

C++ STL

Ciencias de la Computación e Inteligencia Artificial

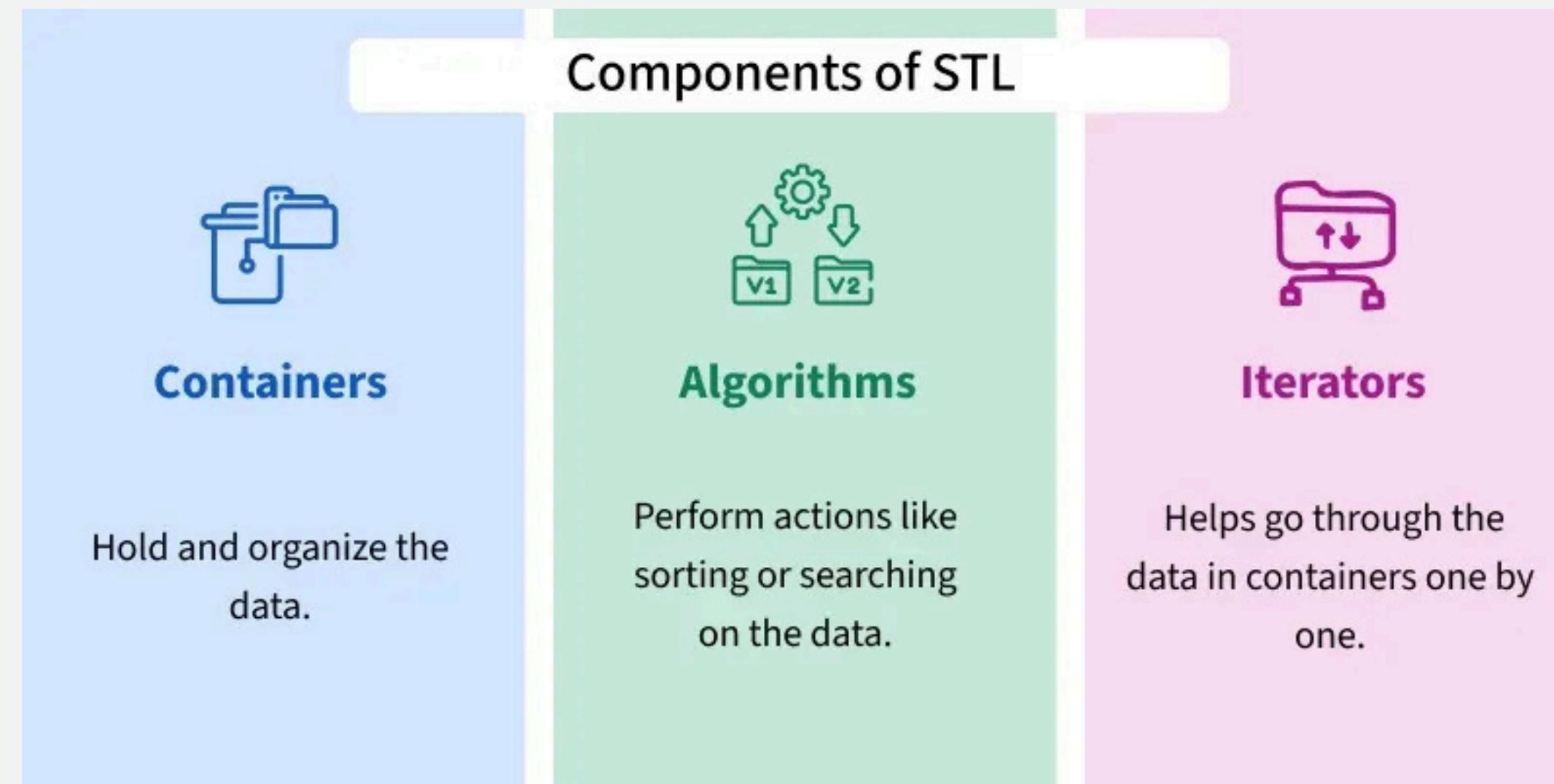
María Jesús Rodríguez Sánchez



STL C++

¿Qué es?

Conjunto de estructuras de datos genéricas. Diseñada para ser eficiente, segura y reutilizable.

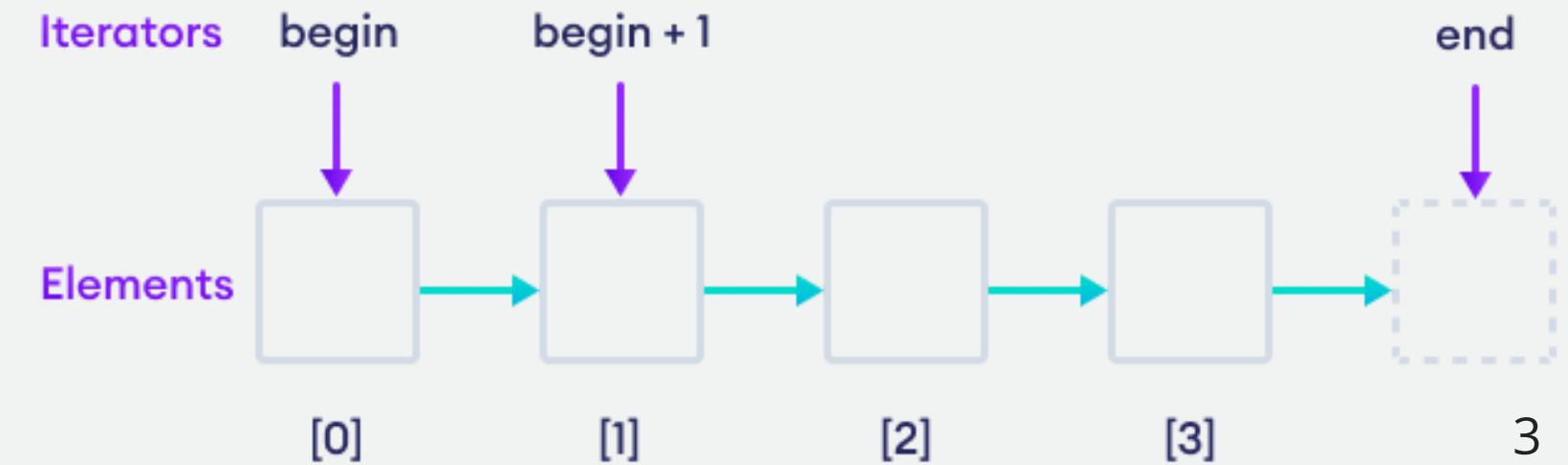


Iteradores

Un iterador es un objeto que permite recorrer un contenedor elemento a elemento.

