

1. Tabla de Hash **estadisticas**

- **Definición:** `unordered_map<string, unordered_map<string, Estadisticas>> estadisticas;`
 - Este mapa usa como clave principal el nombre de un equipo (`string`) y como clave secundaria la competición en la que participa (`string`).
 - Cada entrada en el mapa contiene una instancia de la estructura **Estadisticas**, que almacena datos detallados sobre el rendimiento del equipo en una competición específica.

Then, you have an `unordered_map` named `estadisticas`, which is a nested map. It maps a `std::string` key to another `std::unordered_map` that maps a `std::string` key to an `Estadisticas` object.

Here's the map definition:

Cpp

 Copiar

```
std::unordered_map<std::string, std::unordered_map<std::string, Estadistica
```

How it works:

- The outer map uses a string as the key, representing some primary category (e.g., a team name).
- The inner map also uses a string as the key, representing a secondary category (e.g., a player name).
- The value for each key in the inner map is an `Estadisticas` object, which holds the statistics.

Here's a practical example:

Cpp

 Copiar

```
estadisticas["TeamA"]["Player1"].goles_a_favor = 5;  
estadisticas["TeamA"]["Player1"].triunfos = 3;  
  
std::cout << "Goles a favor de Player1 en TeamA: " << estadisticas["TeamA"]  
std::cout << "Triunfos de Player1 en TeamA: " << estadisticas["TeamA"]
```

In this example, the stats for `Player1` in `TeamA` are updated, and then we print out the goals in favor and the number of wins.

Feel more comfortable with that now?

-
- **RELACIÓN CONCEPTUAL CON EL CURSO**
 - `unordered_map` = Tabla de Hash para storear key-values

provides a hashtable-based implementation for storing key-value pairs. It allows for fast access, insertion, and deletion of elements based on keys, with an average time complexity of $O(1)$ for these operations.

Here's a quick rundown:

- **Headers:** To use `unordered_map`, include the header `<unordered_map>`.
- **Syntax:** Define an `unordered_map` with `std::unordered_map<KeyType, ValueType>`.
- **Insertion:** You can insert elements using the `insert()` method or the subscript operator `[]`.
- **Access:** Access elements using `operator[]` or `at()`.
- **Deletion:** Use `erase()` to remove elements.
- **Iterators:** Like other STL containers, it supports iterators to traverse elements.

Yes, which hash function does it use to map the key to values?

By default, `unordered_map` in C++ uses the `std::hash` function to map keys to values. This function is defined in the `<functional>` header and provides a hash value for various standard types, such as integers, strings, and pointers. You can also define your own custom hash function if you need more control over the hashing process.

Here's a quick example with a custom hash function:

Cpp

 Copiar

```
#include <iostream>
#include <unordered_map>
#include <functional>

struct CustomHash {
    std::size_t operator()(const std::string& key) const {
        // Custom hash function (example: length of the string)
        return std::hash<std::string>{}(key) ^ (key.length() << 1);
    }
};
```

Ok, and, is not necessary to pass any size value to the hash table?

By default, you don't need to pass a size value to the `unordered_map`. It automatically handles resizing as elements are added, ensuring efficient performance. The underlying hash table grows dynamically when the load factor (number of elements divided by the number of buckets) exceeds a certain threshold.

But if you want to improve performance, especially if you know the number of elements in advance, you can use the `reserve` method to set the initial bucket count:

Cpp

 Copiar

```
std::unordered_map<std::string, int> myMap;
myMap.reserve(100); // Reserves space for 100 elements
```

This helps avoid frequent reallocations and can enhance performance in some scenarios.

How does it handle the collision ?

`unordered_map` handles collisions using a technique called **separate chaining**. When two keys hash to the same bucket, they are stored in a list (or chain) within that bucket. Here's a simplified breakdown:

1. **Hashing:** Each key is hashed to determine which bucket it belongs to.
2. **Collision:** If another key hashes to the same bucket, a collision occurs.
3. **Chaining:** Both keys are stored in a list within that bucket.

For example:

- Keys `"key1"` and `"key2"` both hash to bucket 3.
- Bucket 3 will contain a list with both key-value pairs.

When you search for a key, `unordered_map` will hash the key, find the correct bucket, and then iterate through the list to find the exact key-value pair. Separate chaining ensures that even with collisions, all elements are accessible.

Pretty efficient system, huh?

So basically, the `unordered_map` library instantiates a hash table variable which has a built-in hash function, handles collision with a technique called separate chaining, and handles resizing as elements are added, right ?

Nailed it. `Unordered_map` uses built-in hash functions, manages collisions with separate chaining, and dynamically resizes the hash table as new elements get added. Clever, right? Keeps things both speedy and efficient. Anything else you're curious about in C++ or something else entirely?

- **Estructura Estadísticas:**

- **Goles a favor (`int goles_a_favor`):** Total de goles anotados por el equipo en la competición.
- **Goles en contra (`int goles_en_contra`):** Total de goles recibidos por el equipo en la competición.
- **Triunfos (`int triunfos`):** Cantidad de partidos ganados por el equipo en la competición.
- **Derrotas (`int derrotas`):** Cantidad de partidos perdidos por el equipo en la competición.
- **Empates (`int empates`):** Cantidad de partidos empatados por el equipo en la competición.

