Nivel Básico: STL

Instructor: Leonardo Valverde Lara

Febrero 13, 2024

§1. Vector

Es un arreglo dinámico que permite el acceso como en un arreglo común de C++ (usando el operador `[]`), sin embargo también tiene muchas otras propiedades que se detallarán a continuación.

§1.1. Declaración y Constructor

Para declarar un vector se debe añadir la librería `vector` y se debe utilizar la sentencia `vector<T>` donde T es el tipo de dato que almacenará dicho vector y para inicializarlo la librería provee distintos modos de hacerlo.

```cpp
#include <vector>
...
vector<int> v1; // Declara un vector sin elementos
vector<int> v2 = {1,2,3,1}; // Inicializa valores en el vector (a partir de C++11)
vector<int> v3(10); // Declara un vector con 10 elementos (todos son 0 por defecto)
vector<int> v4(8, 3); // Declara un vector con 8 elementos teniendo todos valor 3
```

§1.2. Iteradores

- `begin`: Retorna un iterador al primer elemento del vector.

```cpp
vector<char> v = {'a', 'b', 'd', 'h'};
vector<char>::iterator inicio = v.begin(); // Iterador que apunta al inicio
cout << *inicio; // Muestra el elemento del inicio 'a'
```

- `end`: Retorna un iterador al final del vector que se encuentra fuera del mismo.

```cpp
vector<char> v = {'a', 'b', 'd', 'h'};
vector<char>::iterator fin = v.end(); // Iterador que apunta al final
fin = prev(fin); // Retrocedemos una posicion
cout << *fin; // Muestra el ultimo elemento dentro del vector 'h'
```

- `rbegin`: Retorna un iterador al último elemento del vector (iterar en sentido inverso).

```cpp
vector<char> v = {'a', 'b', 'd', 'h'};
vector<char>::reverse_iterator fin = v.rbegin(); // Apunta al ultimo
cout << *fin; // Muestra el ultimo elemento dentro del vector 'h'
```

- `rend`: Retorna un iterador al inicio del vector que se encuentra fuera del mismo.

```cpp
vector<char> v = {'a', 'b', 'd', 'h'};
vector<char>::reverse_iterator inicio = v.rend(); // Apunta al inicio
inicio = prev(inicio); // Retrocedemos una posicion
cout << *inicio; // Muestra el primer elemento 'a'
```

§1.3. Capacidad

- `size`: Retorna el tamaño actual del vector.

```
vector<char> v = {'a', 'b', 'd', 'h'};
cout << v.size(); // Muestra 4
```

- `resize`: Cambia el tamaño actual del vector

```
vector<int> v = {1, 5, 8, 2};
vector<int> v1 = v; // Copiamos el arreglo v a v1
v1.resize(8); // v1 = {1, 5, 8, 2, 0, 0, 0, 0}

vector<int> v2 = v; // Copiamos el arreglo v a v2
v2.resize(2); // v2 = {1, 5}
```

- empty: Retorna un valor booleano identificando si el vector está vacío o no.

```
vector<int> v;
if(v.empty()) cout << "Vacio";
```

§1.4. Acceso

- `operator[]`: Accede a una posición específica como en un vector.

```
vector<int> v = {4, 9, 12, 31, 8};
cout << v[3]; // Muestra 31
```

- `front`: Retorna la primera posición del vector.

```
vector<int> v = {4, 9, 12, 31, 8};
cout << v.front(); // Muestra 4
```

- `back`: Retorna la útlima posición del vector.

```
vector<int> v = {4, 9, 12, 31, 8};
cout << v.back(); // Muestra 8
```

§1.5. Modificadores

- `push_back`: Añade un elemento al final del vector.

```
vector<int> v = {4, 9, 12, 31, 8};
cout << v.push_back(6); // v = {4, 9, 12, 31, 8, 6}
```

- `pop_back`: Elimina el elemento al final del vector.

```
vector<int> v = {4, 9, 12, 31, 8};
cout << v.pop_back(); // v = {4, 9, 12, 31}
```

- `clear`: Limpia el vector

```
vector<int> v = {4, 9, 12, 31, 8};
cout << v.clear(); // v = {}
```

§2. Queue

Conocida como «cola» es una estructura más ligera que el vector pero que solo funciona en modo FIFO (first-in first-out), esto significa que el primer elemento que fue insertado en la cola será el primero en ser eliminado. Uno de los usos más conocidos de esta estructura es en el algoritmo BFS el cual se aplica a grafos.