Funciona de forma parecida a un puntero, pero es más seguro y funciona con cualquier contenedor de la STL

Acceder al primer elemento → begin()
Acceder a la “posición final” → end()
Avanzar al siguiente → ++it
Acceder al valor → *it



Iteradores

```
vector<int>::iterator it;  
vector<int>::const_iterator cit;
```

Si solo vas a leer hay que usar
const_iterator

```
typedef map<string, int> StringIntMap;  
StringIntMap::iterator it;  
StringIntMap::const_iterator cit;
```

it->first → clave (**siempre const**)

it->second → valor (modificable si
no es const_iterator)

Contenedores

- Un contenedor es una colección de objetos dotado de un conjunto de métodos para gestionarlos (acceder a ellos, eliminarlos, añadir nuevos objetos, buscar, etc.).
- Ofrecen también “herramientas” para recorrerlos (iterar) y revisar los objetos almacenados en ellos.
- Los contenedores se pueden clasificar según distintos criterios, pero el fundamental es la forma de organización:

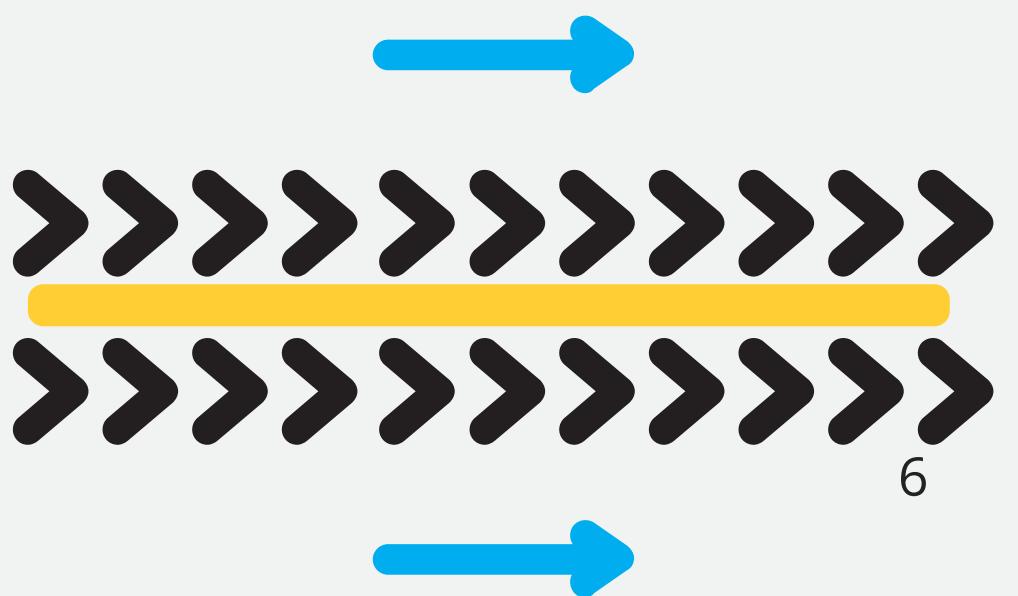
Lineal, jerárquica, interconexión total

Contenedores

También se diferencian por las formas de acceder a los componentes:

- Secuencial
- Directa
- Por clave (también conocidos como asociativos)

Existe un conjunto de **operaciones comunes a todos los contenedores**,
y operaciones adicionales atendiendo a peculiaridades particulares.

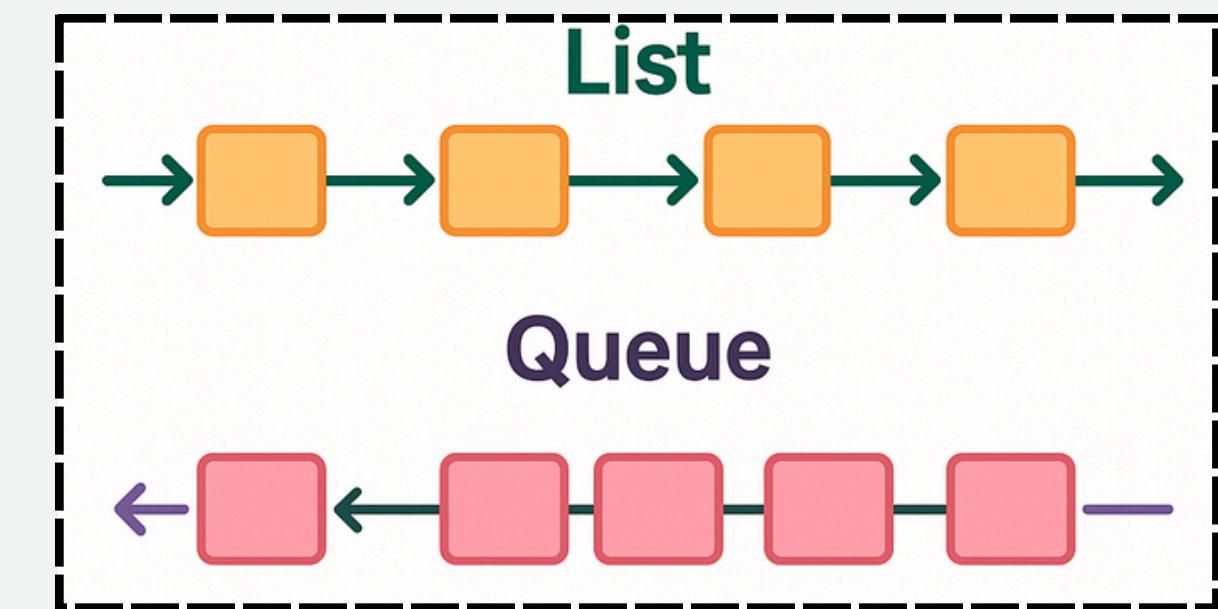


Contenedores Secuenciales

**Recorremos los elementos uno detrás de otro,
desde el primero al último.**

Contenedores secuenciales típicos

- `list` → lista doblemente enlazada
- `forward_list` → lista enlazada simple
- `queue`, `stack` → acceso restringido (FIFO / LIFO)
- `priority_queue` → secuencia interna no accesible directamente
- `deque`



