PILAS EN C++

NOTA:
Para trabajar con pilas en C++, es recomendable tener una comprensión sólida de varios conceptos básicos de programación y de C++. Aquí te detallo los temas previos que deberías conocer o dominar antes de abordar pilas:

1. Estructuras de Datos Básicas
Arreglos (Arrays): Entender cómo funcionan los arreglos es fundamental para comprender cómo se almacenan y manipulan datos en estructuras como pilas.
Listas Enlazadas: Aunque no siempre es necesario, si vas a implementar pilas de forma manual, es útil tener una base en listas enlazadas.

2. Punteros
Conceptos de Punteros: Es esencial entender qué son los punteros, cómo se utilizan y cómo pueden apuntar a diferentes tipos de datos.
Punteros a Estructuras o Clases: Si implementas pilas usando listas enlazadas, necesitarás manipular punteros para enlazar nodos de la lista.

3. Memoria Dinámica
new y delete: Necesitas saber cómo gestionar la memoria dinámicamente, ya que las pilas pueden crecer y decrecer en tamaño, y podrías necesitar asignar y liberar memoria manualmente.

4. Estructuras de Control
Condicionales (if, else): Para gestionar situaciones en las que la pila está llena o vacía, necesitarás condicionales.
Bucles (while, for): Los bucles te ayudarán a recorrer la pila o a realizar operaciones repetitivas sobre ella.

5. Clases y Objetos (Programación Orientada a Objetos)
Declaración de Clases: Para implementar pilas manualmente, necesitarás saber cómo crear clases.
Constructores y Destructores: Saber cómo inicializar y destruir objetos correctamente, especialmente cuando trabajas con pilas dinámicas.
Métodos: Implementar los métodos que manipulan la pila (push, pop, peek) en una clase de pila.

6. Funciones
Funciones Básicas: Conocer cómo declarar, definir y usar funciones te ayudará a implementar las operaciones de una pila en forma de métodos o funciones.
Parámetros por Referencia: Especialmente si deseas optimizar el rendimiento cuando trabajas con datos grandes.

7. Estructuras de Datos de la STL (Standard Template Library)
Contenedores en la STL: Conocer los contenedores como std::vector, std::deque y cómo std::stack usa estos contenedores bajo el capó.
Iteradores: Si trabajas con std::stack, también es útil conocer cómo se puede iterar sobre los contenedores subyacentes (aunque stack no permite iteradores directamente).

8. Algoritmos Básicos
Recursividad: Algunas operaciones con pilas se entienden mejor con recursividad, como resolver problemas de tipo "torres de Hanói" o calcular factoriales.
Manejo de Excepciones: Aunque no es obligatorio, saber manejar excepciones puede ser útil en escenarios donde la pila esté vacía o llena (por ejemplo, std::stack no lanza excepciones por defecto, pero en una implementación manual podría ser útil).

Pilas en C++ con struct (estructuras)
Una pila es una estructura de datos que sigue el principio LIFO (Last In, First Out), es decir, el último elemento que entra es el primero que sale. Vamos a implementarla usando una estructura (struct) en C++, lo que te permitirá trabajar directamente con nodos y punteros para gestionar la memoria de la pila.

A continuación, te explicaré cómo crear, insertar, borrar y modificar una pila en C++ usando struct.

1. Definición de la estructura
Vamos a definir un nodo para la pila con un struct. Cada nodo tendrá dos partes:

Dato: el valor que almacenamos en el nodo.
Puntero al siguiente nodo en la pila (enlace al nodo siguiente).
Código de la estructura Nodo:

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

// Definición de la estructura para un nodo de la pila
struct Nodo {
     int dato;             // Valor que almacena el nodo
     Nodo* siguiente;    // Puntero al siguiente nodo
};
```

2. Operaciones básicas en la pila
Vamos a implementar las siguientes operaciones:

- Crear la pila (inicializarla).
- Insertar un elemento en la pila (push).
- Eliminar un elemento de la pila (pop).

- Modificar un elemento específico.
- Mostrar todos los elementos de la pila.

3. Inicializar la pila
Una pila vacía simplemente es una pila cuyo puntero al nodo superior (la cima) es nullptr.
Código:

```
Nodo* pila = nullptr; // Pila vacía
```

4. Insertar un elemento en la pila (push)
Para insertar un nuevo elemento en la pila:

- Creamos un nuevo nodo.
- Asignamos el valor al nodo.
- El puntero siguiente del nuevo nodo debe apuntar al nodo que actualmente es la cima.
- La cima ahora debe ser el nuevo nodo.
-

Código:

```
void push(Nodo*& pila, int valor) {
     Nodo* nuevo_nodo = new Nodo(); // Crear un nuevo nodo
     nuevo_nodo->dato = valor;        // Asignar el valor
     nuevo_nodo->siguiente = pila;   // El siguiente del nuevo nodo apunta a la cima actual
     pila = nuevo_nodo;                 // La cima ahora es el nuevo nodo
     cout << "Elemento " << valor << " agregado a la pila." << endl;
}
```

5. Eliminar un elemento de la pila (pop)
Para eliminar el nodo superior:

- Guardamos una referencia al nodo que está en la cima.
- La cima pasa a ser el siguiente nodo.
- Liberamos la memoria del nodo eliminado.
-

Código:

```
void pop(Nodo*& pila) {
     if (pila != nullptr) {                // Si la pila no está vacía
          Nodo* nodo_a_eliminar = pila; // Apuntamos al nodo a eliminar
          int valor = nodo_a_eliminar->dato; // Guardamos el valor del nodo eliminado
          pila = pila->siguiente;         // La cima ahora apunta al siguiente nodo
          delete nodo_a_eliminar;         // Liberamos la memoria del nodo eliminado
          cout << "Elemento " << valor << " eliminado de la pila." << endl;
     } else {
          cout << "La pila está vacía." << endl;
     }
}
```

6. Modificar un elemento específico en la pila
Para modificar un nodo en la pila:

- Recorremos la pila desde la cima hasta el final.
- Cuando encontramos el valor que queremos modificar, lo actualizamos.

Código:

```
void modificar(Nodo* pila, int valor_actual, int nuevo_valor) {
     Nodo* actual = pila;
     bool encontrado = false;

     while (actual != nullptr) {
          if (actual->dato == valor_actual) {
               actual->dato = nuevo_valor;
               cout << "Elemento " << valor_actual << " modificado a " << nuevo_valor << "." <<
endl;
               encontrado = true;
               break;
          }
          actual = actual->siguiente;
     }

     if (!encontrado) {
          cout << "Elemento " << valor_actual << " no encontrado en la pila." << endl;
     }
}
```

7. Mostrar los elementos de la pila

Recorremos la pila desde la cima y mostramos cada elemento.

Código:

```
void mostrarPila(Nodo* pila) {
	Nodo* actual = pila;

	cout << "Elementos en la pila: ";
	while (actual != nullptr) {
		cout << actual->dato << " ";
		actual = actual->siguiente;
	}
	cout << endl;
}
```

8. Programa completo
Aquí tienes un ejemplo de cómo usar todas las operaciones de la pila:

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

// Definición de la estructura para un nodo de la pila
struct Nodo {
	int dato;
	Nodo* siguiente;
};

// Funciones para manejar la pila
void push(Nodo*& pila, int valor);
void pop(Nodo*& pila);
void modificar(Nodo* pila, int valor_actual, int nuevo_valor);
void mostrarPila(Nodo* pila);

int main() {
	Nodo* pila = nullptr;  // Inicializamos la pila vacía

	// Insertar elementos en la pila
	push(pila, 10);
	push(pila, 20);
	push(pila, 30);
	mostrarPila(pila);

	// Modificar un elemento en la pila
	modificar(pila, 20, 25);
	mostrarPila(pila);

	// Eliminar un elemento de la pila
	pop(pila);
	mostrarPila(pila);

	return 0;
}

// Implementación de las funciones
void push(Nodo*& pila, int valor) {
	Nodo* nuevo_nodo = new Nodo();
	nuevo_nodo->dato = valor;
	nuevo_nodo->siguiente = pila;
	pila = nuevo_nodo;
	cout << "Elemento " << valor << " agregado a la pila." << endl;
}

void pop(Nodo*& pila) {
	if (pila != nullptr) {
		Nodo* nodo_a_eliminar = pila;
		int valor = nodo_a_eliminar->dato;
		pila = pila->siguiente;
		delete nodo_a_eliminar;
		cout << "Elemento " << valor << " eliminado de la pila." << endl;
	} else {
		cout << "La pila está vacía." << endl;
	}
}

void modificar(Nodo* pila, int valor_actual, int nuevo_valor) {
	Nodo* actual = pila;
	bool encontrado = false;
```

```
    while (actual != nullptr) {
        if (actual->dato == valor_actual) {
            actual->dato = nuevo_valor;
            cout << "Elemento " << valor_actual << " modificado a " << nuevo_valor << "." <<
endl;
            encontrado = true;
            break;
        }
        actual = actual->siguiente;
    }

    if (!encontrado) {
        cout << "Elemento " << valor_actual << " no encontrado en la pila." << endl;
    }
}

void mostrarPila(Nodo* pila) {
    Nodo* actual = pila;
    cout << "Elementos en la pila: ";
    while (actual != nullptr) {
        cout << actual->dato << " ";
        actual = actual->siguiente;
    }
    cout << endl;
}
```

