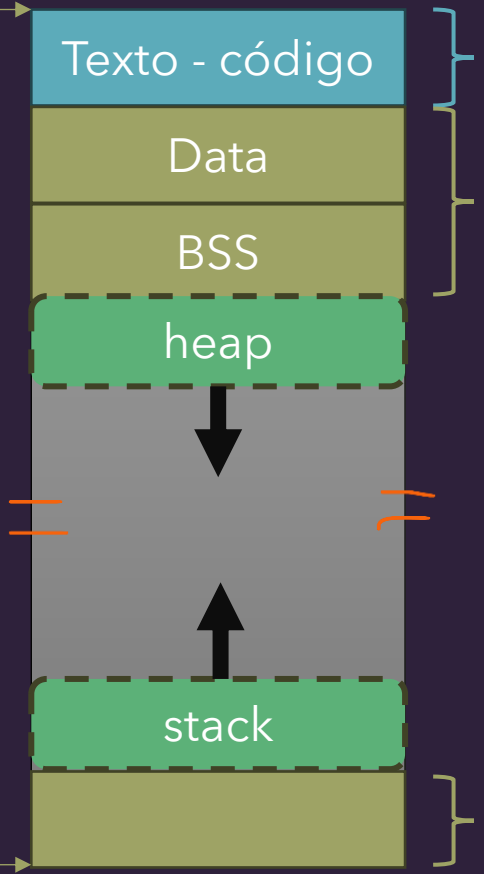
Gestión de memoria dinámica en C

Autor: Ing. Weber
INFO II – UTN FRH

Distribución de memoria de los programas en C

Un programa en C, al ejecutarse, divide su espacio de memoria en **segmentos** bien definidos. Los principales segmentos son:



BSS (block started by symbol)

Text segment

Data segment

Data: datos inicializados
BSS: datos no inicializados

➡Área Estática

Argumentos de línea de comandos y variables de entorno.

Área estática

```
  GNU nano 6.2                                  mem1.c
#include <stdio.h>

int main (void)
{
        return 0;
}
```

```
info2@info2-virtual-machine:~/Desktop/info2$ gcc mem1.c -o mem1
info2@info2-virtual-machine:~/Desktop/info2$ size mem1
   text    data     bss     dec     hex filename
   1228     544       8    1780     6f4 mem1
```

Es recomendable compilar en alguna distribución de Linux para poder verificar estos ejemplos. O si prefieren Windows, pueden usar mingw64 y compilar desde terminal.

Área estática

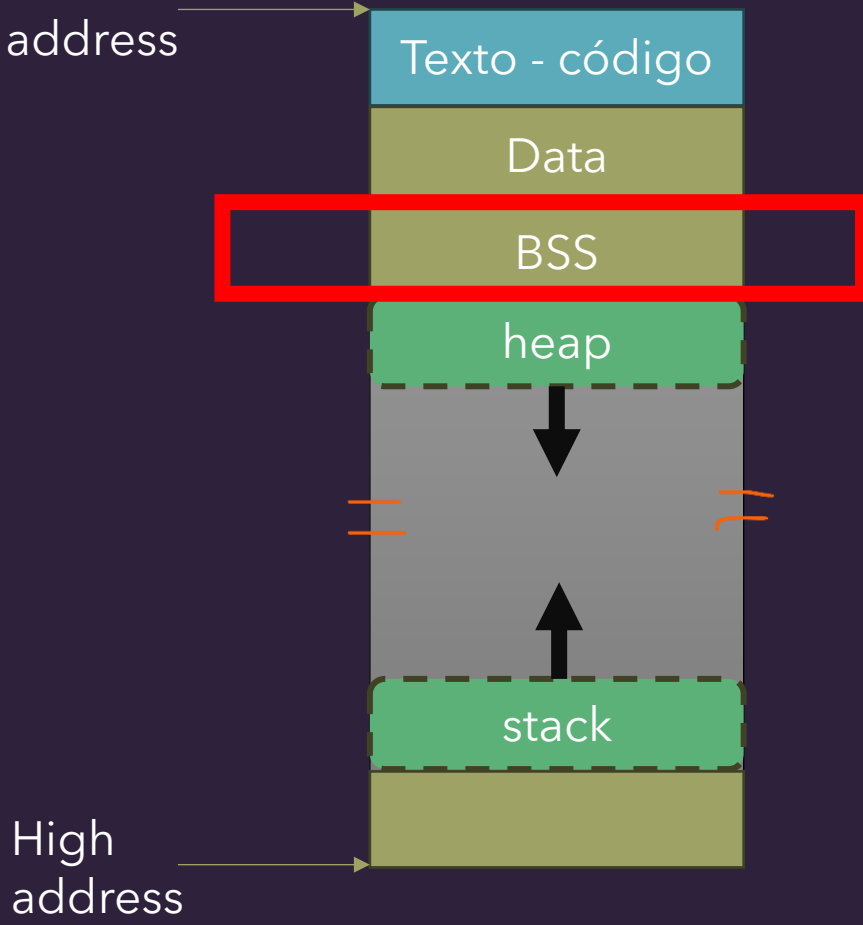
```
 GNU nano 6.2                                  mem2.c *
#include <stdio.h>

int g_x ;
int g_y ;

int main (void)
{
        return 0;
}
```

```
   text    data     bss     dec     hex filename
   1228     544       8    1780     6f4 mem1
```