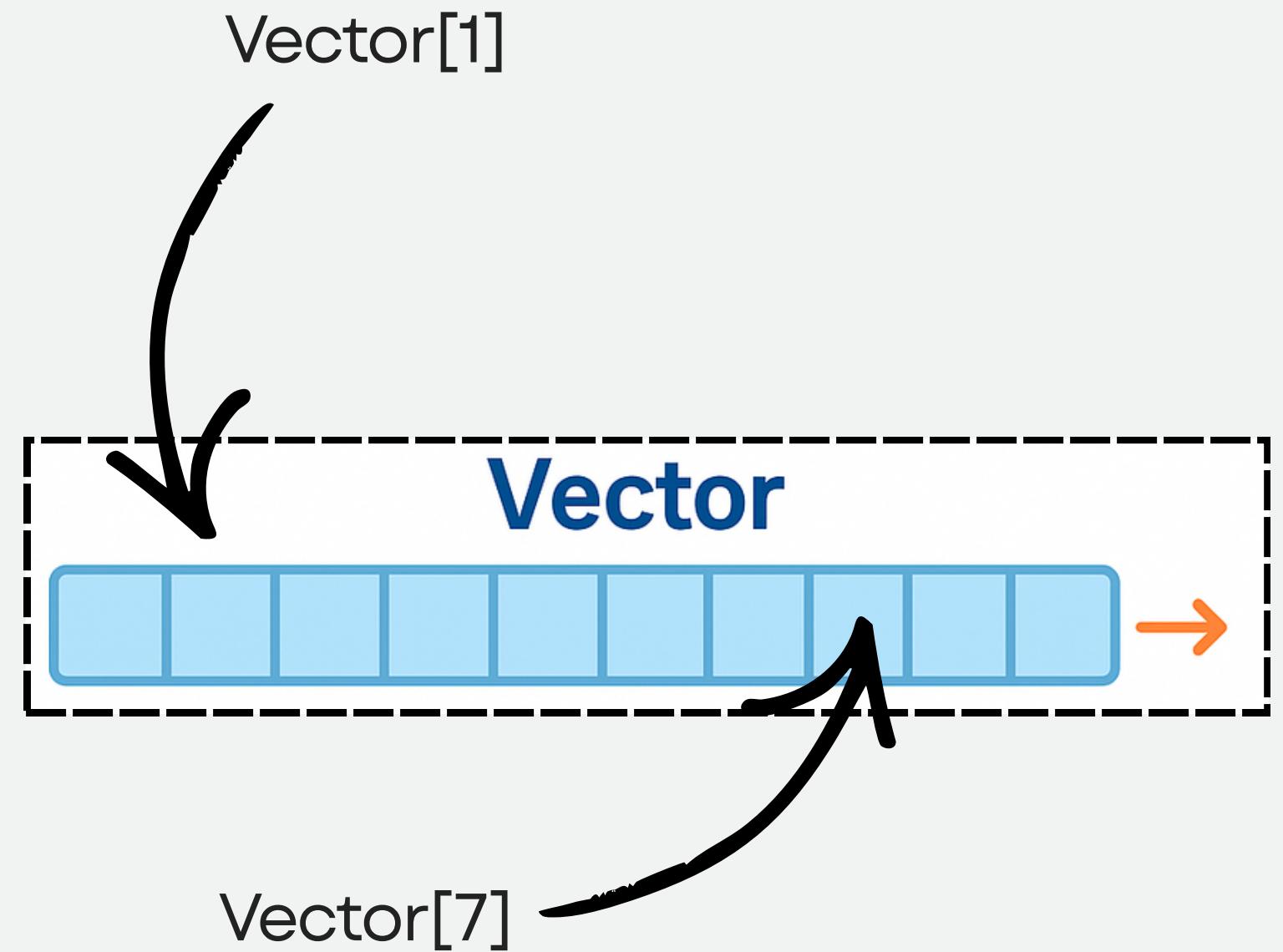
Contenedores de Acceso Directo

Permite acceder a un elemento por su posición, sin recorrer los anteriores.

Es decir, **acceso por índice en $O(1)$.**

Contenedores con acceso directo

- vector
- deque (aunque menos eficiente internamente, puedes acceder como si fuera un array)

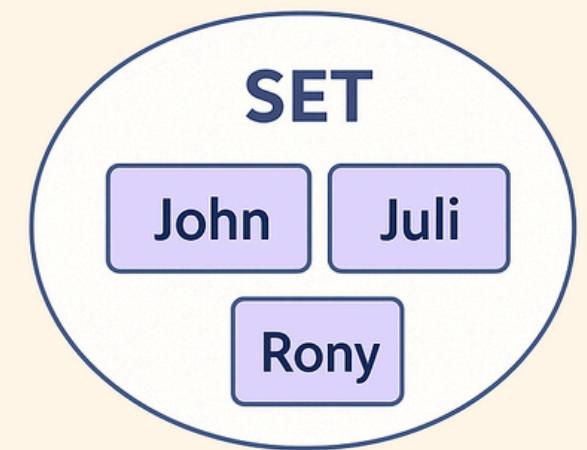
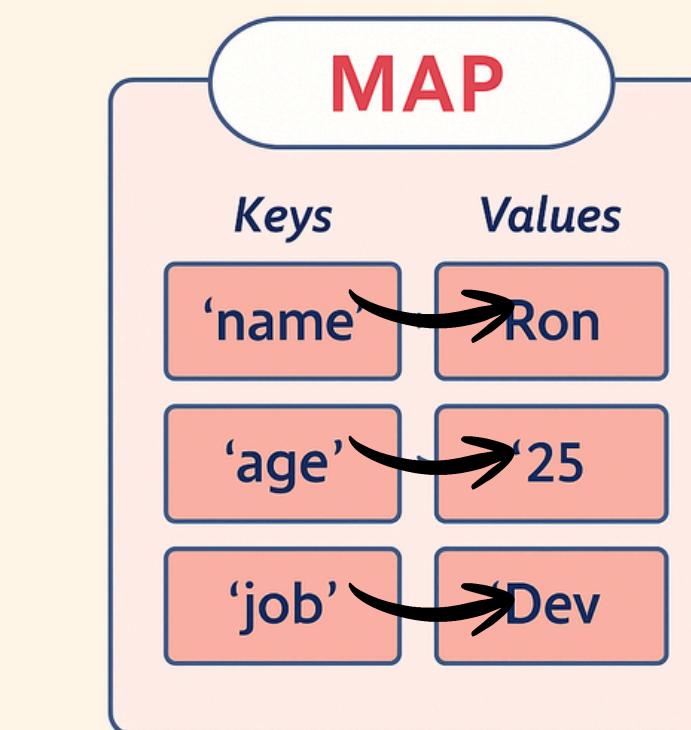


Contenedores Asociativos

Accedemos a los elementos usando una clave que los identifica.

El contenedor mantiene los elementos ordenados automáticamente.

- set (solo claves, sin duplicados)
- multiset (solo claves, con duplicados)
- map (clave → valor, sin duplicados)
- multimap (clave → valor, duplicados)



Contenedores - comparativa

Tipo	Orden	Acceso	Duplicados	Ejemplo
vector	Índices	O(1)	Sí	Lista de notas
list	Secuencial	O(n)	Sí	Playlist
deque	Ambos extremos	O(1)	Sí	Buffer circular
set	Ordenado	Búsqueda O(log n)	No	Diccionario ortográfico
map	Ordenado	Búsqueda por clave	No	Agenda telefónica

Contenedores

Operaciones comunes

- **begin() / end()** → Devuelven iteradores al primer elemento y a la “posición final” (una más allá del último).
- **empty()** → Indica si el contenedor está vacío.
- **size()** → Devuelve cuántos elementos contiene.
- **swap()** → Intercambia en O(1) el contenido con otro contenedor del mismo tipo.
- **Constructores / copia / asignación** → Permiten crear contenedores nuevos, copiarlos o reemplazar su contenido.

Pila



#include <stack>

Una pila es un contenedor secuencial que restringe el acceso, tanto para recuperar elementos existentes, como para insertar nuevos, a un extremo de la secuencia.

- Al extremo donde se llevan a cabo las inserciones como las recuperaciones se denomina “tope de la pila”.
- Los elementos que se insertan se van “apilando” unos encima de otros (el último insertado se dispone encima del penúltimo, y así sucesivamente).



Pila



#include <stack>

El último elemento en ser introducido en ella será el primero en ser recuperado.

```
#include <iostream>
#include <stack>
using namespace std;
int main()
{
    stack<int> st;
    st.push(10);
    st.push(5);

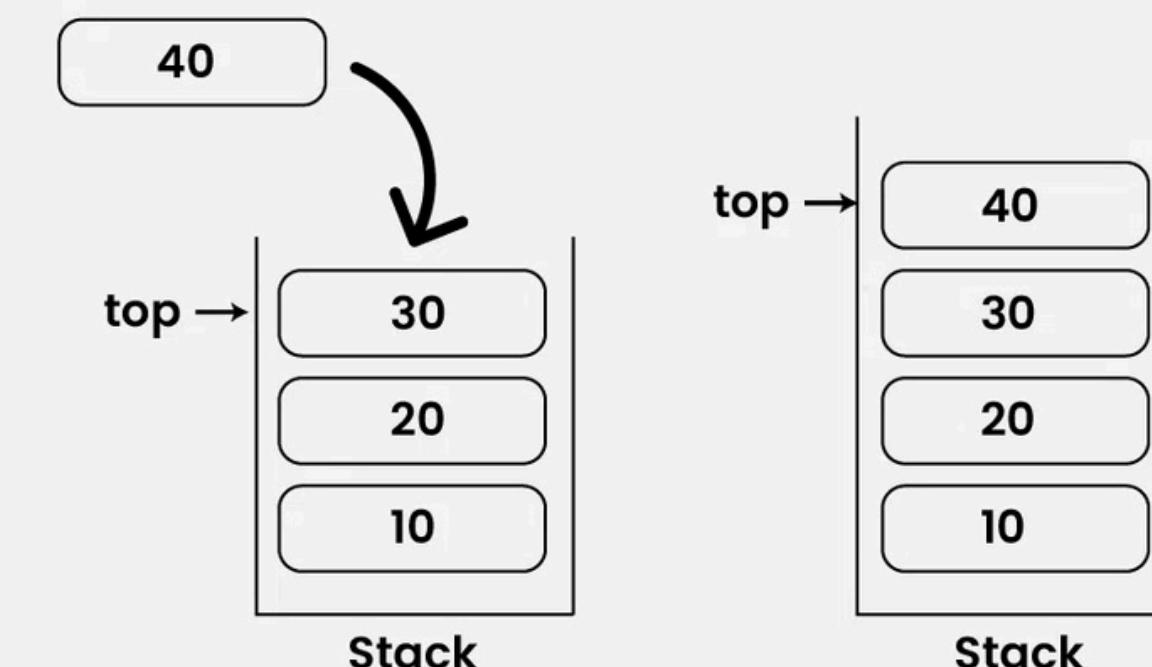
    // Accessing top element
    cout << "Top element: " << st.top() << endl;

    // Popping an element
    st.pop();
    cout << "Top element after pop: " << st.top() << endl;
    return 0;
}
```