Explicación de las operaciones:
push: Agrega un elemento en la cima de la pila.
pop: Elimina el elemento en la cima de la pila.
modificar: Busca un valor específico en la pila y lo cambia por otro.
mostrarPila: Muestra todos los elementos de la pila desde la cima.
Este código te da una implementación básica de una pila utilizando struct. Puedes ampliarlo según tus necesidades, como agregar más validaciones o mejorar la gestión de errores.

Vamos a explorar el concepto de pilas (stacks) en C++, con una explicación completa y ejemplos prácticos.

¿Qué es una Pila?
Una pila es una estructura de datos LIFO (Last In, First Out), **lo que significa que el último elemento en entrar es el primero en salir**. Imagina una pila de platos: *el último plato que pones arriba es el primero que quitas*.

Operaciones principales en una pila
Push: Agregar un elemento a la pila.
Pop: Eliminar el elemento del tope de la pila.
Top (Peek): Obtener el valor del elemento en el tope sin eliminarlo.
Empty: Verificar si la pila está vacía.

Implementación de Pilas en C++
En C++, las pilas se pueden implementar usando el contenedor stack de la STL (Standard Template Library), pero también podemos implementarlas manualmente usando arreglos o listas enlazadas.

Usando la STL (std::stack)
El contenedor std::stack nos facilita mucho la implementación de pilas. Aquí te muestro cómo funciona:

```
#include <iostream>
#include <stack> // Incluimos la biblioteca para usar std::stack

int main() {
    std::stack<int> pila;

    // Push - Agregar elementos
    pila.push(10);
    pila.push(20);
    pila.push(30);

    // Mostrar el elemento en la parte superior
    std::cout << "Elemento en el tope: " << pila.top() << std::endl;

    // Pop - Eliminar el elemento superior
    pila.pop();
    std::cout << "Después de hacer pop, el tope es: " << pila.top() << std::endl;

    // Verificar si la pila está vacía
    if (pila.empty()) {
```

```
          std::cout << "La pila está vacía" << std::endl;
     } else {
          std::cout << "La pila NO está vacía" << std::endl;
     }

     return 0;
}
```

Explicación del código
Creación de la pila:

```
std::stack<int> pila;
```

Aquí, estamos creando una pila de enteros.

Push:

Agregamos tres elementos a la pila.

```
pila.push(10);
pila.push(20);
pila.push(30);
```

Top:
Mostramos el elemento superior sin eliminarlo:

```
std::cout << "Elemento en el tope: " << pila.top() << std::endl;
```

