📘 `std::map` in C++

🧠 What is `std::map`?

`std::map` is a **sorted associative container** in the C++ Standard Template Library (STL) that stores elements as **key-value pairs**. Each key is **unique**, and the map automatically sorts the elements by key using the **less-than operator** (`<`) by default.

🧱 Key Characteristics

- **Ordered**: Elements are stored in sorted order based on the key.
- **Unique keys**: No duplicate keys are allowed.
- **Logarithmic time complexity**: Insertion, deletion, and lookup operations take O(log n) time.
- **Implemented as**: A self-balancing binary search tree (typically a Red-Black Tree).

🧾 Syntax

```cpp
#include <iostream>
#include <map>
using namespace std;

map<KeyType, ValueType> mapName;
```

Example:

```cpp
map<string, int> studentGrades;
```

🧪 Basic Operations

1. Insertion

```cpp
studentGrades["Alice"] = 90;
auto result = studentGrades.insert({"Bob", 85});
```

insert will return a key-value pair. The first of the pair is the iterator to the item in the map. The second of the pair is a bool. If it is true, then the item was inserted into the map otherwise it was not.

NOTES:

- map[key] will **overwrite** any existing value
- insert will NOT overwrite

Insert return value

The insert method will return a pair object.

- The first part of the pair is the iterator to the key-value pair object in the map.

```cpp
auto result = studentGrades.insert({"Bob", 85});
map<string, int>::iterator keyValuePair = result.first;
//this is the key-value pair that is in the map
```

- The second part of the pair is a bool.
  - If the value is **false**, then the key-value pair was NOT inserted.
  - If the value is **true**, then the key-value pair was inserted.

```cpp
auto result = studentGrades.insert({"Bob", 85});
bool wasInserted = result.second;
//this is the bool that indicates if the key-value pair was inserted
```

2. Accessing Elements

```cpp
cout << studentGrades["Alice"]; // Outputs: 90
```

> using map[key] to access a value will ADD the key if it is not in the map. Use find instead if you do not want this side-effect.

3. Iterating

Traditional Iteration (Pre-C++17)

```cpp
for (auto it = myMap.begin(); it != myMap.end(); ++it) {
    cout << it->first << ": " << it->second << endl;
}
```

Range-Based For Loop (C++11 and later)

```cpp
for (const auto& pair : myMap) {
    cout << pair.first << ": " << pair.second << endl;
}
```

Structured Bindings (C++17)

```cpp
for (const auto& [key, value] : myMap) {
    cout << key << ": " << value << endl;
}
```

Explanation:

- `auto& [key, value]` unpacks the `std::pair<const Key, Value>` into named variables.
- This improves readability and avoids accessing `.first` and `.second` explicitly.

4. Finding Elements

The `find()` method is used to **search for a key** in a `std::map`. It returns an **iterator** to the element if found, or `map.end()` if not.

Syntax

```cpp
auto it = myMap.find(key);
if (it != myMap.end()) {
    // Access key and value
    cout << "Key: " << it->first << ", Value: " << it->second << endl;
}
```

With Structured Bindings (C++17)

You can also use structured bindings with the iterator:

```cpp
auto it = myMap.find("Alice");
if (it != myMap.end()) {
    const auto& [key, value] = *it; //bind the first and second to the key,value variables.
    cout << "Found " << key << " with value " << value << endl;
}
```

Explanation:

- `myMap.find("Alice")` returns an iterator to the pair.
- `*it` dereferences the iterator to get the `std::pair`.
- Structured binding unpacks the pair into `key` and `value` variables.

🧪 find Example

```cpp
#include <iostream>
#include <map>
using namespace std;

int main() {
    map<string, int> scores = {{"Alice", 90}, {"Bob", 85}, {"Charlie", 92}};

    // Iteration using structured bindings
    for (const auto& [name, score] : scores) {
        cout << name << ": " << score << endl;
    }

    // Using find to access a specific key
    auto it = scores.find("Bob");
    if (it != scores.end()) {
        const auto& [name, score] = *it;
        cout << "Found " << name << " with score " << score << endl;
    }

    return 0;
}
```

Output:

```
Alice: 90
Bob: 85
Charlie: 92
Found Bob with score 85
```

---

5. Erasing Elements

- **`erase(iterator pos)`**

Removes the element at the position pointed to by the iterator.

```cpp
std::map<int, std::string> myMap;
myMap[1] = "one";
myMap[2] = "two";

auto it = myMap.find(1);
if (it != myMap.end()) {
    myMap.erase(it);  // Removes the element with key 1
}
```

Use Case:

- When you already have an iterator to the element you want to remove.
- Efficient: O(1) amortized time.

- **`erase(const key_type& key)`**

Removes the element with the specified key.

```cpp
myMap.erase(2);  // Removes the element with key 2
```

Return Value:

- Returns the number of elements removed (0 or 1 for `std::map` since keys are unique).

✅ When to Use `std::map`

Use `std::map` when:

- You need **sorted** key-value pairs.
- You require **fast lookups** by key.
- You want to **avoid duplicate keys**.

🎯 Quiz!

Here's a short quiz on the topic: quiz

Footer Separator

🔭 Markdown Viewer

How to view the markdown files in a browser...

- Markdown Viewer

🧠 Lecture Practices

Here are the lecture Practices...

- Day 4
- Day 5
- Day 6

🔍 Lecture Quizzes

Here are the lecture quizzes...

- Day 4
- Day 5
- Day 6

Weekly Topics

Here are the topics for the week...

- Recursion
- Pseudocode
- Sorting
- Searching
- Maps
- Time Complexity