Ejemplo de creación de una pila en C++.

Crea una pila de enteros

Push Operation in Stack



Pila



#include <stack>

Aplicaciones de las pilas

- Evaluación de expresiones (postfija, prefija)
- Conversión de expresiones (infija → postfija/prefija)
- Undo / Redo en editores y aplicaciones
- Recorridos en profundidad (DFS iterativo)
- Gestión de llamadas recursivas (stack de activación)
- Comprobación de paréntesis equilibrados
- Navegación web (Back / Forward)
- Algoritmos de backtracking (laberintos, puzzles)
- Control de estados en juegos (deshacer jugadas)



Cola



#include <queue>

Una cola es un contenedor secuencial que permite el acceso a los elementos que almacena por los dos extremos:

Por el frente (front), para recuperar elementos.

Por la cola (back), para insertarlos.

Tiene una interfaz muy parecida a la del tipo pila.

Es una secuencia del tipo FIFO.

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

int main()
{
    queue<int> q;
    q.push(10);
    q.push(5);

    // Accessing the front and back elements
    cout << "Front element: " << q.front() << endl;
    cout << "Back element: " << q.back() << endl;

    // Removing an element from the front
    q.pop();

    cout << "Front element after pop: " << q.front() << endl;
    return 0;
}
```

Crea una cola de enteros



Cola

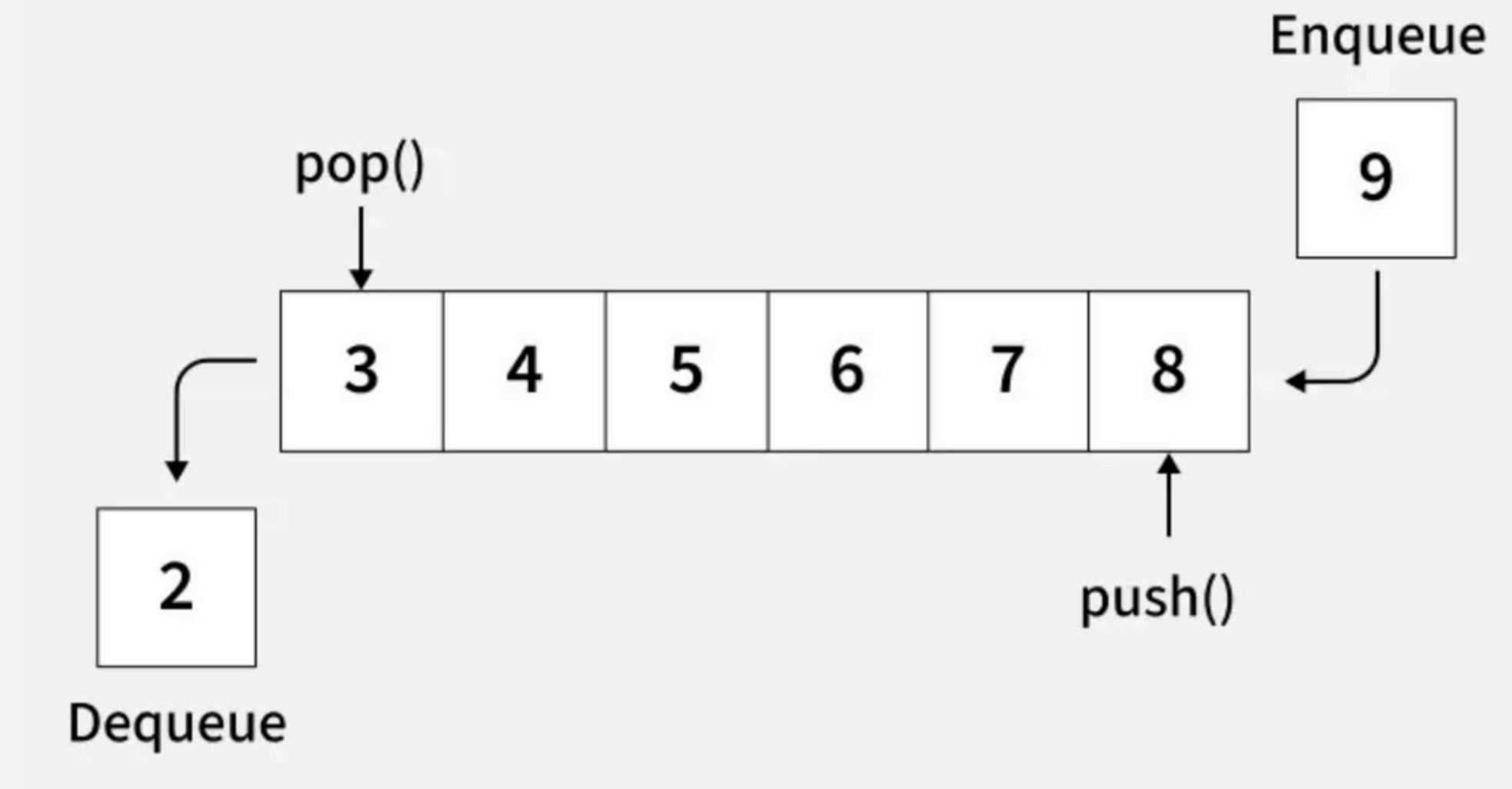


#include <queue>

Primero en entrar → primero en salir.

Ideal para:

- Simulación de colas reales (supermercado, impresora).
- Recorridos en anchura (BFS).
- Procesamiento por orden de llegada.



Cola



#include <queue>

Función	Descripción breve
front()	Devuelve el elemento del frente (el primero en entrar).
back()	Devuelve el último elemento (el más reciente).
empty()	Indica si la cola está vacía.
size()	Devuelve el número de elementos almacenados.
push()	Inserta un elemento al final de la cola.
push_range()	Inserta varios elementos al final (C++23).
emplace()	Construye un nuevo elemento directamente al final (C++11+).
pop()	Elimina el elemento del frente.
swap()	Intercambia el contenido de dos colas.



Cola con prioridad



#include <queue>

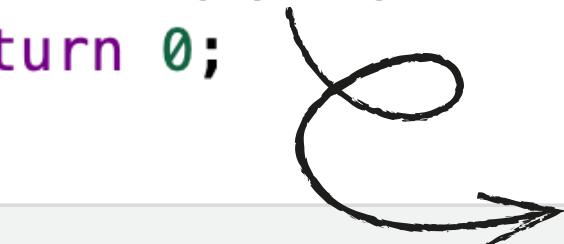
Una cola con prioridad es un contenedor adaptador donde siempre se extrae primero el elemento más prioritario, no el más antiguo como en una cola normal.

La clase con que se instancia la cola con prioridad debe de tener definido el operador <.

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

int main()
{
    priority_queue<int> pq;
    pq.push(9);
    pq.push(8);
    pq.push(6);

    // Accessing top element
    cout << pq.top();
    return 0;
}
```



