**Representación en memoria estática y dinámica**

1. **Introducción:**

Una pila es una estructura de datos lineal que sigue el principio de LIFO (Last In, First Out), lo que significa que el último elemento agregado a la pila es el primero en ser eliminado. Las operaciones principales en una pila son la inserción (push), que agrega un elemento a la parte superior de la pila, y la eliminación (pop), que elimina el elemento superior de la pila.

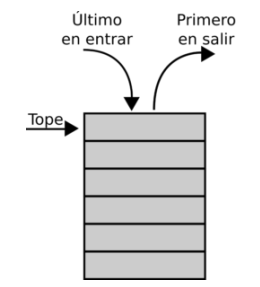


Imagen 1: Estructura de la Pila

**2. Representación en memoria estática:**

En la representación estática de una pila, se reserva un bloque de memoria de tamaño fijo durante la compilación del programa.

* La cantidad de memoria asignada para la pila se determina antes de tiempo y no cambia durante la ejecución del programa.
* Se utiliza un arreglo estático para almacenar los elementos de la pila.
* El acceso a los elementos se realiza mediante índices.

Una desventaja es que la capacidad de la pila está limitada por el tamaño del arreglo estático, lo que puede llevar a problemas de desbordamiento si se intenta agregar más elementos de los que el arreglo puede contener.

**2.1 Ejemplo:**

En la implementación de una pila utilizando memoria estática en C++, se puede utilizar un arreglo para almacenar los elementos de la pila.

***PilaEstatica.h:***

#ifndef PILA\_ESTATICA\_H

#define PILA\_ESTATICA\_H

#include <iostream>

#define TAM\_MAX 100 // Tamaño máximo de la pila

**class** **PilaEstatica** {

**private:**

**int** arreglo[TAM\_MAX];

**int** tope; // Índice del elemento superior de la pila

**public:**

PilaEstatica();

**void** **empujar**(**int** dato);

**void** **sacar**();

**int** **elementoTope**();

**bool** **estaVacia**();

};

#endif

***PilaEstatica.cpp:***

#include "PilaEstatica.h"

PilaEstatica::PilaEstatica() {

tope = -**1**; // Inicializar la pila como vacía

}

**void** PilaEstatica::empujar(**int** dato) {

**if** (tope == TAM\_MAX - **1**) {

std::cout << "Error: La pila está llena." << std::endl;

**return**;

}

arreglo[++tope] = dato;

}

**void** PilaEstatica::sacar() {

**if** (tope == -**1**) {

std::cout << "Error: La pila está vacía." << std::endl;

**return**;

}

tope--;

}

**int** PilaEstatica::elementoTope() {

**if** (tope == -**1**) {

std::cout << "Error: La pila está vacía." << std::endl;

**return** -**1**;

}

**return** arreglo[tope];

}

**bool** PilaEstatica::estaVacia() {

**return** tope == -**1**;

}

***main.cpp***

#include <iostream>

#include "PilaEstatica.h"

**int** **main**() {

PilaEstatica pila;

pila.empujar(**10**);

pila.empujar(**20**);

pila.empujar(**30**);

std::cout << "Elemento superior de la pila: " << pila.elementoTope() << std::endl;

pila.sacar();

std::cout << "Elemento superior de la pila después de sacar(): " << pila.elementoTope() << std::endl;

**return** **0**;

}

1. **Representación en memoria dinámica:**

En la representación dinámica de una pila, se utiliza la asignación de memoria dinámica durante la ejecución del programa.

* No se reserva un tamaño fijo de memoria durante la compilación; en su lugar, la memoria se asigna y libera según sea necesario durante la ejecución del programa.
* Se utiliza una estructura de datos enlazada, como una lista enlazada, para almacenar los elementos de la pila.
* Cada elemento de la pila se almacena en un nodo de la lista enlazada, y los nodos están enlazados entre sí mediante punteros.

Esta representación permite que la pila crezca y se reduzca dinámicamente según sea necesario, lo que es útil cuando la cantidad de elementos de la pila puede variar durante la ejecución del programa.

Una desventaja potencial es el costo adicional de la asignación y liberación de memoria dinámica, así como la necesidad de gestionar correctamente la liberación de la memoria para evitar fugas de memoria.

* 1. **Ejemplo:**

En la implementación de una pila utilizando memoria dinámica en C++, se puede utilizar una lista enlazada

***PilaDinamica.h***

#ifndef PILA\_DINAMICA\_H

#define PILA\_DINAMICA\_H

#include <iostream>

**class** **PilaDinamica** {

**private:**

**class** **Nodo** {

**public:**

**int** dato;

Nodo\* siguiente;

Nodo(**int** valor) : dato(valor), siguiente(nullptr) {}

};

Nodo\* tope; // Puntero al elemento superior de la pila

**public:**

PilaDinamica();

**void** **empujar**(**int** dato);

**void** **sacar**();

**int** **elementoTope**();

**bool** **estaVacia**();

~PilaDinamica(); // Destructor

};

#endif // PILA\_DINAMICA\_H

***PilaDinamica.cpp***

#include "PilaDinamica.h"

PilaDinamica::PilaDinamica() {

tope = nullptr; // Inicializar la pila como vacía

}

**void** PilaDinamica::empujar(**int** dato) {

Nodo\* nuevoNodo = **new** Nodo(dato);

nuevoNodo->siguiente = tope;

tope = nuevoNodo;

}

**void** PilaDinamica::sacar() {

**if** (tope == nullptr) {

std::cout << "Error: La pila está vacía." << std::endl;

**return**;

}

Nodo\* temp = tope;

tope = tope->siguiente;

**delete** temp;

}

**int** PilaDinamica::elementoTope() {

**if** (tope == nullptr) {

std::cout << "Error: La pila está vacía." << std::endl;

**return** -**1**;

}

**return** tope->dato;

}

**bool** PilaDinamica::estaVacia() {

**return** tope == nullptr;

}

PilaDinamica::~PilaDinamica() {

**while** (tope != nullptr) {

sacar();

}

}

***main.cpp***

#include <iostream>

#include "PilaDinamica.h"

**int** **main**() {

PilaDinamica pila;

pila.empujar(**10**);

pila.empujar(**20**);

pila.empujar(**30**);

std::cout << "Elemento superior de la pila: " << pila.elementoTope() << std::endl;

pila.sacar();

std::cout << "Elemento superior de la pila después de sacar(): " << pila.elementoTope() << std::endl;

**return** **0**;

}

**4. Conclusiones**

* La implementación utilizando memoria estática es más simple y directa, ya que no requiere la gestión manual de la memoria.
* Sin embargo, la implementación utilizando memoria dinámica permite que la pila crezca dinámicamente según sea necesario, lo que puede ser más eficiente en términos de uso de memoria en casos donde la cantidad de elementos de la pila es variable.
* La elección entre memoria estática y dinámica depende de los requisitos específicos del programa y del rendimiento deseado. En general, si la cantidad máxima de elementos de la pila es conocida y limitada, la memoria estática puede ser preferible por su simplicidad y eficiencia. Por otro lado, si la cantidad de elementos puede variar o es desconocida, la memoria dinámica puede ser más adecuada.

**5. Referencias**

* Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.

<https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf>

* Drozdek, Adam. "Data Structures and Algorithms in C++." 5th Edition, Cengage Learning, 2019.

<https://itlectures.ro/wpcontent/uploads/2016/04/AdamDrozdek__DataStructures_and_Algorithms_in_C_4Ed.pdf>