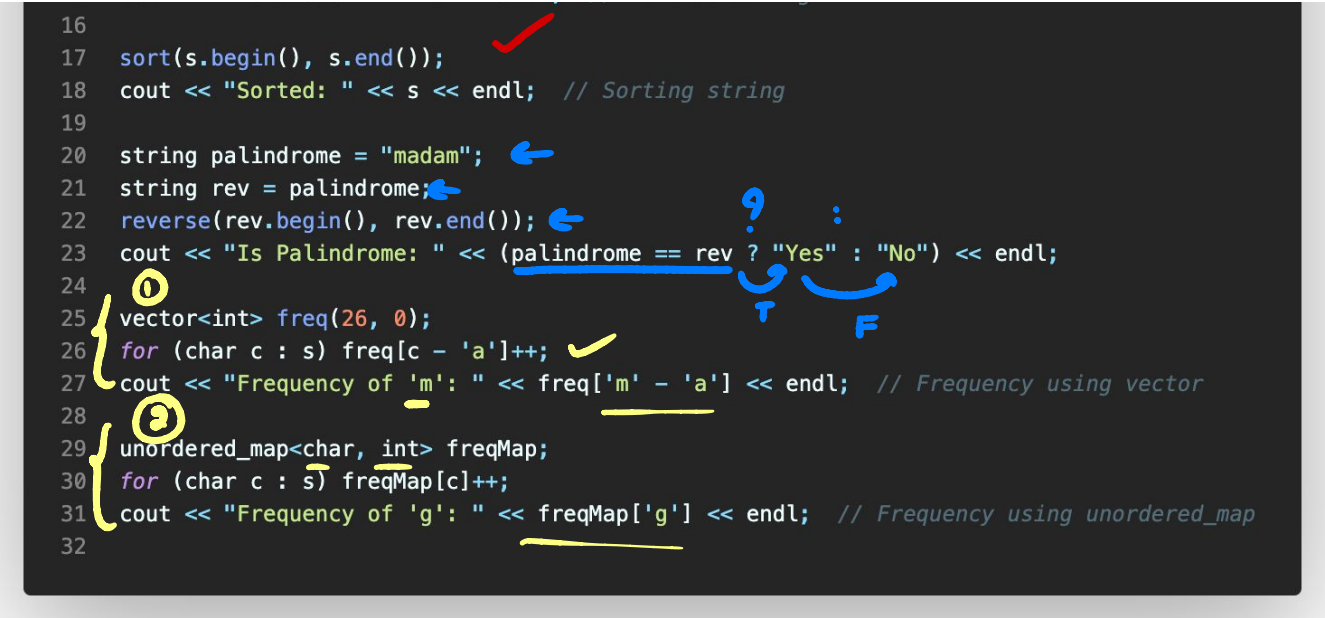
↳ optional

① S.substr (4) ;

(4th index → end) → substr

② S.substr(3, 5) → 3 to 4 index

String - Code Example



```
16
17  sort(s.begin(), s.end());
18  cout << "Sorted: " << s << endl;  // Sorting string
19
20  string palindrome = "madam";
21  string rev = palindrome;
22  reverse(rev.begin(), rev.end());
23  cout << "Is Palindrome: " << (palindrome == rev ? "Yes" : "No") << endl;
24
25  vector<int> freq(26, 0);
26  for (char c : s) freq[c - 'a']++;
27  cout << "Frequency of 'm': " << freq['m' - 'a'] << endl;  // Frequency using vector
28
29  unordered_map<char, int> freqMap;
30  for (char c : s) freqMap[c]++;
31  cout << "Frequency of 'g': " << freqMap['g'] << endl;  // Frequency using unordered_map
32
```

abcdaa

a - 3

b - 1

c - 1

d - 1

0 1 0 0 0 0 . . . . . .