Output: 9

Cola con prioridad



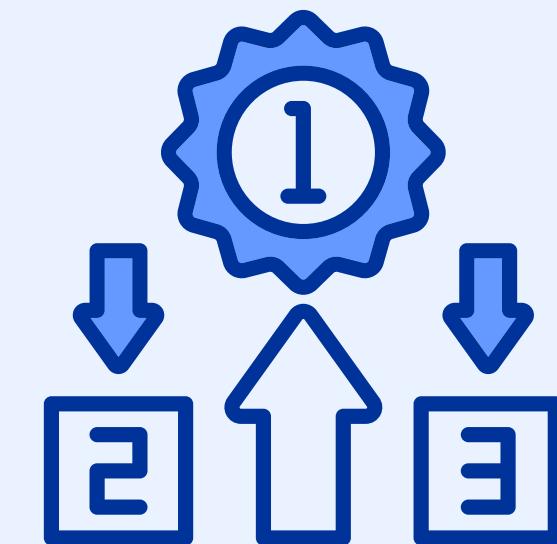
#include <queue>

Podemos modificar el comportamiento de la cola con prioridad para permitir que los elementos se ordenen con otro criterio

Para ello, es necesario indicarle:

- Contenedor sobre los que se almacenarán los elementos(vector o deque)
- Criterio de comparación

```
priority_queue<int, vector<int>, greater<int>> pq;
```



Cola con prioridad



#include <queue>

```
class MiTDA {  
public:  
    bool operator<(const MiTDA & a) const {  
        return x < a.x;  
    }  
    bool operator>(const MiTDA & a) const {  
        return x > a.x;  
    }  
private:  
    int x;  
};
```

Ls STL no sabe cómo comparar tus clases.
Tú decides qué significa “ser mayor” o
“ser menor”.



Cola con prioridad



#include <queue>

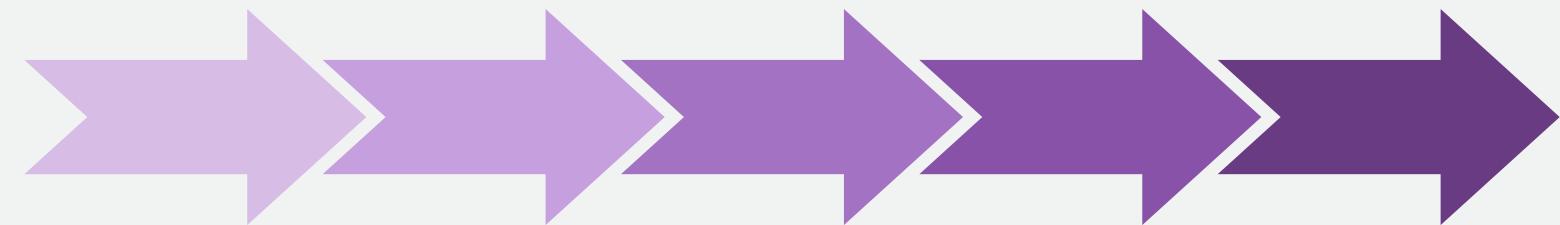
Cola con prioridad normal (el más mayor primero)

- **priority_queue<MiTDA, vector<MiTDA>, less<MiTDA>> cp1;**

Cola con prioridad inversa (el menor primero)

- **priority_queue<MiTDA, vector<MiTDA>, greater<MiTDA>> cp2;**

Contenedores secuenciales



Vector



#include <vector>

- El contenedor de secuencia más simple que existe.
- Puede concebirse como un vector de tamaño variable.
- Generalmente, almacena datos en bloques de tamaño fijo y cuando se sobrepasa ese tamaño, se reserva otro bloque.

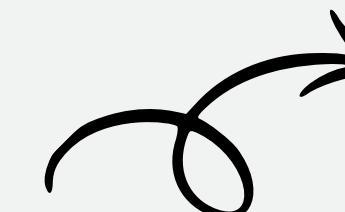


Vector



#include <vector>

En este tipo de contenedor se puede sumar un valor a un iterador porque los vectores se almacenan en memoria contigua.

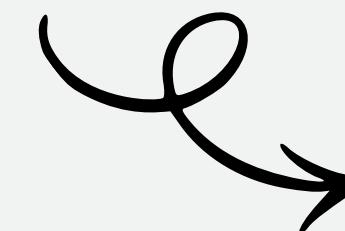


v.begin() apunta al primer elemento. Si sumamos +200, el iterador pasa a apuntar al elemento número 200 (posición 200).

Se puede hacer v.begin() + 200

v.erase(v.begin()+200); Elimina el elemento que hay en la posición 200 del vector.

v.insert(v.end()-100,12); // Debe existir esa posición!!



v.end() - 100 → posición: “último elemento menos 100”.
El insert coloca el valor 12 justo antes de esa posición.

Vector (matrices)



```
#include <vector>
```

Una matriz se puede ver como un vector de vectores

```
vector<vector<int>> v(3)
```

Para inicializarlo a un valor para cada fila:

```
vector<vector<int>> v(3, vector<int>(8)) →
```

crea un vector con 3 filas y 8 columnas
poniendo un cero en cada posición

Para acceder se usa []

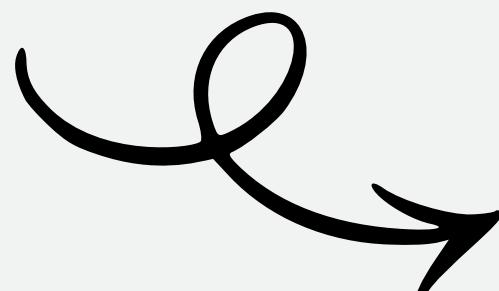
```
cout << v[2][1];
```

Listas



#include <list>

```
list<int> listaEnteros;  
list<string> listaCadenas;  
list<MiTipo> listaMiTipo;  
list<list<double> > listaDelistas;
```



Como todos los contenedores, se puede crear una lista del tipo que quieras!

Todos los contenedores son clases parametrizadas.

Listas



#include <list>

```
int main() {  
    list<int> list1, list2;  
  
    // Rellenamos list1 con push_back (añadir al final) y list2 con push_front (añadir al principio)  
    for (int i = 0; i < 6; ++i) {  
        list1.push_back(i); // [0, 1, 2, 3, 4, 5]  
        list2.push_front(i); // [i ... 0] → queda al revés  
    }  
  
    // Imprimimos el contenido inicial  
    cout << "list1 (push_back):";  
    for (int x : list1)  
        cout << x << " ";  
    cout << endl;  
  
    cout << "list2 (push_front):";  
    for (int x : list2)  
        cout << x << " ";  
    cout << endl;  
}
```

OUT:

list1 (push_back): 0 1 2 3 4 5

En list, insertar al principio o al final es siempre O(1) → muy eficiente.

En una list, los iteradores solo pueden avanzar o retroceder de uno en uno (++it, --it).

No se puede hacer it + 3 como en vector, porque no es acceso aleatorio.