§2.1. Declaración

Para declarar una cola se debe añadir la librería `queue` y se debe utilizar la sentencia `queue<T>` donde T es el tipo de dato que almacenará la cola.

```
#include <queue>
...
queue<int> q; // Declara un queue vacio de enteros
```

§2.2. Capacidad

- `size`: Retorna el tamaño actual de la cola.

```
queue<string> q; // q = {"casa", "perro", "arbol", "vaso"};
cout << q.size(); // Muestra 4
```

- `empty`: Retorna un valor booleano identificando si el vector está vacío o no.

```
queue<string> q;
if(q.empty()) cout << "Vacio";
```

§2.3. Acceso

- `front`: Retorna la primera posición de la cola.

```
queue<string> q; // q = {"casa", "perro", "arbol", "vaso"};
cout << q.front(); // Muestra "casa"
```

- `back`: Retorna la última posición de la cola.

```
queue<string> q; // q = {"casa", "perro", "arbol", "vaso"};
cout << q.back(); // Muestra "vaso"
```

§2.4. Modificadores

- `push`: Inserta un elemento al final de la cola

```
queue<string> q; // q = {"casa", "perro", "arbol"};
cout << q.push("cubo"); // q = {"casa", "perro", "arbol", "cubo"};
```

- `pop`: Elima el primer elemento de la cola

```
queue<string> q; // q = {"casa", "perro", "arbol"};
cout << q.pop; // q = {"casa", "perro"};
```

§3. Stack

Conocida como «pila», es una estructura de datos que sigue el principio de "último en entrar, primero en salir"(LIFO). Esto significa que el último elemento que se inserta en el stack es el primero en ser eliminado. Entre sus usos más básicos se encuentran la notación posfija, y verificar si una secuencia de brackets está balanceada.

§3.1. Declaración

Para usar un stack en C++, se debe incluir la librería `stack` y se puede declarar utilizando la sentencia `stack<T>` donde T es el tipo de dato que se almacenará en el stack.

```
#include <stack>
...
stack<int> s; // Declara un stack vacio de enteros
```

§3.2. Capacidad

- `size`: Devuelve el número de elementos en el stack.

```
stack<int> s; // s = {4, 2, 0, 4, 7}
cout << s.size(); // Muestra 5
```

- `empty`: Devuelve verdadero si el stack está vacío, falso de lo contrario.

```
stack<int> s;
if (s.empty()) cout << "Vacio";
```

§3.3. Acceso

- `top`: Devuelve el elemento en la parte superior del stack.

```
stack<int> s; // s = {4, 2, 0, 4, 7} (El 7 es la cima)
cout << s.top(); // Muestra el valor 7 sin eliminarlo del stack
```

§3.4. Modificadores

- `push`: Inserta un elemento en la parte superior del stack.

```
    stack<int> s; // s = {4, 2, 0, 4, 7}
    s.push(5); // s = {4, 2, 0, 4, 7, 5} (5 es la nueva cima)
```

- `pop`: Elimina el elemento en la parte superior del stack.

```
    stack<int> s; // s = {4, 2, 0, 4, 7}
    s.pop(); // Elimina el 7, ahora el 4 de la derecha es la cima
```

§4. Deque

Es una estructura de datos que combina las características de las colas (queue) y pilas (stack). Permite la inserción y eliminación eficiente tanto al principio como al final de la secuencia, además también permite el acceso como un `vector`.

Esta estructura tiene casi las mismas funciones que un `vector`, sin embargo puede insertar/eliminar elementos no solo al final sino también al principio, sin embargo no se garantiza que los elementos se ubiquen de modo contiguo y tiene mayor coste que un vector normal.