0 1 2 3 4 5 25

'b' - 'a'

↳ 98 - 97 = 1

['b' - 'a']

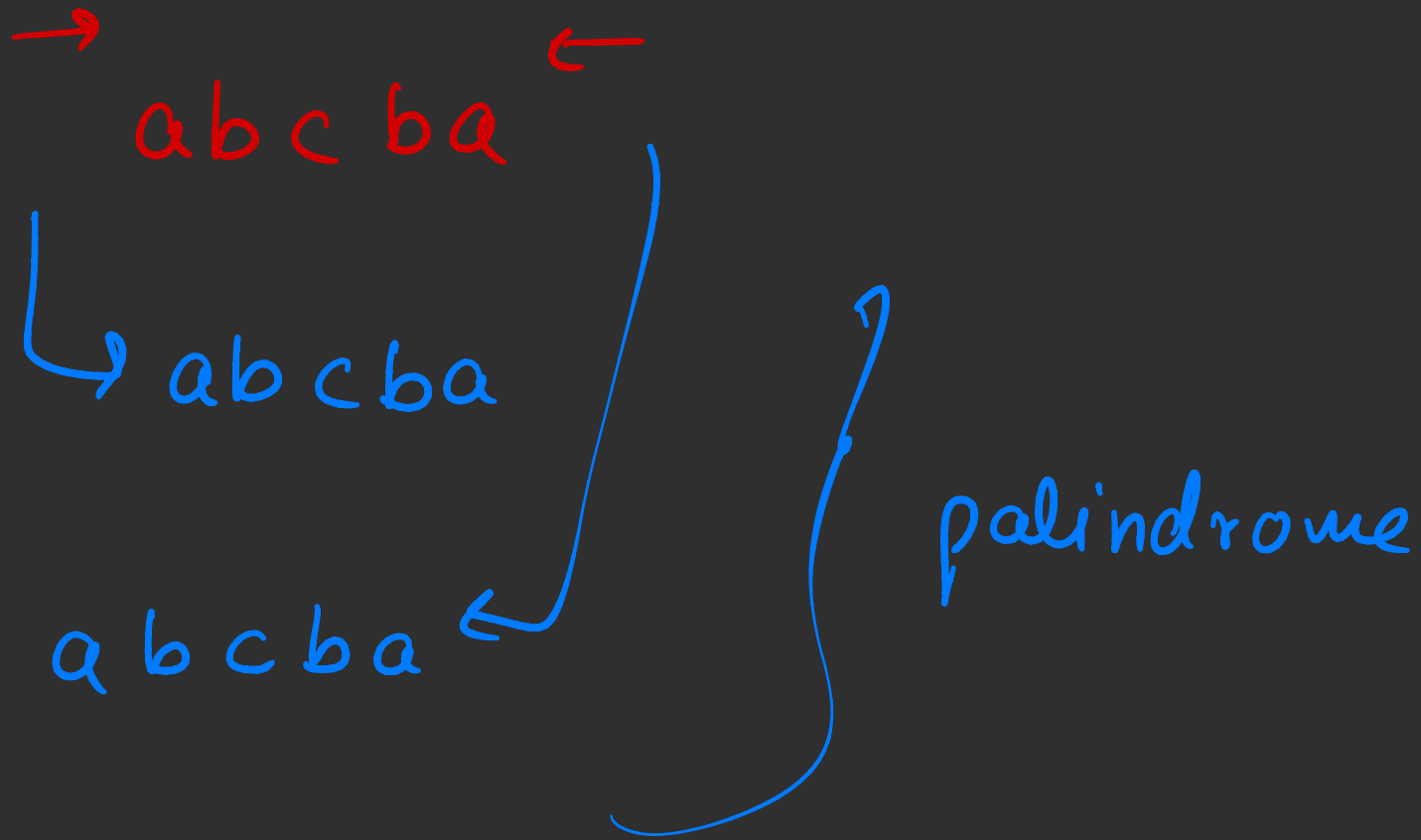
ASCII Code

'A' → 'Z'

65 → 90

'a' → 'z'

97 → 122



Custom Sorting

$a < b$
$b < a$
} $O(1)$

By default, **sort()** sorts elements in ascending order using a hybrid of QuickSort, HeapSort, and InsertionSort.