Listas



#include <list>

```
cout << "list2 (push_front):";
for (int x : list2) cout << x << "";
cout << endl;
```

OUT:

list2 (push_front): 5 4 3 2 1 0

```
// Añadimos al final de list2 todos los elementos de list1
for (list<int>::const_iterator it = list1.begin(); it != list1.end(); ++it)
    list2.push_back(*it);
```

// Imprimimos el resultado final

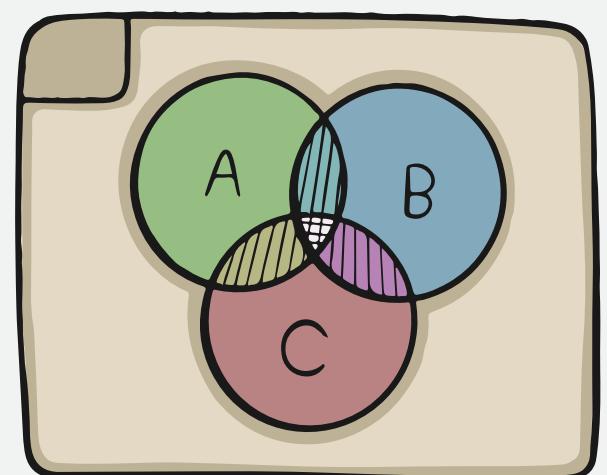
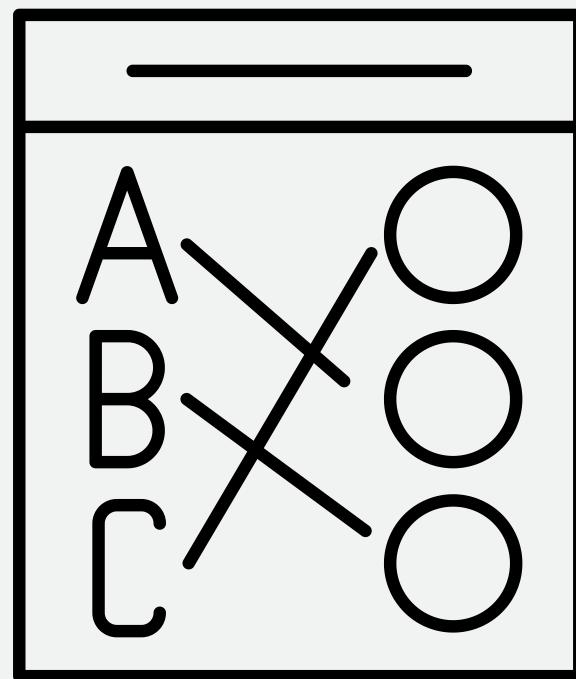
```
cout << "list2 después de añadir list1 al final: ";
for (int x : list2)
    cout << x << "";
cout << endl;
```

OUT:

list2 después de añadir list1 al final: 5 4 3 2 1 0 0 1 2 3 4 5

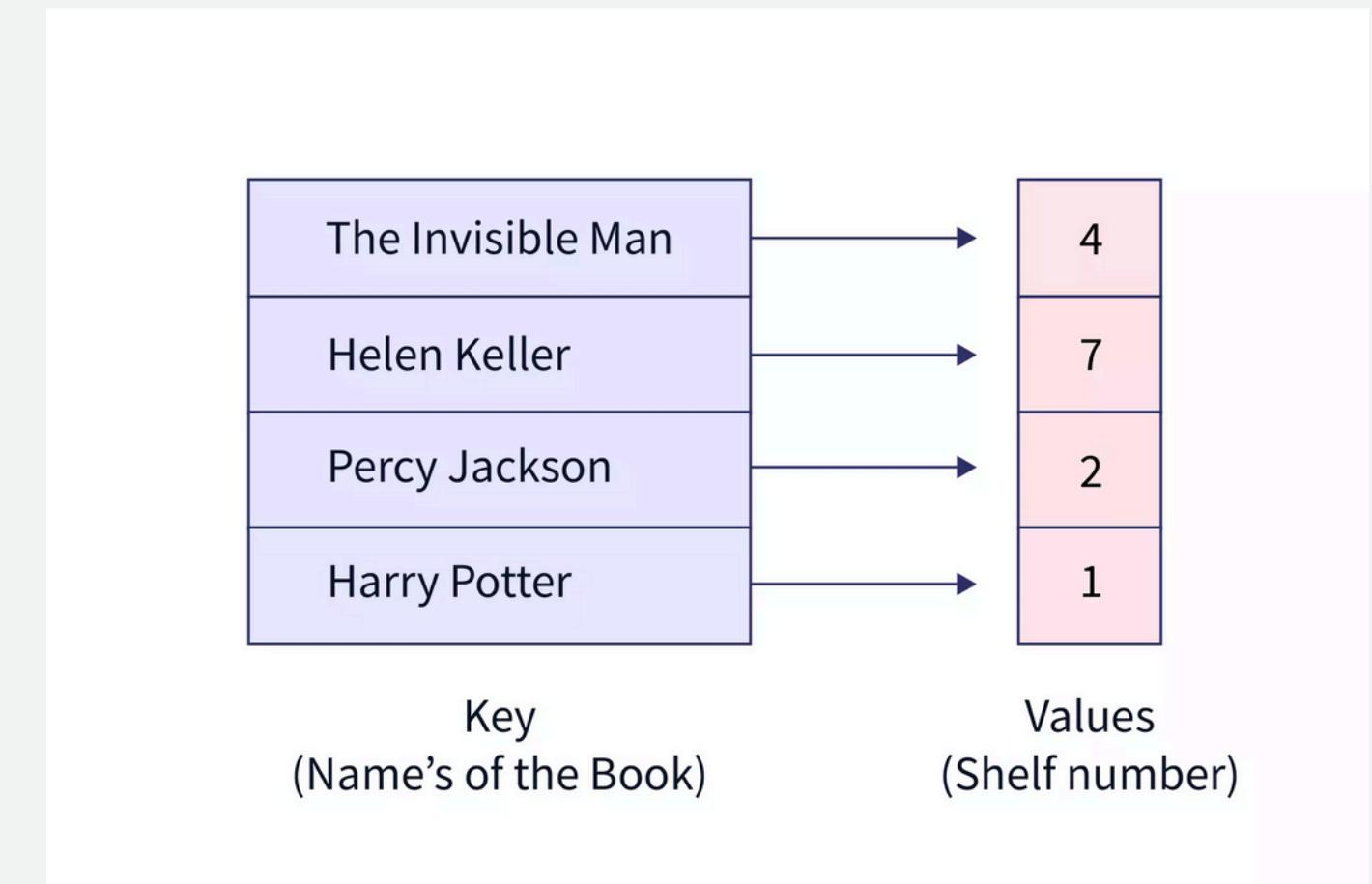
...

Contenedores asociativos



Contenedores asociativos

- Los elementos están siempre ordenados por la clave.
- Las búsquedas, inserciones y borrados tienen coste $O(\log n)$.
- No permiten acceso por índice ($a[3]$ no existe).
- Se recorren mediante iteradores bidireccionales.