- **Funcionamiento:**

- **Construcción:** Este mapa se llena inicialmente al cargar y procesar todos los partidos. Cada vez que se procesa un partido, se incrementan los contadores de goles, victorias, derrotas o empates en las instancias de `Estadísticas` correspondientes.
- **Actualización:**

- Cuando se agrega un nuevo partido, se incrementan los valores de goles, triunfos, derrotas, o empates según corresponda.
 - Al eliminar un partido, se resta la contribución de dicho partido en `goles_a_favor`, `goles_en_contra`, y en el conteo de triunfos, derrotas o empates.
 - En una modificación de partido, se ajustan los valores previos y se suman los nuevos resultados.
 - **Uso:** Este mapa es clave en funciones de consulta que calculan y muestran estadísticas de rendimiento por equipo y competición, como:
 - **Comparación de goles a favor y en contra entre equipos.**
 - **Conteo de triunfos, derrotas y empates.**
 - **Cálculo de promedios de goles.**
 - **Consulta de fechas con mayor y menor cantidad de goles.**
-

2. Vector `partidos`

- **Definición:** `vector<unique_ptr<Partido>> partidos;`
 - Este vector contiene un conjunto de punteros únicos (`unique_ptr`) a objetos `Partido`, cada uno representando un partido individual en la base de datos.
- **Estructura `Partido`:**
 - **Fecha** (`string fecha`): Fecha en que se jugó el partido.
 - **Equipo local** (`string equipo_local`): Nombre del equipo que jugó como local.
 - **Goles local** (`int goles_local`): Cantidad de goles anotados por el equipo local.
 - **Equipo visitante** (`string equipo_visitante`): Nombre del equipo visitante.
 - **Goles visitante** (`int goles_visitante`): Cantidad de goles anotados por el equipo visitante.
 - **Competición** (`string competicion`): Nombre de la competición en la que se jugó el partido.
- **Funcionamiento:**
 - **Construcción:** El vector `partidos` se inicializa y carga al leer los datos del archivo CSV. Cada fila del archivo representa un partido y se convierte en un objeto `Partido`.
 - **Actualización:**
 - **Agregar:** Al agregar un nuevo partido, se crea un nuevo objeto `Partido` y se inserta en el vector.
 - **Eliminar:** Al eliminar un partido, el puntero correspondiente se elimina del vector.
 - **Modificar:** Para modificar un partido, se actualizan los atributos del objeto `Partido` en la posición correspondiente del vector.

- **Uso:**
 - El vector `partidos` es el recurso principal para consultas directas sobre los partidos, como obtener todos los partidos entre dos fechas, consultar enfrentamientos directos entre equipos, o filtrar partidos en una competición específica.
 - Al procesar el vector, se generan las estadísticas que luego se almacenan en `estadisticas`, `goles_por_competicion` y `goles_totales_por_competicion`.
-

3. Tabla de Hash `goles_por_competicion`

- **Definición:** `unordered_map<string, multimap<int, Partido *>> goles_por_competicion;`
 - Este mapa utiliza como clave el nombre de una competición (`string`) y como valor un `multimap` donde la clave es el total de goles en un partido y el valor es un puntero al partido (`Partido *`).
- **RELACIÓN CONCEPTUAL CON EL CURSO**

Sure! The `std::multimap` is an associative container in the C++ Standard Library that allows multiple elements to have equivalent keys ¹. It's part of the `<map>` header ¹. Here are some key points about `std::multimap`:

Key Features:

- **Associative Container:** Stores key-value pairs ².
- **Multiple Keys:** Allows multiple entries with the same key ².
- **Sorted by Key:** Keys are sorted according to a comparison function ².
- **Logarithmic Complexity:** Search, insertion, and removal operations have logarithmic complexity ².

Basic Usage:

To use `std::multimap`, you need to include the `<map>` header ¹. Here's a simple example:

```
Cpp Copiar

#include <iostream>
#include <map>

int main() {
    std::multimap<int, std::string> myMultimap;
    myMultimap.insert({1, "One"});
    myMultimap.insert({2, "Two"});
    myMultimap.insert({1, "Uno"});

    for (const auto& pair : myMultimap) {
        std::cout << pair.first << " - " << pair.second << std::endl;
    }

    return 0;
}
```

○

Common Methods:

- **insert()**: Inserts one or more key-value pairs ¹.
- **count()**: Returns the number of elements with a specific key ¹.
- **find()**: Returns an iterator to the first element with a specific key ¹.
- **erase()**: Removes elements with a specific key ¹.
- **clear()**: Removes all elements ¹.

Would you like more details on any specific method or usage?

What is an associative container?

An associative container in C++ is a type of container that allows for efficient retrieval of elements based on keys. Unlike sequence containers (like `vector` or `list`) where elements are accessed by their position, associative containers allow access through a key, making them ideal for lookups.

