ESAT INNOVATION SCHOOL | ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

MEMORY STACK

Nombre Asignatura:Algoritmos y estructuras de datos
Profesor:Iván Sancho
Curso: 2025/2026
Autor/es:Pau Alvarez Belenguer

Índice/Index

1.- INTRODUCCIÓN

En este documento vamos a ver, de forma clara y bastante práctica, qué es exactamente la memory stack (o pila de memoria), para qué sirve y cómo funciona internamente. Este concepto es básico en programación de sistemas y aparece constantemente cuando trabajamos con funciones, variables locales y llamadas anidadas.

La stack es una zona concreta de la memoria del programa que se gestiona sola durante la ejecución. A diferencia del heap, donde somos nosotros quienes pedimos y liberamos memoria, la stack trabaja de forma automática siguiendo un patrón ordenado y muy eficiente.

1.1 Definición y propósito de la memory stack

La pila de memoria es una estructura de datos que funciona siguiendo la regla LIFO (Last In, First Out): lo último en entrar es lo primero en salir. Su misión principal es guardar:

- Variables locales de cada función
- Los parámetros que se le pasan a las funciones
- La dirección a la que el programa debe volver después de una llamada
- El estado necesario para continuar la ejecución (como punteros de marco y otros datos)

La gran ventaja de la stack es que “se limpia sola”: cuando una función termina, todo lo que reservó desaparece automáticamente. Esto evita muchos problemas típicos como las fugas de memoria. Las ventajas principales son: Muy rápida de usar, gestión automática, operaciones en tiempo constante y muy eficiente para la caché del procesador. De todas formas también tiene sus limitaciones como: tamaño limitado, no permite estructuras con tamaño realmente dinámico, las variables solo viven mientras la función está activa.

2.- Estructura de la pila

Internamente, la stack está organizada de forma jerárquica y muy lógica, pensada para encajar perfectamente con el funcionamiento de las llamadas a funciones.

2.1 Organización LIFO

El funcionamiento LIFO hace que la pila se comporte como una torre de platos: el último que colocas es el primero que quitas.

Este modelo encaja genial con las llamadas a función porque cada vez que llamas a una función, se crea un “bloque” nuevo en la parte superior , cuando la función termina, ese bloque desaparece y las funciones anidadas se gestionan solas gracias a este sistema

Ejemplo rápido:

main() llama a A() → stack: [main] [A]
A() llama a B() → stack: [main] [A] [B]
B() termina → stack: [main] [A]
A() termina → stack: [main]

2.2 Stack Pointer y Frame Pointer

El sistema utiliza dos registros clave para manejar la stack:

-Stack Pointer (SP): Apunta siempre al tope de la pila, cambia con cada PUSH o POP, en x86-64 es el registro RSP y la stack crece hacia direcciones más bajas

-Frame Pointer (FP o BP): Marca la base del frame de la función actual, permite acceder a variables y parámetros con offsets fijos y en x86-64 es RBP

2.3 Stack Frames

Cada vez que llamamos a una función, se crea un stack frame, un paquete de información que incluye los parámetros, la dirección de retorno (desde donde ha sido llamada para saber volver) , el frame pointer previo, las variables locales y finalmente un espacio temporal para cálculos internos. Cuando esta función termina, todo el Stack Frame se elimina.

3.- Operaciones Básicas

Aunque la stack pueda parecer una estructura muy ligada al hardware, en realidad su funcionamiento se basa en un conjunto de operaciones sorprendentemente simples. Estas operaciones permiten que la pila sea extremadamente rápida y predecible, cualidades que la hacen ideal para gestionar funciones y su información interna.

3.1 PUSH

La operación push consiste en añadir un nuevo elemento a la pila. Cuando pensamos en “apilar”, solemos imaginar añadir algo encima de una torre; en la stack pasa lo mismo, con la diferencia de que la memoria crece hacia direcciones más bajas. Cada vez que hacemos un push, el Stack Pointer (SP) se mueve para abrir hueco y después se escribe el valor correspondiente.

Este gesto ocurre en una fracción de tiempo minúscula y siempre con la misma velocidad. El procesador está tan optimizado para este tipo de operaciones que prácticamente se realizan de forma instantánea. En esencia, el push es simplemente “guardar este dato justo donde apunta la pila”.

3.2 POP

El pop es la operación complementaria: saca el último elemento que se añadió. Aquí la idea es leer el valor que está actualmente en la cima y, acto seguido, mover el SP para que la pila lo considere “eliminado”. Curiosamente, el dato suele seguir ahí físicamente hasta que se sobrescribe, pero a nivel lógico deja de existir en cuanto el SP se mueve.

Esto mantiene coherente la idea del funcionamiento LIFO: lo último que entró es lo primero que sale. Y, al igual que el push, esta operación ocurre en tiempo constante.

