Listas en c++:

Índice

Introducción a las listas en C++:

En C++, una lista es una estructura de datos que forma parte de la Biblioteca Estándar de Plantillas (STL, por sus siglas en inglés). La clase std::list implementa una lista doblemente enlazada, lo que permite una inserción y eliminación eficiente de elementos en cualquier posición de la lista.

Características principales de std::list:

- Doble enlace: Cada elemento tiene un puntero al anterior y al siguiente.
- Inserción y eliminación eficientes: Especialmente en el medio de la lista.
- No proporciona acceso aleatorio: A diferencia de **std::vector**, no puedes acceder a elementos por índice rápidamente.
- Uso de memoria: Consume más memoria que **std::vector** debido a los punteros adicionales.

Creación de una lista

Para utilizar std::list, primero debes incluir la cabecera correspondiente y especificar el tipo de datos que almacenará la lista.

```
#include <iostream>
#include <list> // Necesario para std::list

int main() {
        // Crear una lista de enteros vacía
        std::list<int> miLista;

        // Crear una lista con elementos iniciales
        std::list<int> otraLista = {1, 2, 3, 4, 5};

        return 0;
}
```

Declaración de una lista

```
std::list<Tipo> nombreLista;
```

1. Tipo: Tipo de datos que almacenará la lista (por ejemplo, int, std::string, objetos personalizados).
2. nombreLista: Nombre que le darás a la lista.

Insertar elementos en una lista

Existen varias formas de insertar elementos en una lista:

Al final de la lista **(push_back)**
Al inicio de la lista (push_front)
En una posición específica utilizando iteradores (insert)

1. Insertar al final de la lista

```
miLista.push_back(10); // Agrega 10 al final de la lista
```

2. Insertar al inicio de la lista

```
miLista.push_front(5); // Agrega 5 al inicio de la lista
```

3. Insertar en una posición específica

Para insertar en una posición específica, necesitas usar iteradores para indicar dónde insertar el nuevo elemento.

```
// Supongamos que queremos insertar 7 después del segundo elemento
std::list<int>::iterator it = miLista.begin();
```

```
std::advance(it, 2); // Mueve el iterador a la tercera posición
miLista.insert(it, 7); // Inserta 7 antes de la posición apuntada por it
```

Nota: std::advance mueve el iterador un número específico de posiciones. En listas, esto tiene una complejidad lineal.

Modificar elementos en una lista

Para modificar elementos en una lista, debes acceder a ellos a través de iteradores o referencias.

Usando iteradores

```
for (std::list<int>::iterator it = miLista.begin(); it != miLista.end(); ++it) {
        if (*it == 10) {
                *it = 20; // Cambia el valor de 10 a 20
        }
}
```

Usando referencias en un bucle for

```
for (int &elemento : miLista) {
        if (elemento == 10) {
                elemento = 20;
        }
}
```

Eliminar elementos de una lista

Existen varias formas de eliminar elementos:

Eliminar el primer elemento **(pop_front)**
Eliminar el último elemento **(pop_back)**
Eliminar un elemento específico utilizando iteradores **(erase)**
Eliminar todos los elementos que coincidan con un valor **(remove)**

1. Eliminar el primer elemento

```
miLista.pop_front(); // Elimina el primer elemento de la lista
```

2. Eliminar el último elemento

```
miLista.pop_back(); // Elimina el último elemento de la lista
```

3. Eliminar un elemento específico

Para eliminar un elemento en una posición específica, usa un iterador junto con erase.

```
std::list<int>::iterator it = miLista.begin();
std::advance(it, 2); // Mueve el iterador a la tercera posición
miLista.erase(it); // Elimina el elemento en la posición apuntada por it
```

4. Eliminar todos los elementos que coincidan con un valor

```
miLista.remove(10); // Elimina todos los elementos que sean iguales a 10
```

Recorrer una lista

Para procesar o mostrar los elementos de una lista, puedes utilizar diferentes métodos:

1. Usando iteradores

```
for (std::list<int>::iterator it = miLista.begin(); it != miLista.end(); ++it) {
        std::cout << *it << " ";
}
std::cout << std::endl;
```

2. Usando bucles for basados en rangos (C++11 en adelante)

```
for (const int &elemento : miLista) {
        std::cout << elemento << " ";
}
std::cout << std::endl;
```

Ejemplos completos

A continuación, se presenta un ejemplo completo que demuestra la creación, inserción, modificación, eliminación y recorrido de una lista en C++.