However, we can customize the sorting order using a custom comparator function.

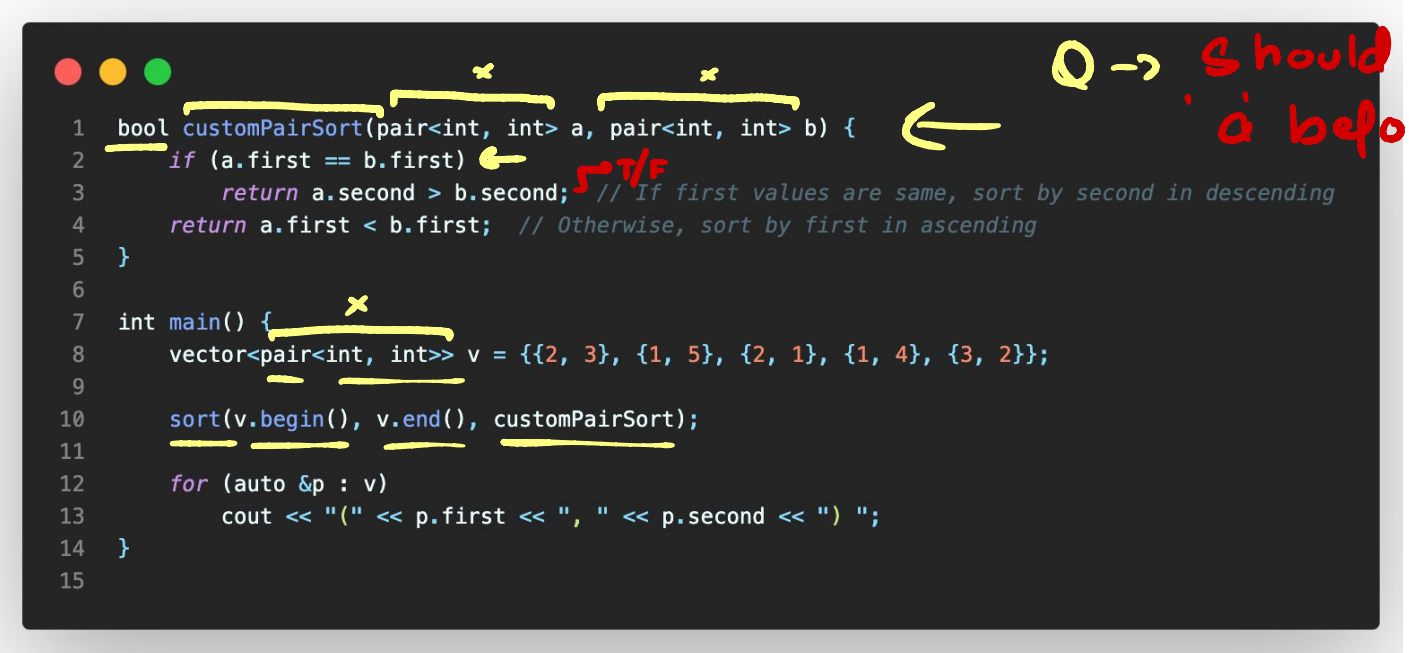
Time Complexity: O(NlogN * TC(comparator))