3.3 TOP o PEEK

La operación top (a veces llamada peek) es una especie de vistazo rápido a la cima de la pila sin alterar nada. Es útil para comprobar cuál es el elemento que está en uso sin modificar la pila. Internamente es solo una lectura simple, sin mover el SP, lo que hace que sea igual de eficiente que las anteriores.

3.4 Una secuencia típica de operaciones

En la práctica, estas operaciones se combinan constantemente. Al empezar, la pila contiene un valor inicial. Después se pueden ir añadiendo elementos con varios push, consultar con top sin alterar nada y finalmente recuperar valores con pop. Todo esto ocurre de forma ordenada y automatizada, lo que permite que la pila sea muy predecible.

Si en algún momento intentamos meter más datos de los que caben —algo poco habitual pero posible si abusamos de la recursión o creamos estructuras muy grandes dentro de funciones— aparece un stack overflow, un error que indica que hemos sobrepasado el límite asignado. En el extremo opuesto, si intentamos sacar un valor cuando la pila está vacía, ocurre un stack underflow, aunque este caso es mucho más raro en programas bien escritos.

4.- Llamadas a funciones

Aunque las operaciones básicas ya muestran cómo funciona la pila, su verdadero protagonismo aparece cuando hablamos de llamadas a funciones. Cada vez que una función se ejecuta, necesita su propio “espacio de trabajo”: un lugar donde guardar sus parámetros, sus variables locales y cierta información sobre cómo volver al punto exacto donde había quedado el programa.

La stack es perfecta para esto porque nos permite crear y destruir estos espacios de forma automática y en el orden correcto.

4.1 Cómo se construye una llamada a función

El proceso empieza cuando una función llama a otra. Lo primero que se hace es preparar los argumentos: dependiendo del sistema, algunos irán en registros y otros se colocarán en la pila. Después se ejecuta la instrucción CALL, que no solo salta al código de la función llamada, sino que antes guarda en la pila la dirección a la que deberá volver al terminar.

Al entrar en la función, empieza el llamado prólogo: se guarda el Frame Pointer anterior, se establece uno nuevo y se reserva espacio para variables locales. Es como si la función extendiera la pila un poquito para crear su propio “mini escritorio”.

4.2 Parámetros y variables locales

Cada función accede tanto a los parámetros como a sus variables mediante posiciones relativas al Frame Pointer. Esto permite que, independientemente de cuántos elementos se hayan metido en la pila antes, la función siempre sepa exactamente dónde está cada uno de sus datos.

Dependiendo de la convención de llamada (por ejemplo, cdecl, stdcall o la convención x64 moderna), los parámetros pueden pasar por la pila o por registros específicos, pero al final todos terminan siendo accesibles de forma parecida desde el frame de la función.

4.3 La dirección de retorno

Uno de los elementos más importantes dentro del frame es la dirección de retorno. Este dato le dice al procesador exactamente dónde debe continuar el programa una vez que la función termine. Es tan crítico que muchos de los ataques clásicos de desbordamiento de buffer se basaban precisamente en sobrescribir esta dirección para desviar el flujo del programa. Hoy en día existen mecanismos como los stack canaries o la aleatorización del espacio de direcciones (ASLR) para frustrar estos intentos.

4.4 Cómo se realiza el retorno

Cuando la función ha hecho su trabajo, llega el epílogo. Este proceso consiste básicamente en revertir todo lo que se hizo al entrar: se restauran los valores previos del Frame Pointer y del Stack Pointer, y luego se ejecuta la instrucción RET. Esta instrucción toma la dirección de retorno (que la CALL dejó guardada anteriormente) y salta a ese punto, devolviendo así el control a la función que llamó.

El valor de retorno suele colocarse en un registro especial (como RAX en x86-64), de manera que el caller pueda usarlo inmediatamente al continuar la ejecución.

5. BIBLIOGRAFÍA

Recursos online:

INTEL CORPORATION. Intel® 64 and IA-32 Architectures Software Developer's Manual [en línea]. 2024 [consulta: 15 enero 2025]. Disponible en: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
GNU PROJECT. GCC Calling Conventions [en línea]. 2024 [consulta: 15 enero 2025]. Disponible en: https://gcc.gnu.org/onlinedocs/

Documentación técnica:

MICROSOFT. x64 Calling Convention [en línea]. Microsoft Docs, 2024 [consulta: 15 enero 2025]. Disponible en: https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention
AGNER FOG. Calling conventions for different C++ compilers and operating systems [en línea]. 2024 [consulta: 15 enero 2025]. Disponible en: https://www.agner.org/optimize/calling_conventions.pdf

Material académico:

SANCHO, Iván. Apuntes de Algoritmos y Estructuras de Datos. ESAT Valencia, 2025. Material del curso.