Contenedores asociativos

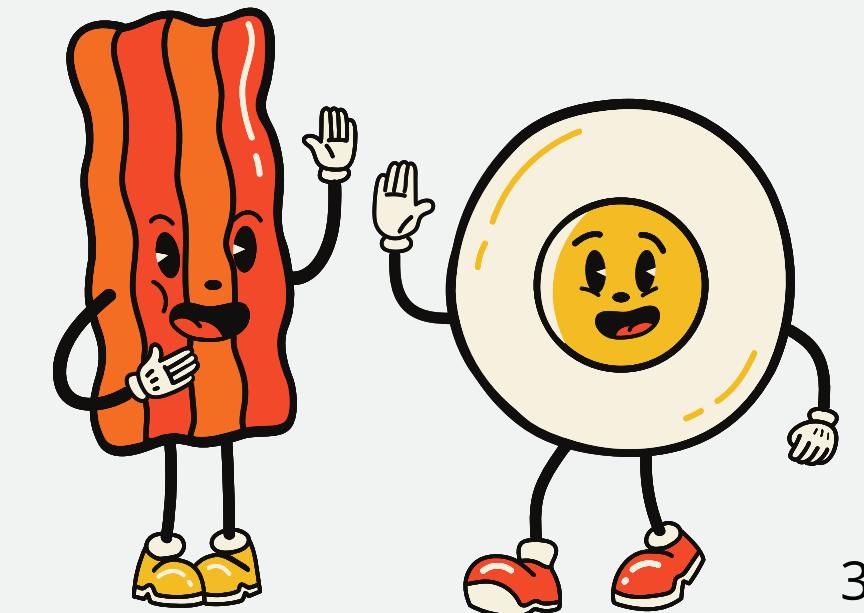
- **set<T> (conjunto)** sólo almacena valores de clave y no permite que haya valores repetidos.
- **multiset<T> (bolsa)** sólo almacena valores de clave y permite la existencia de valores repetidos.
- **map<key, T>** (diccionario) almacena pares (clave, dato) pero no permite valores de clave repetidos.
- **multimap<key, T>** almacena pares (clave, dato) y permite valores repetidos en la clave.

Pair

pair<const Key, T>

Este TDA es una utilidad que permite tratar un par de valores como una unidad.

Miembro	Descripción
<code>pair<T1, T2> pair()</code>	Crea un par vacío
<code>pair(const pair<T1, T2> & p)</code>	Crea un par que es copia de <code>p</code>
<code>pair(const T1 & v1, const T2 & v2)</code>	Constructor primitivo
<code>first</code>	Dato de tipo <code>T1</code>
<code>second</code>	Dato de tipo <code>T2</code>
<code>x = y</code>	Asigna al par <code>x</code> una copia del contenido de <code>y</code>
<code>x == y</code>	Devuelve <code>true</code> si ambos pares son iguales en ambas coordenadas; <code>false</code> , en otro caso
<code>x < y</code>	Devuelve <code>true</code> si ambas coordenadas de <code>x</code> son menores que las de <code>y</code> ; <code>false</code> , en otro caso



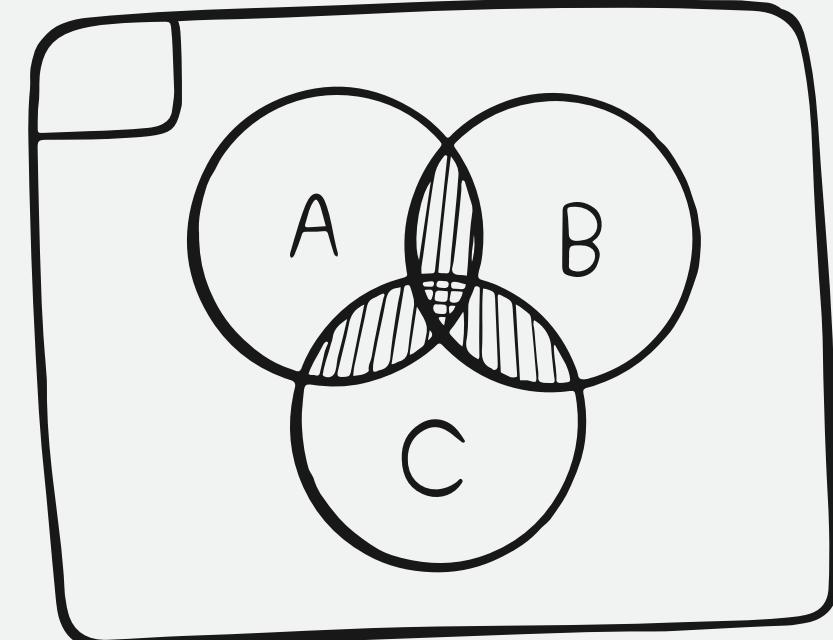
Set



#include <set>

Un conjunto es un contenedor asociativo de **valores únicos**, llamados claves o miembros del conjunto, que se almacenan de manera ordenada, no descendente.

El tipo Conjunto está basado en el concepto matemático de conjunto. **No admite repetidos!**



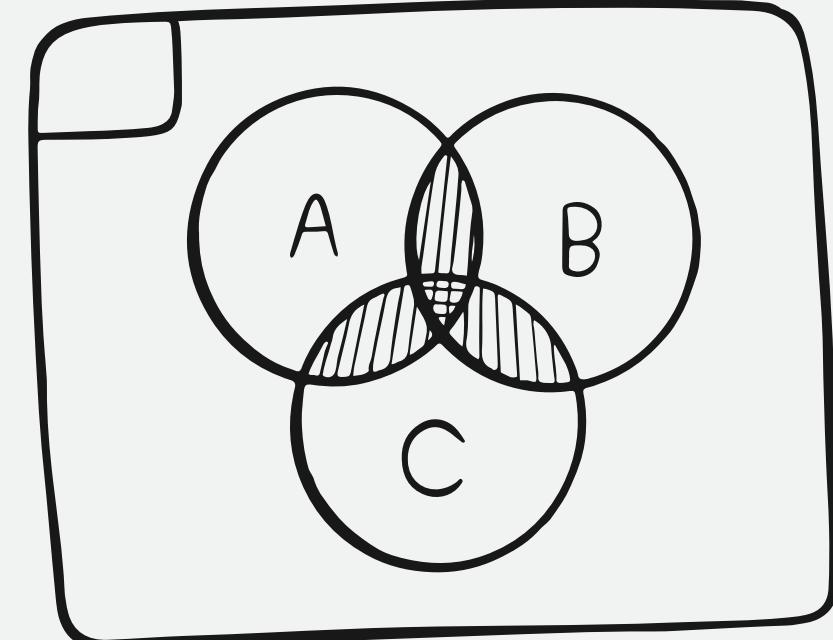
Set



#include <set>

Un conjunto es un contenedor asociativo de **valores únicos**, llamados claves o miembros del conjunto, que se almacenan de manera ordenada, no descendente.

El tipo Conjunto está basado en el concepto matemático de conjunto. **No admite repetidos!**