$s1 < s2$

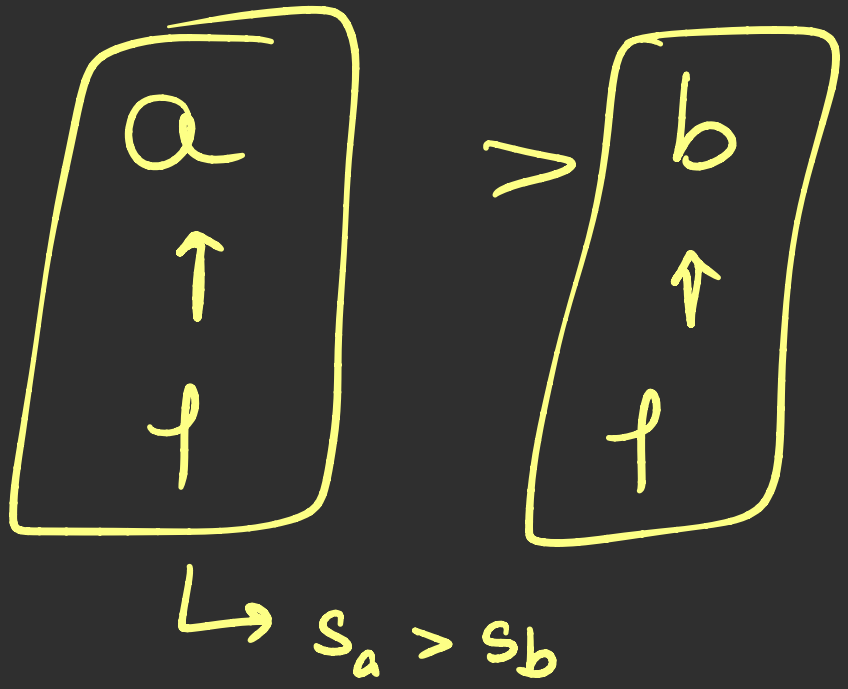
vector < string > v

NlogN x (size of string)

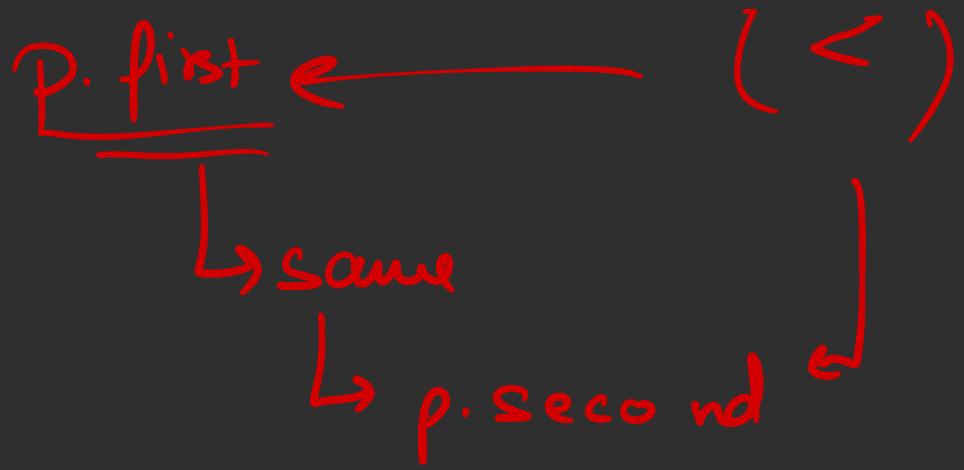
Custom Sorting - Code Example




```cpp
bool customPairSort(pair<int, int> a, pair<int, int> b) {
    if (a.first == b.first)
        return a.second > b.second; // If first values are same, sort by second in descending
    return a.first < b.first;  // Otherwise, sort by first in ascending
}

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

    sort(v.begin(), v.end(), customPairSort);

    for (auto &p : v)
        cout << "(" << p.first << ", " << p.second << ") ";
}
```



$\hookrightarrow s_a > s_b$



P. first

same → p. second (>)

When to Use Custom Sorting?

- When sorting pairs, structures, or complex objects.
- When a non-standard sorting order is needed.
- When sorting in descending order with specific conditions.

NOTE: Avoid if **default sorting** meets the requirement.

Important Links [Bonus]

- https://www.cppreference.com/Cpp_STL_ReferenceManual.pdf
- https://devdocs.io/cpp/container (for STL containers)
- https://devdocs.io/cpp/algorithm (for STL algorithms)

Using the above resources, try to learn about multiset, multimap.

{1, 2, 3, 3} X