Debido a su similitud con `vector` solo se mostrará lo distintivo de esta estructura.

§4.1. Declaración

Para usar un deque en C++, se debe incluir la librería `deque` y se puede declarar utilizando la sentencia `deque<T>` donde T es el tipo de dato que se almacenará en el deque.

```
#include <deque>
...
deque<int> d; // Declara un deque vacio de enteros
```

§4.2. Métodos Principales

- `push_back`: Inserta un elemento al final del deque.

```
deque<int> d;
d.push_back(5); // Inserta el valor 5 al final del deque
```

- `push_front`: Inserta un elemento al principio del deque.

```
deque<int> d;
d.push_front(10); // Inserta el valor 10 al principio del deque
```

- `pop_back`: Elimina el último elemento del deque.

```
deque<int> d;
d.push_back(5);
d.pop_back(); // Elimina el ultimo elemento (5) del deque
```

- `pop_front`: Elimina el primer elemento del deque.

```
deque<int> d;
d.push_front(10);
d.pop_front(); // Elimina el primer elemento (10) del deque
```

§5. Priority Queue

Es una estructura de datos que organiza sus elementos según un orden especificado. En un `priority_queue`, el elemento de mayor prioridad siempre se encuentra en la parte superior, y los elementos con menor prioridad se encuentran en posiciones inferiores.

§5.1. Declaración

Para usar un `priority_queue` en C++, se debe incluir la librería `queue` y se puede declarar utilizando la sentencia `priority_queue<T>` donde T es el tipo de dato que se almacenará en la cola de prioridad. Uno de los usos más conocidos de esta estructura es en el algoritmo **Dijkstra**.

```cpp
#include <queue>
...
priority_queue<int> pq; // Declara una cola de prioridad vacia de enteros
```

§5.2. Métodos Principales

Las funciones son las mismas que la estructura `queue`, a continuación se muestran algunas a modo de ejemplo.

- `push`: Inserta un elemento en la cola de prioridad. $O(\log N)$.

```cpp
priority_queue<int> pq;
pq.push(5); // Inserta el valor 5 en la cola de prioridad
```

- `pop`: Elimina el elemento de mayor prioridad de la cola de prioridad. $O(\log N)$.

```cpp
priority_queue<int> pq;
pq.push(5);
pq.push(2);
pq.push(7);
pq.pop(); // Elimina el elemento de mayor prioridad (7)
```

- `top`: Devuelve una referencia al elemento de mayor prioridad sin eliminarlo.

```cpp
priority_queue<int> pq;
pq.push(5);
pq.push(2);
pq.push(7);
cout << pq.top(); // Muestra el elemento de mayor prioridad (7)
```

- `empty`: Devuelve verdadero si la cola de prioridad está vacía, falso de lo contrario.

```cpp
priority_queue<int> pq;
if (pq.empty()) cout << "Vacia";
```

§5.3. Variante Ascendente

Como se mencionó previamente esta estructura coloca a los que tienen mayor prioridad en la cima. En los enteros son los que tengan mayor valor, en los `string` coloca al mayor lexicográficamente. Para establecer que se ordenen de modo ascendente se requiere la siguiente modificación: `priority_queue<T, vector<T>, greater<T>>`, donde T es el tipo de dato.

```cpp
// Declaracion de un priority_queue para que funcione ascendentemente
priority_queue<int, vector<int>, greater<int>> pq;
```

§6. Set

Es una estructura de datos que almacena elementos **únicos** en orden ascendente. Cada elemento en un `set` debe ser único y los elementos se almacenan ordenados automáticamente de modo ascendente. Está basado en un árbol de búsqueda binaria, que permite realizar operaciones como las que mostrarán a continuación.

