Contenedores

1.1 Contenedores Secuenciales
Los contenedores secuenciales mantienen los elementos en un orden lineal. Los principales tipos son:

std::vector

Descripción: Un contenedor de tamaño dinámico que almacena elementos en un array continuo. Proporciona acceso rápido a los elementos mediante índices.

Operaciones principales:
push_back(): Añade un elemento al final.
pop_back(): Elimina el último elemento.
at(), operator[]: Acceso a elementos.
size(), capacity(), resize(): Gestión del tamaño y capacidad.

Ejemplo:

```
#include <vector>
#include <iostream>

int main() {
        std::vector<int> vec = {1, 2, 3};

        vec.push_back(4);
        std::cout << "Elemento en el índice 2: " << vec[2] << std::endl;
        std::cout << "Tamaño: " << vec.size() << std::endl;

        return 0;
}
```

std::deque

Descripción: Un contenedor de doble cola que permite inserciones y eliminaciones rápidas tanto al principio como al final.
Operaciones principales:
push_front(), pop_front(): Añadir o eliminar elementos al principio.
push_back(), pop_back(): Añadir o eliminar elementos al final.

Ejemplo:

```
#include <deque>
#include <iostream>

int main() {
        std::deque<int> deq = {1, 2, 3};

        deq.push_front(0);
        deq.push_back(4);
        std::cout << "Elemento en el frente: " << deq.front() << std::endl;
        std::cout << "Elemento en el final: " << deq.back() << std::endl;

        return 0;
}
```

std::list

Descripción: Una lista doblemente enlazada que permite inserciones y eliminaciones rápidas en cualquier posición.
Operaciones principales:
push_front(), pop_front(): Inserción y eliminación al principio.
push_back(), pop_back(): Inserción y eliminación al final.
insert(), erase(): Inserción y eliminación en posiciones específicas.

Ejemplo:

```
#include <list>
#include <iostream>

int main() {
        std::list<int> lst = {1, 2, 3};

        lst.push_back(4);
        lst.push_front(0);
        std::cout << "Primer elemento: " << lst.front() << std::endl;

        return 0;
}
```

std::array

Descripción: Un array de tamaño fijo que proporciona una interfaz similar a los arrays tradicionales pero con características adicionales.
Operaciones principales:
at(), operator[]: Acceso a elementos.
size(): Obtención del tamaño.

Ejemplo:

```
#include <array>
#include <iostream>

int main() {
        std::array<int, 3> arr = {1, 2, 3};

        std::cout << "Elemento en el índice 1: " << arr[1] << std::endl;
        std::cout << "Tamaño: " << arr.size() << std::endl;

        return 0;
}
```

1.2 Contenedores Asociativos
Los contenedores asociativos permiten el almacenamiento de elementos en una estructura ordenada o no ordenada y proporcionan acceso rápido basado en claves.

std::set

Descripción: Un contenedor que almacena elementos únicos en un orden específico. Los elementos están ordenados de forma predeterminada.
Operaciones principales:
insert(): Añade un elemento.
find(): Busca un elemento.
erase(): Elimina un elemento.

Ejemplo:

```
#include <set>
#include <iostream>

int main() {
        std::set<int> s = {1, 2, 3};

        s.insert(4);
        if (s.find(2) != s.end()) {
                std::cout << "Elemento 2 encontrado." << std::endl;
        }

        return 0;
}
```

std::map

Descripción: Un contenedor que almacena pares ***clave-valor***, donde las claves son únicas y están ordenadas.
Operaciones principales:
insert(): Añade un par clave-valor.
find(): Busca un valor por clave.
erase(): Elimina un par clave-valor.

Ejemplo:

```
#include <map>
#include <iostream>

int main() {
        std::map<int, std::string> m = {{1, "one"}, {2, "two"}};

        m[3] = "three";
        std::cout << "Valor asociado con la clave 2: " << m[2] << std::endl;

        return 0;
}
```

std::unordered_set

Descripción: Un contenedor que almacena elementos únicos en una tabla hash, sin un orden específico.
Operaciones principales:
insert(): Añade un elemento.
find(): Busca un elemento.
erase(): Elimina un elemento.

Ejemplo:

```
#include <unordered_set>
#include <iostream>

int main() {
        std::unordered_set<int> us = {1, 2, 3};

        us.insert(4);
        if (us.find(3) != us.end()) {
                std::cout << "Elemento 3 encontrado." << std::endl;
        }
```

```
        return 0;
}
```

std::unordered_map

Descripción: Un contenedor que almacena pares clave-valor en una tabla hash, sin un orden específico.
Operaciones principales:
insert(): Añade un par clave-valor.
find(): Busca un valor por clave.
erase(): Elimina un par clave-valor.