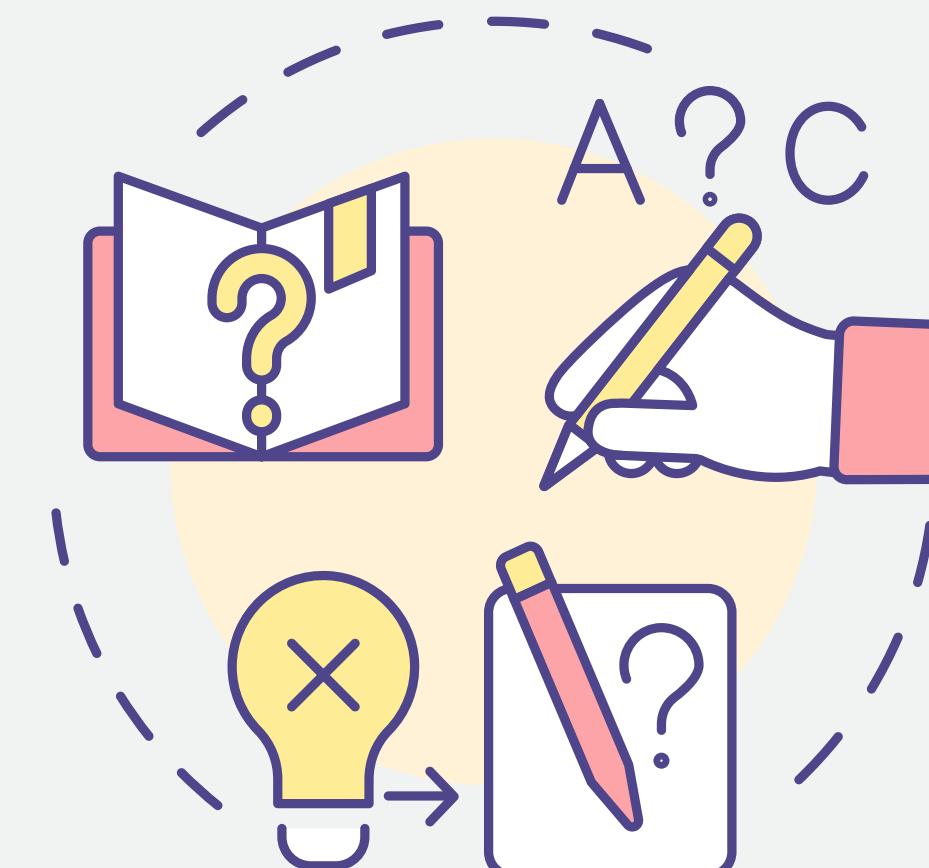
Set



#include <set>

Ejemplo práctico:

Un corrector ortográfico de un procesador de textos puede utilizar un conjunto para almacenar todas las palabras que considera correctas ortográficamente hablando.



Set

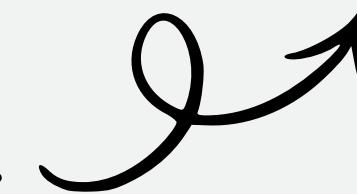


#include <set>

```
class Person {  
    private:  
        string fn; // first name  
        string ln; // last name  
    public:  
        Person() {};  
        Person(const string& f, const string& n): fn(f), ln(n) {};  
        string firstname() const;  
        string lastname() const;  
    };  
    inline string Person::firstname() const { return fn; };  
    inline string Person::lastname() const { return ln; };
```

declaro una clase Person que contiene dos String

Recuerda!: lista de inicialización



Recuerda!: Se implementan fuera de la clase, usando el operador de resolución de ámbito **Person::**



Set



```
#include <set>
```

```
ostream& operator<< (ostream& s, const Person& p){  
    s << "[" << p.firstname() << " " << p.lastname() << "]";  
    return s;    devuelve: [nombre apellido]  
}
```

/* class for function predicate operator () devuelve si una persona está antes que otra */

```
class PersonSortCriterion {  
public:  
    bool operator() (const Person& p1, const Person& p2) const {  
        /* una persona antes que otra si los apellidos están antes.  
        Si los apellidos son iguales pero el nombre está antes */  
        return p1.lastname()<p2.lastname()  
            || (p1.lastname()==p2.lastname())  
            && p1.firstname()<p2.firstname());  
    }  
}
```

Sobrecarga del operador << para imprimir Person. Aquí se define cómo imprimir un objeto Person con cout u otro ostream.



Esta clase es un “functor”: un objeto que se puede llamar como si fuera una función. Se usa como criterio de comparación para contenedores.



Ejemplo de uso:

```
PersonSortCriterion cmp;  
bool r = cmp(p1, p2);
```

Set



#include <set>

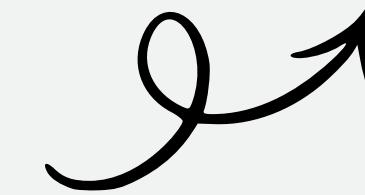
```
int main()
{
    Person p1("nicolai","josuttis");
    Person p2("ulli","josuttis");
    Person p3("anica","josuttis");
    Person p4("lucas","josuttis");
    Person p5("lucas","otto");
    Person p6("lucas","arm");
    Person p7("anica","holle");
    // declara el set con un criterio de ordenación especial
    typedef set<Person,PersonSortCriterion> PersonSet;
```

se crean 7 objetos persona con nombre y apellido



El contenedor es un set de Person.

Para ordenar los elementos, usa la función/objeto PersonSortCriterion que definimos antes.



Set



#include <set>

```
// crear la colección  
PersonSet coll;  
coll.insert(p1);  
coll.insert(p2);  
coll.insert(p3);  
coll.insert(p4);  
coll.insert(p5);  
coll.insert(p6);  
coll.insert(p7);  
  
// sacar por pantalla los elementos  
cout << "set:" << endl;  
PersonSet::iterator pos;  
for (pos = coll.begin(); pos != coll.end(); ++pos) {  
    cout << *pos << endl;  
}  
}
```

se crea un set vacío de Person con el criterio PersonSortCriterion.

set no permite duplicados. Si dos personas se consideraran “iguales” según el criterio de orden (PersonSortCriterion), uno de ellos no se insertaría.

En este caso, como los nombres y apellidos no se repiten con la misma combinación exacta, se aceptan todos.

Declara un iterador pos para recorrer el set.

Multiset



#include <multiset>

- Una bolsa almacena valores de clave que pueden estar repetidos (múltiples ocurrencias). También los guarda de manera ordenada, no descendente.
- En la STL, recibe el nombre de multiset.
- Está especialmente pensado para realizar operaciones de inserción, borrado y pertenencia de un elemento a una bolsa.



Map



#include <map>

Un diccionario es un contenedor que almacena elementos (valores) identificados mediante una clave, es decir, pares (clave, valor), es decir, (key_type, data_type).

No existen pares (clave,valor) repetidos.



Map



#include <map>

Ejemplos y aplicaciones:

- El diccionario de la RAE contiene parejas (palabra, definición).
- El stock de un almacén: artículo, número de unidades que existe.
- Un guía telefónica: abonado, número de teléfono.



Map

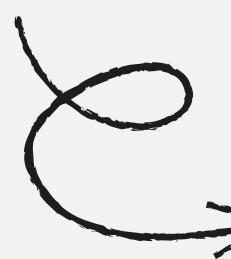


```
#include <map>
```

Acceso a los elementos:

```
map<string, int> edades;  
edades["Ana"] = 21;
```

```
cout << edades.first << ", " << edades.second << " años";
```



Output:Ana, 21 años

Map



```
#include <map>
```

```
typedef map<string, float> StringFloatMap;
```

```
StringFloatMap acciones;
```

```
// Inserción de elementos.
```

```
acciones["BASF"] = 369.50; acciones["W"] = 413.50;
```

```
acciones["Daimler"] = 819.00; acciones["BMW"] = 834.00;
```

```
acciones["Siemens"] = 842.20;
```

```
// Impresión
```

```
StringFloatMap::iterator pos;
```

```
for (pos = acciones.begin(); pos != acciones.end(); ++pos)
```

```
    cout << "acción: " << (*pos).first << "\t" << "precio: " << pos->second << endl;
```

```
// Se doblan los precios de la acción
```

```
for (pos = acciones.begin(); pos != acciones.end(); ++pos)
```

```
    pos->second *= 2;
```

```
}
```

escribir StringFloatMap es lo mismo que escribir map<string, float>

operator[] en map inserta la clave si no existe.
También sirve para modificar una ya existente.

pos será una especie de “puntero” que recorre los elementos del map.

(*pos).first → la clave (el nombre de la empresa, string).

(*pos).second → el valor (el precio float).

Multimap



#include <map>

Un contenedor asociativo ordenado que almacena pares (clave → valor) igual que map, pero permitiendo varias entradas con la misma clave.

- Una misma clave puede estar asociada a varios valores.
- Los elementos están ordenados por clave de manera automática.
- Ideal para representar relaciones 1 → muchos.
 - Agrupar objetos bajo la misma categoría
 - Índice de un libro o documento
 - Alumnos agrupados por curso