§6.1. Declaración

Para usar un set en C++, se debe incluir la librería `set` y se puede declarar utilizando la sentencia `set<T>` donde T es el tipo de dato que se almacenará en el set.

```
#include <set>
...
set<int> s1; // Declara un set vacio de enteros
set<int> s2 = {1,2,3,1}; // Inicializa valores en el set (duplicados se eliminan)
```

§6.2. Iteradores

- `begin`: Retorna un iterador al primer elemento del set.

```
set<char> s = {'h', 'm', 'c', 'z'};
set<char>::iterator inicio = s.begin(); // Iterador que apunta al inicio
cout << *inicio; // Muestra el elemento del inicio 'c'
```

- `end`: Retorna un iterador al final del set que se encuentra fuera del mismo.

```
set<char> s = {'h', 'm', 'c', 'z'};
set<char>::iterator fin = s.end(); // Iterador que apunta al final
fin = prev(fin); // Retrocedemos una posicion
cout << *fin; // Muestra el ultimo elemento dentro del set 'z'
```

- `rbegin`: Retorna un iterador al último elemento del set (iterar en sentido inverso).

```
set<char> s = {'h', 'm', 'c', 'z'};
set<char>::reverse_iterator fin = s.rbegin(); // Apunta al ultimo
cout << *fin; // Muestra el ultimo elemento dentro del set 'z'
```

- `rend`: Retorna un iterador al inicio del set que se encuentra fuera del mismo.

```
set<char> s = {'h', 'm', 'c', 'z'};
set<char>::reverse_iterator inicio = s.rend(); // Apunta al inicio
inicio = prev(inicio); // Retrocedemos una posicion
cout << *inicio; // Muestra el primer elemento 'c'
```

§6.3. Capacidad

- `size`: Retorna el tamaño actual del set.

```
set<int> s = {1, 3, 4, 1};
cout << s.size(); // Muestra 3;
```

- `empty`: Retorna un valor booleano identificando si el vector está vacío o no.

```
set<int> s;
if (s.empty()) cout << "vacio";
```

§6.4. Acceso

El set al no tener índices, no es posible acceder por posición, pero sí usando iteradores.

- `begin`: Retorna un iterador al menor elemento del set.

```
set<int> s = {1, 3, 4, 1};
cout << *s.begin(); // Muestra 1;
```

- `rbegin`: Retorna un iterador al mayor elemento del set.

```
set<int> s = {1, 3, 4, 1};
cout << *s.rbegin(); // Muestra 4;
```

- `find`: Retorna un iterador al elemento que se pase como parámetro. Si el valor no existe retorna `end`. $O(\log N)$.

```
set<int> s = {1, 3, 4, 1};
set<int>::iterator it = s.find(3);
if(it != s.end()) cout << *it;
else cout << "El valor no existe.";
```

- `count`: Retorna el número de ocurrencias de un elemento en el set (1 si está o 0 si no está). $O(\log N)$.

```
set<int> s = {4, 9, 12, 31, 12};
cout << s.count(12); // Muestra 1
cout << s.count(5); // Muestra 0
```

§6.5. Modificadores

- `insert`: Inserta un elemento en el set. Si ya está presente simplemente se omite. $O(\log N)$.

```
set<int> s = {1, 3, 4, 1};
s.insert(10); // s = {1, 3, 4, 10}
```

- `erase`: Elimina un elemento en el set. Si no existe ese elemento simplemente se omite. $O(\log N)$.

```
set<int> s = {1, 3, 4, 1};
s.erase(1); // s = {3, 4}
```

- `clear`: Elimina todos los elementos del set.

```
set<int> s = {1, 3, 4, 1};
s.clear(); // s = {}
```

§6.6. Recorrer un set

- **Por iteradores**:

```
set<int> s = {1, 3, 4, 1};
for(set<int>::iterator it = s.begin(); it != s.end(); it++) {
    cout << *it << ' ';
}
```

- **Por elementos**:

```
set<int> s = {1, 3, 4, 1};
for(int x : s){
    cout << x << ' ';
}
```

§7. Multiset

Es una estructura de datos similar a `set`, la única diferencia es que el `multiset` puede almacenar datos duplicados, lo cual hace que ciertas funciones cambien.