Original (mem1)

```
info2@info2-virtual-machine:~/Desktop/info2$ gcc mem2.c -o mem2
info2@info2-virtual-machine:~/Desktop/info2$ size mem2
   text    data     bss     dec     hex filename
   1228     544      16    1788     6fc mem2
```

Actual



Área estática

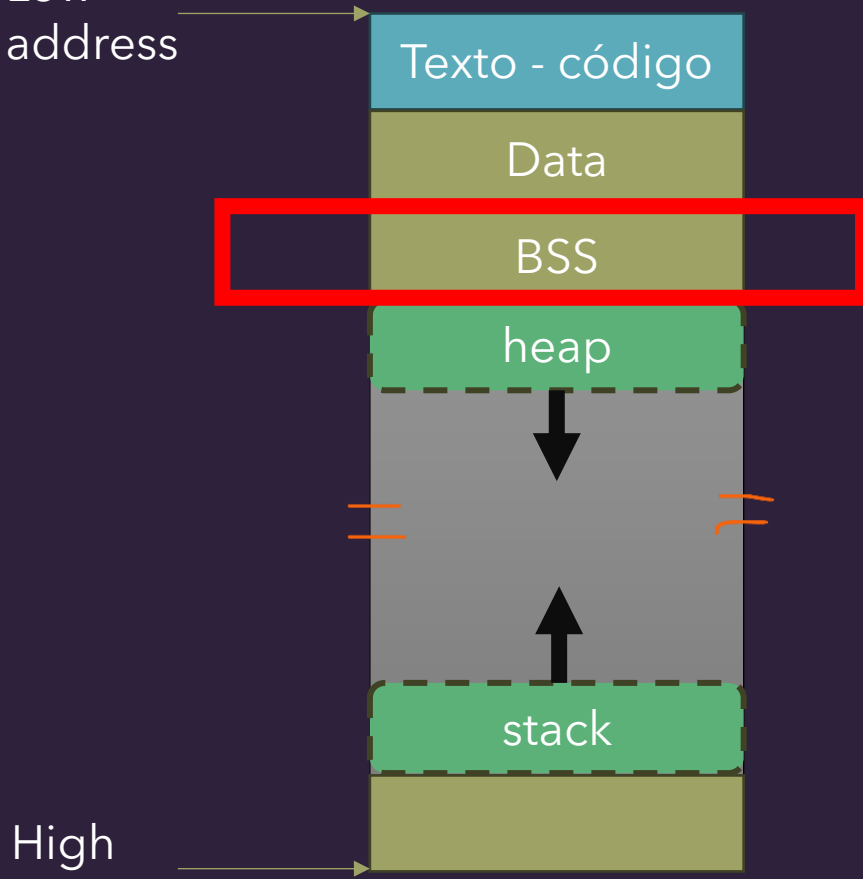
```
  GNU nano 6.2                                  mem3.c
#include <stdio.h>

int g_x ;

int main (void)
{
        static int y ;
        return 0;
}
```

```
   text    data     bss     dec     hex filename
   1228     544       8    1780     6f4 mem1
```