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

int main() {
        // 1. Creación de una lista de enteros

        std::list<int> miLista = {1, 2, 3, 4, 5};

        // 2. Insertar elementos

       miLista.push_back(6); // Lista: 1, 2, 3, 4, 5, 6
        miLista.push_front(0); // Lista: 0, 1, 2, 3, 4, 5, 6

        // Insertar 99 después del tercer elemento

        std::list<int>::iterator it = miLista.begin();
        std::advance(it, 3); // Posición 3 (cuarto elemento)
        miLista.insert(it, 99); // Lista: 0, 1, 2, 99, 3, 4, 5, 6

        // 3. Modificar elementos

      for (int &elemento : miLista) {
                if (elemento == 99) {
                        elemento = 100; // Reemplaza 99 por 100
                }
        }

        // 4. Eliminar elementos

        miLista.pop_front(); // Elimina 0
        miLista.pop_back(); // Elimina 6
        miLista.remove(2); // Elimina todas las ocurrencias de 2

        // 5. Recorrer y mostrar la lista

        std::cout << "Contenido de la lista: ";
        for (const int &elemento : miLista) {
                std::cout << elemento << " ";
        }
        std::cout << std::endl;

        return 0;
}
```

Salida esperada:

```
Contenido de la lista: 1 100 3 4 5
```

Explicación del ejemplo:

Creación: Se inicializa la lista con los elementos {1, 2, 3, 4, 5}.
Inserción:
push_back(6): Agrega 6 al final.
push_front(0): Agrega 0 al inicio.
insert(it, 99): Inserta 99 en la cuarta posición.
Modificación: Reemplaza 99 por 100.
Eliminación:
pop_front(): Elimina 0.
pop_back(): Elimina 6.
remove(2): Elimina todas las ocurrencias de 2.
Recorrido: Muestra los elementos restantes de la lista.

Consideraciones adicionales

Eficiencia: Aunque `std::list` permite inserciones y eliminaciones eficientes en cualquier posición, no es tan eficiente como `std::vector` para acceso aleatorio. Si necesitas acceder a elementos por índice frecuentemente, considera usar std::vector.

Operaciones adicionales: `std::list` ofrece otras funciones útiles como ***sort***, ***reverse***, ***unique***, entre otras, que permiten manipular la lista de diversas maneras.

Memoria: Cada elemento de una `std::list` almacena, además del dato, dos punteros (anterior y siguiente), lo que implica un mayor consumo de memoria comparado con otros contenedores como std::vector.

Iteradores: Es importante manejar correctamente los iteradores, especialmente al insertar o eliminar elementos, ya que algunas operaciones pueden invalidar los iteradores.

C++ Moderno: Utilizar características de C++11 y posteriores, como bucles for basados en rangos y funciones lambda, puede hacer que el código sea más limpio y eficiente.

TEMA Pilas y Colas INTRO

Pilas y Colas en C++ utilizando la Biblioteca Estándar de Plantillas (STL). Incluiré cómo crearlas, ingresar datos, modificarlas y eliminarlas, además de ejemplos prácticos.

Introducción a pilas y colas en C++

Las pilas y colas son estructuras de datos fundamentales que siguen un orden específico para el acceso a sus elementos:

Pila (Stack): Sigue el principio LIFO **(Last In, First Out)**. El último elemento en entrar es el primero en salir.
Cola (Queue): Sigue el principio FIFO **(First In, First Out).** El primer elemento en entrar es el primero en salir.

En C++, tanto las pilas como las colas se implementan utilizando contenedores de la STL **(std::stack y std::queue respectivamente).**

Pilas

Creación de una pila

En C++, la pila se implementa utilizando el contenedor ***std::stack***. La pila puede almacenar cualquier tipo de dato, incluyendo tipos primitivos y objetos.

```
#include <iostream>
#include <stack> // Necesario para std::stack

int main() {
	// Crear una pila de enteros
	std::stack<int> miPila;

	return 0;
}
```

Insertar elementos en una pila

Para insertar elementos en una pila, se utiliza la función push, que siempre agrega el elemento en la parte superior de la pila.