§7.1. Declaración

Para usar un multiset en C++, se debe incluir la librería `set` y se puede declarar utilizando la sentencia `multiset<T>` donde T es el tipo de dato que se almacenará en el set.

```
#include <set>
...
multiset<int> s1; // Declara un multiset vacio de enteros
multiset<int> s2 = {1,2,3,1}; // Inicializa valores en el multiset (con duplicados)
```

§7.2. Métodos Principales

A continuación se mostrarán aquellas funciones que difieren de set.

- `erase`: A diferencia del set, la función erase borra **todas** las ocurrencias de un elemento. Si se desea borrar solo una ocurrencia entonces se debe eliminar un iterador en vez de un valor. $O(\log N)$.

```
multiset<int> ms = {1, 2, 2, 1, 3, 1, 2, 1};
ms.erase(2); // ms = {1, 1, 1, 1, 3};
ms.erase(ms.find(1)); // ms = {1, 1, 1, 3};
```

- `count`: Retorna la cantidad de ocurrencias de un determinado elemento en el multiset. $O(\log N)$.

```
multiset<int> ms = {1, 2, 2, 1, 3, 1, 2, 1};
cout << ms.count(2); // 3
cout << ms.count(1); // 4
cout << ms.count(5); // 0
```

§8. Map

Es una estructura que almacena pares de elementos clave-valor, donde cada clave está asociada a un **único** valor garantizando un orden en estas, lo cual permite un acceso eficiente a los elementos.

§8.1. Declaración

Para declarar un map en C++ se debe incluir la librería `map` y se declara utilizar la sentencia `map<K,V>` donde K es el tipo de dato que tendrá la clave y V es el tipo de dato del valor que se almacenarán en conjunto en dicho map y para inicializarlo se tienen en cuenta algunos de los siguientes modos.

```
map<int, int> m1;     // Declara un map sin elementos
map<int, int> m2 = {{4, 10}, {2, 2}, {21, 30}, {18, 4}};  // Inicializando valores
```

§8.2. Iteradores

- `begin`: Retorna un iterador al menor elemento del mapa.

```
map<char, int> mp = {{'a', 1}, {'b', 2}, {'d', 4}, {'h', 8}};
map<char, int>::iterator inicio = mp.begin();
cout << inicio->first << ' ' << inicio->second;  // Muestra "a 1"
```

- `end`: Retorna un iterador al final del mapa, que está fuera del mismo.

```
map<char, int> mp = {{'a', 1}, {'b', 2}, {'d', 4}, {'h', 8}};
std::map<char, int>::iterator fin = mp.end();
--fin; // Retrocedemos una posicion
cout << fin->first << ' ' << fin->second;  // Muestra "h 8"
```

- `rbegin`: Retorna un iterador al mayor elemento del mapa.

```
map<char, int> mp = {{'a', 1}, {'b', 2}, {'d', 4}, {'h', 8}};
map<char, int>::reverse_iterator fin = mp.rbegin();
cout << fin->first << ' ' << fin->second;  // Muestra "h 8"
```

- `rend`: Retorna un iterador al inicio del mapa, que está fuera del mismo.

```
map<char, int> mp = {{'a', 1}, {'b', 2}, {'d', 4}, {'h', 8}};
map<char, int>::reverse_iterator inicio = mp.rend();
inicio--; // Aumentamos una posicion
cout << inicio->first << ' ' << inicio->second;  // Muestra "a 1"
```

§8.3. Capacidad

- `size`: Retorna la cantidad de elementos en el mapa.

```
map<char, int> mp = {{'a', 1}, {'b', 2}, {'d', 4}, {'h', 8}};
cout << mp.size(); // Muestra 4
```

- `empty`: Retorna true si el mapa está vacío o false en caso contrario.

```
map<int, int> mp;
if(mp.empty()) cout<<"Vacio";
```

§8.4. Acceso

- `operator[]`: Accede al valor de una clave en específico. $O(\log N)$.

```
map<char, int> mp = {{'a', 1}, {'b', 2}, {'d', 4}, {'h', 8}};
cout << mp['d']; // Muestra el valor 4
```