Key features of associative containers:

- **Key-Value Storage**: Elements are stored as key-value pairs.
- **Automatic Sorting**: Keys are automatically sorted (in containers like `map` and `set`).
- **Efficient Lookup**: Offers fast retrieval based on keys (usually logarithmic complexity).

Common associative containers:

- **map**: Stores unique key-value pairs with sorted keys.
- **multimap**: Similar to `map` but allows multiple elements with the same key.
- **set**: Stores unique keys in a sorted order.
- **multiset**: Similar to `set` but allows multiple elements with the same key.
- **unordered_map** and **unordered_set**: Use hash tables for faster (average $O(1)$) lookups and are not sorted.

Does that clarify things? Anything specific you want to dive deeper into?

What data structure would you classify an associative container into?

Associative containers are primarily classified as **tree-based** or **hash-based** data structures:

1. **Tree-Based**: These containers (like `std::map` and `std::set`) are typically implemented using balanced trees such as Red-Black Trees. They maintain order among elements, providing sorted data and allowing for efficient operations like insertion, deletion, and lookup, all in $O(\log n)$ time.
2. **Hash-Based**: Containers such as `std::unordered_map` and `std::unordered_set` use hash tables to store elements. They provide average $O(1)$ time complexity for insertions, deletions, and lookups, but the elements are not stored in any specific order.

So, an associative container can be classified into tree-based or hash-based structures. Cool, right?

So will multimap be classified into a Hash table?

Nope, `std::multimap` isn't hash-based. It's actually implemented using a balanced tree structure, like a Red-Black Tree. This ensures that the elements are stored in a sorted order and allows for efficient insertions, deletions, and lookups—all in logarithmic time. If you're looking for a hash-based version with multiple keys, you'd typically use `std::unordered_multimap`. Cool, right?

Could you delve deeper into Red-Black trees?

Red-Black trees are a type of self-balancing binary search tree, which ensures that the tree remains approximately balanced, allowing for efficient operations such as insertion, deletion, and lookup. Here's a deeper dive into their characteristics and properties:

Key Properties:

1. **Nodes are Red or Black:** Each node in the tree is either red or black.
2. **Root is Always Black:** The root of the tree is always black.
3. **Red Nodes Can't Have Red Children:** If a node is red, its children must be black (no two red nodes can be adjacent).
4. **Every Path from a Node to its Descendant NULL Nodes must have the Same Number of Black Nodes:** This ensures a roughly balanced tree structure.

Key Operations:

- **Insertion:** When a new node is inserted, it's initially colored red. The tree is then adjusted to maintain balance and the red-black properties. This might involve rotations and color changes.
- **Deletion:** When a node is deleted, the tree is again adjusted to maintain balance, which might involve multiple steps including rotations and color changes.
- **Rotations:** These are used to maintain balance:
 - **Left Rotation:** Shifts nodes to the left.
 - **Right Rotation:** Shifts nodes to the right.

Benefits:

- **Balanced:** Ensures the tree height is $O(\log n)$, which provides efficient insertion, deletion, and lookup operations.
- **Complexity:** Most operations are performed in $O(\log n)$ time due to the balancing properties.

-
- **Funcionamiento:**
 - **Construcción:** Al procesar el archivo de datos, cada partido se inserta en el `multimap` correspondiente de su competición, con la clave de goles totales

(suma de goles local y visitante). Esto permite que los partidos se ordenen automáticamente por cantidad de goles.

- **Actualización:**
 - **Agregar:** Un nuevo partido se inserta en el `multimap` de su competición con el total de goles como clave.
 - **Eliminar:** Cuando se elimina un partido, este se borra del `multimap` de la competición correspondiente.
 - **Modificar:** Para una modificación, se elimina el partido con los datos antiguos y se reinsertan los datos actualizados.
 - **Uso:**
 - **Top 5 partidos con más goles por competición:** Al estar ordenado, permite obtener rápidamente los partidos con más goles para cualquier competición.
 - **Consultas sobre rendimiento:** Almacena los datos de manera ordenada por goles, facilitando consultas sobre partidos con alta o baja puntuación en cada competición.
-

4. Tabla de Hash `goles_totales_por_competicion`

- **Definición:** `unordered_map<string, int>`
`goles_totales_por_competicion;`
 - Este mapa utiliza como clave el nombre de la competición (`string`) y como valor un entero que representa el total de goles acumulados en esa competición.
- **Funcionamiento:**
 - **Construcción:** Al procesar el vector `partidos`, los goles de cada partido se suman en el valor correspondiente de `goles_totales_por_competicion`.
 - **Actualización:**
 - **Agregar:** Al añadir un partido, su total de goles se suma al total de la competición.
 - **Eliminar:** Al eliminar un partido, se resta su total de goles del total de la competición.
 - **Modificar:** Al modificar un partido, se ajusta el total de goles restando los goles anteriores y sumando los nuevos.
 - **Uso:**
 - **Competición con mayor cantidad de goles:** Este mapa permite identificar rápidamente cuál es la competición con el total más alto de goles.
 - **Análisis global de rendimiento:** Almacenar el total de goles por competición es útil para evaluaciones globales o análisis comparativos entre competiciones.