```
miPila.push(10); // Inserta 10 en la pila
miPila.push(20); // Inserta 20 en la pila (20 está en la parte superior ahora)
miPila.push(30); // Inserta 30 en la pila (30 está en la parte superior ahora)
```

Acceder y modificar elementos en una pila

En una pila, solo se puede acceder al elemento que está en la parte superior, y esto se realiza mediante la función top.

```
int elementoSuperior = miPila.top(); // Devuelve 30, el elemento en la parte superior
```

Aunque top() permite acceder al elemento superior, no modifica la pila ni elimina el elemento.

Eliminar elementos de una pila

Para eliminar el elemento superior de la pila, se utiliza la función pop.

```
miPila.pop(); // Elimina 30, que estaba en la parte superior
```

Es importante mencionar que **pop()** no devuelve el valor del elemento eliminado, solo lo elimina.

Ejemplo completo de pila

```
#include <iostream>
#include <stack>

int main() {
        // 1. Crear una pila de enteros

        std::stack<int> miPila;

        // 2. Insertar elementos en la pila

        miPila.push(10);
        miPila.push(20);
        miPila.push(30);

        // 3. Acceder al elemento superior y modificarlo

        std::cout << "Elemento en la cima: " << miPila.top() << std::endl; // Muestra 30

        // 4. Eliminar el elemento superior

        miPila.pop();

        // 5. Acceder nuevamente al elemento superior

      std::cout << "Nuevo elemento en la cima: " << miPila.top() << std::endl; // Muestra 20

        return 0;
}
```

Salida esperada:

Elemento en la cima: 30
Nuevo elemento en la cima: 20

Colas

Creación de una cola

En C++, la cola se implementa utilizando el contenedor **std::queue**. Una cola puede almacenar cualquier tipo de dato.

```
#include <iostream>
#include <queue> // Necesario para std::queue

int main() {
        // Crear una cola de enteros

        std::queue<int> miCola;

        return 0;
}
```

Insertar elementos en una cola

Para insertar elementos en una cola, se utiliza la función push, que agrega el elemento al final de la cola.

```
miCola.push(10); // Inserta 10 en la cola
miCola.push(20); // Inserta 20 en la cola (10 es el primer elemento, 20 el último)
miCola.push(30); // Inserta 30 en la cola (10 es el primer elemento, 30 el último)
```

Acceder y modificar elementos en una cola

En una cola, puedes acceder al elemento en el frente (el primero en entrar) con la función front, y al elemento en la parte posterior (el último en entrar) con la función back.

```
int primerElemento = miCola.front(); // Devuelve 10, el primer elemento
int ultimoElemento = miCola.back();    // Devuelve 30, el último elemento
```

Eliminar elementos de una cola

Para eliminar el primer elemento de la cola, se utiliza la función pop.

```
miCola.pop(); // Elimina 10, que estaba al frente
```

Al igual que en la pila, pop() no devuelve el valor del elemento eliminado.

Ejemplo completo de cola

```
#include <iostream>
#include <queue>

int main() {
        // 1. Crear una cola de enteros

        std::queue<int> miCola;

        // 2. Insertar elementos en la cola

        miCola.push(10);
        miCola.push(20);
        miCola.push(30);

        // 3. Acceder al primer y último elemento

        std::cout << "Primer elemento: " << miCola.front() << std::endl; // Muestra 10
        std::cout << "Último elemento: " << miCola.back() << std::endl;  // Muestra 30

        // 4. Eliminar el primer elemento

        miCola.pop();

        // 5. Acceder nuevamente al primer elemento

        std::cout << "Nuevo primer elemento: " << miCola.front() << std::endl; // Muestra 20

       return 0;
}
```

Salida esperada:

Primer elemento: 10
Último elemento: 30
Nuevo primer elemento: 20

Consideraciones adicionales

Tamaño de la estructura: Tanto **std::stack** como **std::queue** proporcionan la función **size()** para obtener el número de elementos en la estructura.

Comprobación de vacío: Puedes utilizar la función **empty()** para comprobar si la pila o cola están vacías.

Adaptadores de contenedores: Tanto **std::stack** como std::queue son adaptadores de contenedores. Por defecto, std::stack utiliza std::deque como contenedor subyacente, y std::queue también usa std::deque. Puedes cambiar el contenedor subyacente si lo deseas, utilizando std::vector o std::list.

Rendimiento: std::stack y std::queue están optimizados para su propósito específico de acceso secuencial, lo que los hace eficientes para las operaciones de inserción y eliminación.