- `find`: Retorna un iterador al elemento que se pase como parámetro. Si el valor no existe retorna `end`. $O(\log N)$.

```
map<char, int> mp = {{'a', 1}, {'b', 2}, {'d', 4}, {'h', 8}};
map<char, int>::iterator it = mp.find('b');
if(it != mp.end()) cout << it->first << ' ' << it->second;
else cout << "El valor no existe";
```

- `count`: Retorna el número de ocurrencias de una clave en el mapa (1 si está o 0 si no está). $O(\log N)$.

```
map<char, int> mp = {{'a', 1}, {'b', 2}, {'d', 4}, {'h', 8}};
cout << mp.count('a'); // Muestra 1
cout << mp.count('x'); // Muestra 0
```

§8.5. Modificadores

- `operator[]`: Para insertar basta con asignar un valor a una clave del mapa. $O(\log N)$.

```
map<int, int> mp; // Se declara un mapa vacio
mp[1] = 1;
mp[2] = 4;
cout << mp.size(); // Muestra 2
```

- `erase`: Elimina una clave en específico del mapa. $O(\log N)$.

```
map<char, int> mp = {{'a', 1}, {'b', 2}, {'d', 4}, {'h', 8}};
mp.erase('a') ; // mp = {{'b', 2}, {'d', 4}, {'h', 8}};
```

- `clear`: Elimina todos los elementos del mapa.

```
map<char, int> mp = {{'a', 1}, {'b', 2}, {'d', 4}, {'h', 8}};
mp.clear() ; // mp = {}
```

§8.6. Recorrer un mapa

- **Por iteradores**:

```
map<int, int> mp = {{1, 10}, {2, 20}, {3, 30}, {4, 40}};
for (map<int, int>::iterator it = mp.begin(); it != mp.end(); ++it) {
    cout << "Clave: " << it->first << ", Valor: " << it->second << endl;
}
```

- **Por elementos**:

```
map<int, int> mp = {{1, 10}, {2, 20}, {3, 30}, {4, 40}};
for (pair<int, int> p : mp) {
    cout << "Clave: " << p.first << ", Valor: " << p.second << endl;
}
```

§9. Multimap

Al igual que multiset, la única diferencia entre map con multimap es que este último puede almacenar datos duplicados, lo cual hace que ciertas funciones cambien.

§9.1. Declaración

Para usar multimap en C++ se debe incluir la librería `map` y se declara utilizando la sentencia `multimap<T>` donde T es el tipo de dato que se almacenará en el set.

```
#include <map>
...
multimap<int, int> mp; // Declara un multimap de enteros vacio
multimap<int, int> mp = {{1, 10}, {2, 20}, {1, 30}}; // Inicializa valores sin
    borrar duplicados
```

§9.2. Métodos Principales

Las funciones que contiene `multimap` son las mismas que `map`, si embargo el comportamiento de las siguientes funciones difieren.

- `erase`: A diferencia del map, la función erase borra **todas** las ocurrencias de una clave. Si se desea borrar solo una ocurrencia entonces se debe eliminar un iterador en vez de una clave, sin embargo el iterador que se obtiene usando `find` es aquel con dicha clave pero en el valor se considera el mínimo. $O(\log N)$.

```
multimap<int, int> mp = {{1, 30}, {2, 20}, {1, 10}, {1, 50}, {2, 30}, {3, 1}};
mp.erase(2); // mp = {{1, 10}, {1, 30}, {1, 50}, {3, 1}}
mp.erase(mp.find(1)); // mp.find(1) = {1, 10}, mp = {{1, 30}, {1, 50}, {3, 1}}
```

- `count`: Retorna la cantidad de ocurrencias de un determinada clave en el map. $O(\log N)$.

```
multimap<int, int> mp = {{1, 30}, {2, 20}, {1, 10}, {1, 50}, {2, 30}, {3, 1}};
cout << mp.count(1); // 3
cout << mp.count(3); // 1
cout << mp.count(5); // 0
```