Pop:
Eliminamos el elemento superior:

```
pila.pop();
```

Empty:
Verificamos si la pila está vacía.

```
if (pila.empty()) {
```

Implementación Manual de Pilas Usando Arreglos
Veamos ahora cómo podemos implementar una pila manualmente usando arreglos.

```
#include <iostream>
#define MAX 1000 // Tamaño máximo de la pila

class Pila {
     int tope;

public:
     int arr[MAX]; // Arreglo para almacenar la pila

     Pila() { tope = -1; } // Constructor para inicializar el tope

     bool push(int x);
     int pop();
     int peek();
     bool estaVacia();
};

// Función para agregar un elemento a la pila
bool Pila::push(int x) {
     if (tope >= (MAX - 1)) {
          std::cout << "Desbordamiento de pila\n";
          return false;
     } else {
          arr[++tope] = x;
          std::cout << x << " se agregó a la pila\n";
          return true;
     }
}

// Función para eliminar el elemento superior
int Pila::pop() {
     if (tope < 0) {
          std::cout << "Subdesbordamiento de pila\n";
          return 0;
     } else {
          int x = arr[tope--];
          return x;
```

```
    }
}

// Función para obtener el elemento superior
int Pila::peek() {
    if (tope < 0) {
        std::cout << "La pila está vacía\n";
        return 0;
    } else {
        int x = arr[tope];
        return x;
    }
}

// Función para verificar si la pila está vacía
bool Pila::estaVacia() {
    return (tope < 0);
}

// Programa principal
int main() {
    Pila pila;
    pila.push(10);
    pila.push(20);
    pila.push(30);

    std::cout << pila.pop() << " fue eliminado de la pila\n";

    std::cout << "El elemento en el tope es: " << pila.peek() << std::endl;

    if (pila.estaVacia())
        std::cout << "La pila está vacía\n";
    else
        std::cout << "La pila no está vacía\n";

    return 0;
}
```

Explicación del código
Constructor: Inicializa el tope de la pila a -1, indicando que está vacía.

Método push:
Agrega un elemento al arreglo, aumentando el tope en cada inserción.

Método pop:
Elimina el elemento superior y lo retorna. Si la pila está vacía, devuelve 0 y muestra un mensaje de error.

Método peek:
Retorna el valor del elemento superior sin eliminarlo.

Método estaVacia:
Retorna true si la pila está vacía, es decir, si el tope es menor que 0.

Implementación Manual de Pilas Usando Listas Enlazadas
También podemos implementar una pila usando listas enlazadas, lo que nos da la ventaja de un tamaño dinámico. Aquí te dejo un ejemplo de cómo hacerlo:

```
#include <iostream>

class Nodo {
public:
    int dato;
    Nodo* siguiente;
};

// Clase que representa una pila
class Pila {
    Nodo* tope;

public:
    Pila() { tope = nullptr; }

    void push(int x);
    int pop();
    int peek();
    bool estaVacia();
```

```
};

// Función para agregar un elemento a la pila
void Pila::push(int x) {
      Nodo* nuevoNodo = new Nodo();
      nuevoNodo->dato = x;
      nuevoNodo->siguiente = tope;
      tope = nuevoNodo;
      std::cout << x << " se agregó a la pila\n";
}

// Función para eliminar el elemento superior
int Pila::pop() {
      if (tope == nullptr) {
            std::cout << "Subdesbordamiento de pila\n";
            return 0;
      } else {
            Nodo* temp = tope;
            tope = tope->siguiente;
            int x = temp->dato;
            delete temp;
            return x;
      }
}

// Función para obtener el elemento superior
int Pila::peek() {
      if (tope == nullptr) {
            std::cout << "La pila está vacía\n";
            return 0;
      } else {
            return tope->dato;
      }
}

// Función para verificar si la pila está vacía
bool Pila::estaVacia() {
      return tope == nullptr;
}

// Programa principal
int main() {
      Pila pila;
      pila.push(10);
      pila.push(20);
      pila.push(30);

      std::cout << pila.pop() << " fue eliminado de la pila\n";

      std::cout << "El elemento en el tope es: " << pila.peek() << std::endl;

      if (pila.estaVacia())
            std::cout << "La pila está vacía\n";
      else
            std::cout << "La pila no está vacía\n";

      return 0;
}
```

Explicación del código

Clase Nodo:
Cada nodo tiene un valor (dato) y un puntero al siguiente nodo.

Clase Pila:
La pila utiliza nodos para agregar y eliminar elementos dinámicamente.

Push:
Crea un nuevo nodo, lo enlaza al nodo anterior y actualiza el tope.

Pop:
Elimina el nodo superior y actualiza el puntero tope.

Peek:
Retorna el valor del nodo superior sin eliminarlo.

Empty:
Verifica si la pila está vacía.

Resumen

Las pilas son una estructura de datos fundamental en programación. En C++, podemos implementarlas de varias formas:

Usando la STL (std::stack), *que es muy fácil de usar.*
Implementaciones manuales usando arreglos o listas enlazadas, para un mayor control sobre el comportamiento y el tamaño.