Ejemplo:

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

int main() {
        std::unordered_map<int, std::string> um = {{1, "one"}, {2, "two"}};

        um[3] = "three";
        std::cout << "Valor asociado con la clave 1: " << um[1] << std::endl;

        return 0;
}
```

2. Elementos Clave Asociados con Contenedores
2.1 Iteradores
Los iteradores proporcionan una forma de acceder a los elementos de un contenedor de manera secuencial.

Cada tipo de contenedor tiene su propia implementación de iteradores.

begin(): Devuelve un iterador al primer elemento.
end(): Devuelve un iterador al final (posición después del último elemento).

2.2 Algoritmos STL
STL proporciona una serie de algoritmos que funcionan con contenedores. Algunos ejemplos incluyen:

std::sort(): Ordena los elementos.
std::find(): Busca un elemento.
std::copy(): Copia elementos de un contenedor a otro.

2.3 Capacidad y Tamaño
size(): Devuelve el número de elementos.
empty(): Devuelve true si el contenedor está vacío.
resize(): Cambia el tamaño del contenedor.

Resumen
Contenedores Secuenciales: std::vector, std::deque, std::list, std::array. Almacenan elementos en un orden lineal.
Contenedores Asociativos: std::set, std::map, std::unordered_set, std::unordered_map. Almacenan elementos en estructuras ordenadas o no ordenadas.
Iteradores y Algoritmos: Proporcionan formas eficientes de recorrer y manipular los elementos de los contenedores.
Capacidad y Tamaño: Métodos para obtener y gestionar el tamaño de los contenedores.
Estos contenedores y características permiten a los programadores manejar colecciones de datos de manera eficiente y flexible. Si necesitas más información o ejemplos específicos, no dudes en preguntar.

información y ejemplos prácticos.

1. Contenedores Secuenciales
1.1 std::vector
Descripción: Un contenedor de tamaño dinámico que almacena elementos en un array continuo. Ofrece acceso aleatorio rápido y es eficiente para la inserción y eliminación de elementos al final.
Características:

Tamaño dinámico.
Acceso aleatorio eficiente.
Añade o elimina elementos al final.

```
#include <vector>
#include <iostream>

int main() {
        // Crear un vector e inicializarlo
        std::vector<int> vec = {1, 2, 3};

        // Añadir elementos
        vec.push_back(4);
        vec.push_back(5);

        // Acceder a elementos
       std::cout << "Elemento en el índice 2: " << vec[2] << std::endl;

        // Imprimir todos los elementos
       std::cout << "Todos los elementos: ";
```

```
        for (int num : vec) {
                std::cout << num << " ";
        }
        std::cout << std::endl;

        return 0;
}
```

1.2 std::deque

Descripción: Un contenedor de doble cola que permite inserciones y eliminaciones rápidas tanto al principio como al final.
Características:

Tamaño dinámico.
Inserciones y eliminaciones rápidas en ambos extremos.
Ejemplo:

```
#include <deque>
#include <iostream>

int main() {
        // Crear un deque e inicializarlo
        std::deque<int> deq = {1, 2, 3};

        // Añadir elementos
        deq.push_front(0);
        deq.push_back(4);

        // Acceder a elementos
        std::cout << "Primer elemento: " << deq.front() << std::endl;
        std::cout << "Último elemento: " << deq.back() << std::endl;

        // Imprimir todos los elementos
        std::cout << "Todos los elementos: ";
        for (int num : deq) {
                std::cout << num << " ";
        }
        std::cout << std::endl;

        return 0;
}
```

1.3 std::list

Descripción: Una lista doblemente enlazada que permite inserciones y eliminaciones rápidas en cualquier posición.
Características:

Tamaño dinámico.
Inserciones y eliminaciones rápidas en cualquier posición.
Acceso secuencial.

Ejemplo:

```
#include <list>
#include <iostream>

int main() {
        // Crear una lista e inicializarla
        std::list<int> lst = {1, 2, 3};

        // Añadir elementos
       lst.push_front(0);
        lst.push_back(4);

        // Acceder a elementos
        std::cout << "Primer elemento: " << lst.front() << std::endl;

        // Imprimir todos los elementos
        std::cout << "Todos los elementos: ";
        for (int num : lst) {
                std::cout << num << " ";
        }
        std::cout << std::endl;

        return 0;
}
```

1.4 std::array

Descripción: Un array de tamaño fijo que proporciona una interfaz segura con características adicionales.
Características:

Tamaño fijo.
Seguridad en el tamaño.
Métodos adicionales como `.size()`.

Ejemplo:

```
#include <array>
#include <iostream>

int main() {
        // Crear un array e inicializarlo
        std::array<int, 3> arr = {1, 2, 3};

        // Acceder a elementos
        std::cout << "Elemento en el índice 1: " << arr[1] << std::endl;

        // Imprimir todos los elementos
        std::cout << "Todos los elementos: ";
        for (int num : arr) {
                std::cout << num << " ";
        }
        std::cout << std::endl;

        return 0;
}
```

2. Contenedores Asociativos

2.1 std::set

Descripción: Un contenedor que almacena elementos únicos en un orden específico. Utiliza un árbol binario auto-balanceado para mantener el orden.
Características:

Elementos únicos.
Ordenado automáticamente.
Búsqueda rápida.

Ejemplo:

```
#include <set>
#include <iostream>

int main() {
        // Crear un set e inicializarlo
        std::set<int> s = {1, 2, 3};

