C++ STL `map` Overview

Definition

The **`map`** is an **ordered associative container** in the Standard Template Library (STL) that stores elements in **key-value pairs**. It is implemented as a **self-balancing binary search tree (typically a Red-Black Tree)**, ensuring that elements are always sorted by key.

- **Keys are unique** – no duplicate keys are allowed.
- **Values are mutable** – can be modified after insertion.
- **Operations are logarithmic (O(log n))** due to tree-based implementation.

Example 1: Create a Map to translate digits from English to Italian.

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

void main()
{
    //Example 1: Create a Map to translate digits from English to Italian.
    map<string, string> mItalianDigits =
    {
        { "zero",   "zero"},
        { "one",    "uno"},
        { "two",    "due"},
        { "three",  "tre"},
        { "four",   "quattro"},
        { "five",   "cinque"},
        { "six",    "sei"},
        { "seven",  "sette"},
        { "eight",  "otto" },
        { "nine",   "nove"},
    };

}
```

Figure 1. A red-black tree representing an English-to-Italian dictionary (mDictionary). The keys are English words, and the values are their Italian equivalents.

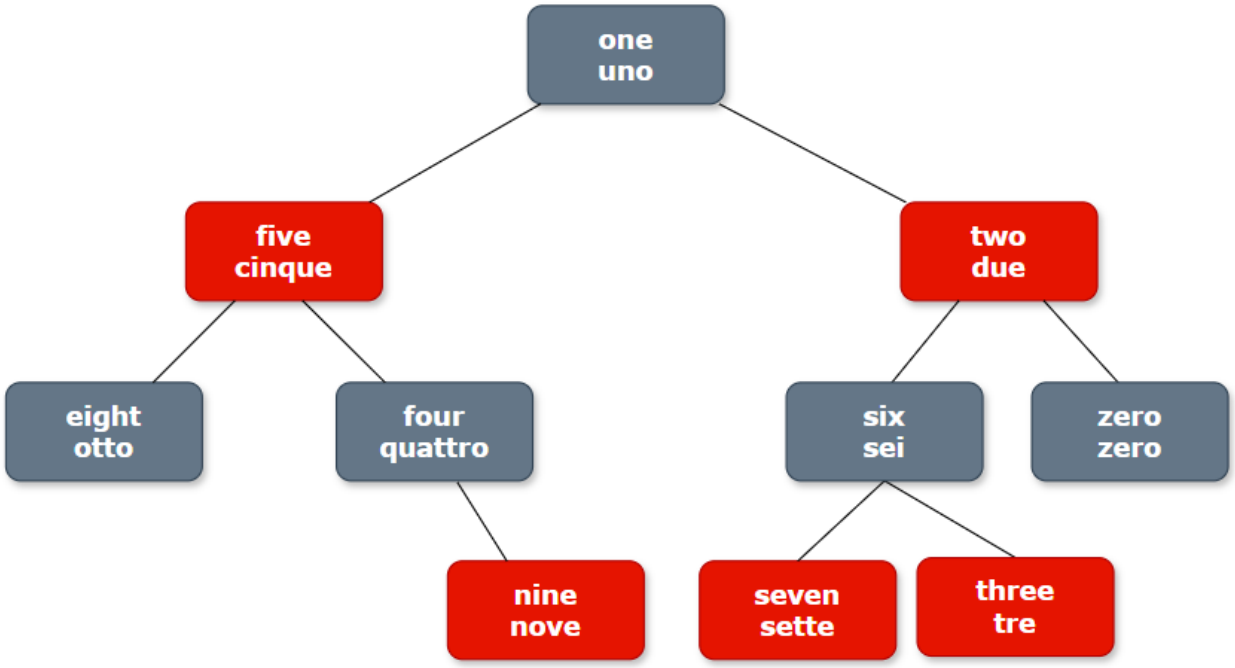

Figure 2. An alternative interpretation of the mDictionary of Example 1, shown as a sorted table.

Key	Value
eight	Otto
five	Cinque
four	Quattro
nine	Nove
one	Uno
seven	Sette
Six	Sei
three	Tre
two	Due
zero	Zero

Basic Operations with `map`

1. Declare and Initialize a `map`

The following code fragment creates a map where **the key is a character's name (string) and the value is their age (int).**

```
    // Declare and initialize a map with Simpsons characters (name, age)
    map<string, int> simpsons = {
        {"Homer", 39},
        {"Marge", 36},
        {"Bart", 10},
        {"Lisa", 8},
        {"Maggie", 1}
    };

    // Output the map
    for (const auto& [name, age] : simpsons) {
        cout << name << " is " << age << " years old.\n";
    }
```

Output: Maps automatically sort keys in ascending order, so the output will be:

```
Bart is 10 years old.
Homer is 39 years old.
Lisa is 8 years old.
Maggie is 1 years old.
Marge is 36 years old.
```

2. Inserting and Accessing Elements

You can insert new key-value pairs using **`insert()`** or **`operator[]`**.

```
    // Insert elements
    simpsons["Grandpa"] = 83;               // Insert using operator[]
    simpsons.insert({ "Mr. Burns", 104 });  // Insert using insert()

    // Access elements
    cout << "Grandpa's age: " << simpsons["Grandpa"] << "\n";

    // Access an element that doesn't exist
    cout << "Flanders' age: " << simpsons["Flanders"] << "\n";
    // This will insert a new pair with key "Flanders" and value 0
```

3. Searching for a Key (`find()`)

Use `find()` to check if a key exists.

```
    // Check if an element exists - use find() or count()
    string search_name = "Lisa";
    auto it = simpsons.find(search_name);

    if (it != simpsons.end()) {
        cout << search_name << " found! Age: " << it->second << "\n";
    }
    else {
        cout << search_name << " not found.\n";
    }
```

4. Removing Elements (`erase()`)

To remove an entry:

```
simpsons.erase("Maggie");  // Remove Maggie from the map
```

5. Iterating Over a `map`

We can iterate over a **`map`** using:

Range-based for loop (C++17)

```
    for (const auto& [name, age] : simpsons) {
        cout << name << " is " << age << " years old.\n";
    }
```

Iterator-based loop

```
    for (auto it = simpsons.begin(); it != simpsons.end(); ++it) {
        cout << it->first << " -> " << it->second << "\n";
    }
```

6. Checking Size and Clearing

```
    cout << "Total people: " << simpsons.size() << "\n";

    simpsons.clear();  // Remove all elements
```

Key Takeaways

+ **Ordered** – Keys are always sorted.
+ **Efficient** – Logarithmic time complexity (`O(log n)`) for insert, search, and delete.
+ **Unique Keys** – Each key appears only once.
+ **Mutability** – Values can be updated.