        // Añadir elementos
        s.insert(4);

        // Buscar elementos
       if (s.find(2) != s.end()) {
                std::cout << "Elemento 2 encontrado." << std::endl;
        }

        // Imprimir todos los elementos
        std::cout << "Todos los elementos: ";
        for (int num : s) {
                std::cout << num << " ";
        }
        std::cout << std::endl;

        return 0;
}
```

2.2 std::map

Descripción: Un contenedor que almacena pares clave-valor únicos, ordenados por clave. Utiliza un árbol binario auto-balanceado.
Características:

Pares **clave-valor** únicos.
Claves ordenadas automáticamente.
Ejemplo:

```
#include <map>
#include <iostream>

int main() {
        // Crear un map e inicializarlo
        std::map<int, std::string> m = {{1, "one"}, {2, "two"}};

        // Añadir pares clave-valor
        m[3] = "three";

        // Acceder a elementos
        std::cout << "Valor asociado con la clave 2: " << m[2] << std::endl;

        // Imprimir todos los pares clave-valor
        std::cout << "Todos los pares clave-valor: ";
        for (const auto& pair : m) {
                std::cout << pair.first << "->" << pair.second << " ";
        }
        std::cout << std::endl;
```

```
        return 0;
}
```

2.3 std::unordered_set

Descripción: Un contenedor que almacena elementos únicos en una tabla hash. No mantiene un orden específico.
Características:

Elementos únicos.
Sin orden específico.
Operaciones en promedio O(1).
Ejemplo:

```
#include <unordered_set>
#include <iostream>

int main() {
        // Crear un unordered_set e inicializarlo
        std::unordered_set<int> us = {1, 2, 3};

        // Añadir elementos
        us.insert(4);

        // Buscar elementos
        if (us.find(3) != us.end()) {
                std::cout << "Elemento 3 encontrado." << std::endl;
        }

        // Imprimir todos los elementos
        std::cout << "Todos los elementos: ";
        for (int num : us) {
                std::cout << num << " ";
        }
        std::cout << std::endl;

        return 0;
}
```

2.4 std::unordered_map

Descripción: Un contenedor que almacena pares clave-valor en una tabla hash. No mantiene un orden específico de las claves.
Características:

Pares clave-valor únicos.
Sin orden específico de claves.
Operaciones en promedio O(1).
Ejemplo:

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

int main() {
        // Crear un unordered_map e inicializarlo
        std::unordered_map<int, std::string> um = {{1, "one"}, {2, "two"}};

        // Añadir pares clave-valor
        um[3] = "three";

        // Acceder a elementos
        std::cout << "Valor asociado con la clave 1: " << um[1] << std::endl;

        // Imprimir todos los pares clave-valor
        std::cout << "Todos los pares clave-valor: ";
        for (const auto& pair : um) {
                std::cout << pair.first << "->" << pair.second << " ";
        }
        std::cout << std::endl;

        return 0;
}
```

3. Elementos Clave Asociados con Contenedores

3.1 Iteradores

Los iteradores permiten recorrer los elementos de un contenedor. Los principales tipos de iteradores son:

Iteradores de entrada: Solo permiten la lectura de los datos.
Iteradores de salida: Solo permiten la escritura de los datos.
Iteradores bidireccionales: Permiten avanzar y retroceder.
Iteradores aleatorios: Permiten el acceso aleatorio a los elementos (como en std::vector).
Ejemplo de uso de iteradores:

```
#include <vector>
#include <iostream>

int main() {
        std::vector<int> vec = {1, 2, 3, 4, 5};

        // Usar iteradores para imprimir los elementos
        std::cout << "Elementos usando iteradores: ";
```

```
        for (auto it = vec.begin(); it != vec.end(); ++it) {
                std::cout << *it << " ";
        }
        std::cout << std::endl;

        return 0;
}
```

3.2 Algoritmos STL

STL proporciona una variedad de algoritmos que pueden trabajar con contenedores, como:

std::sort(): Ordena elementos.
std::find(): Busca un elemento.
std::copy(): Copia elementos de un contenedor a otro.
Ejemplo usando std::sort():

```
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
        std::vector<int> vec = {5, 3, 4, 1, 2};

        // Ordenar los elementos
        std::sort(vec.begin(), vec.end());

        // Imprimir los elementos ordenados
        std::cout << "Elementos ordenados: ";
        for (int num : vec) {
                std::cout << num << " ";
        }
        std::cout << std::endl;

        return 0;
}
```

4. Capacidad y Tamaño

size(): Obtiene el número de elementos.
empty(): Verifica si el contenedor está vacío.
resize(): Cambia el tamaño del contenedor.
Ejemplo de uso de size() y empty():

```
#include <vector>
#include <iostream>

int main() {
        std::vector<int> vec = {1, 2, 3};

        // Obtener el tamaño
        std::cout << "Tamaño del vector: " << vec.size() << std::endl;

        // Verificar si está vacío
        if (vec.empty()) {
                std::cout << "El vector está vacío." << std::endl;
        } else {
                std::cout << "El vector no está vacío." << std::endl;
        }

        return 0;
}
```