Original (mem1)

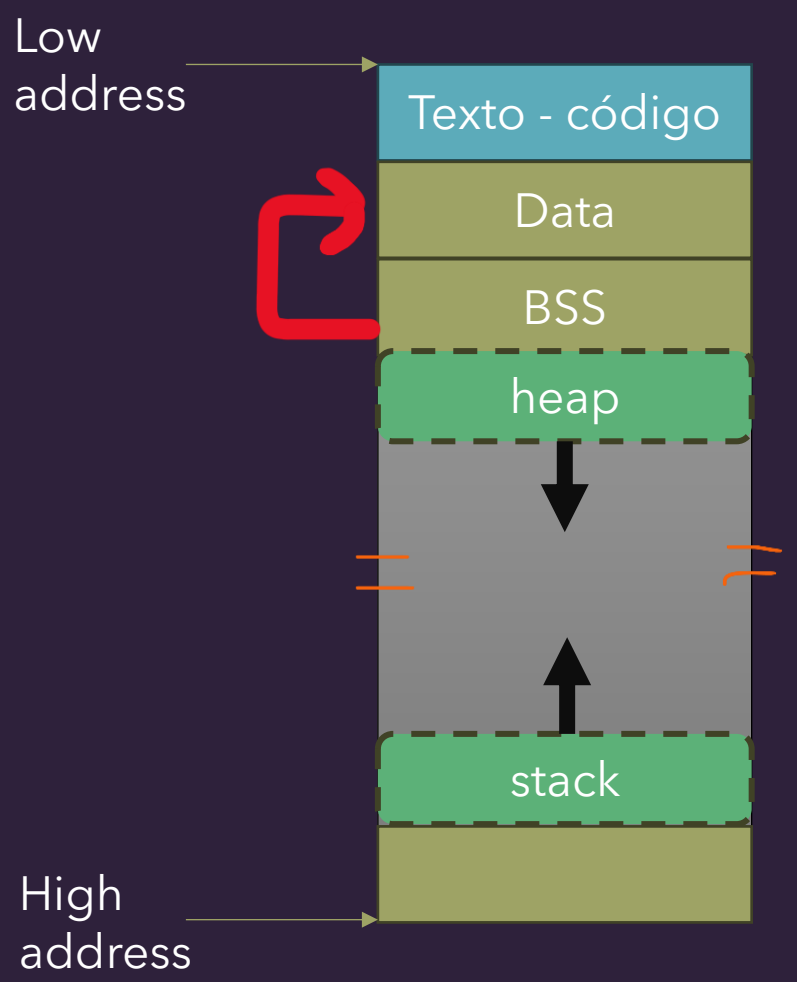
```
info2@info2-virtua.-machine:~/Desktop/info2$ gcc mem3.c -o mem3
info2@info2-virtua.-machine:~/Desktop/info2$ size mem3
   text    data     bss     dec     hex filename
   1228     544      16    1788     6fc mem3
```

Actual



Área estática

```
  GNU nano 6.2                                  mem4.c
#include <stdio.h>

int g_x ;

int main (void)
{
        static int y = 22 ;
        return 0;
}
```



Anterior (mem3)

```
   text    data     bss     dec     hex filename
   1228     544      16    1788     6fc mem3
```

```
info2@info2-virtual-machine:~/Desktop/info2$ gcc mem4.c -o mem4
info2@info2-virtual-machine:~/Desktop/info2$ size mem4
   text    data     bss     dec     hex filename
   1228     548      12    1788     6fc mem4
```

Actual

Área estática

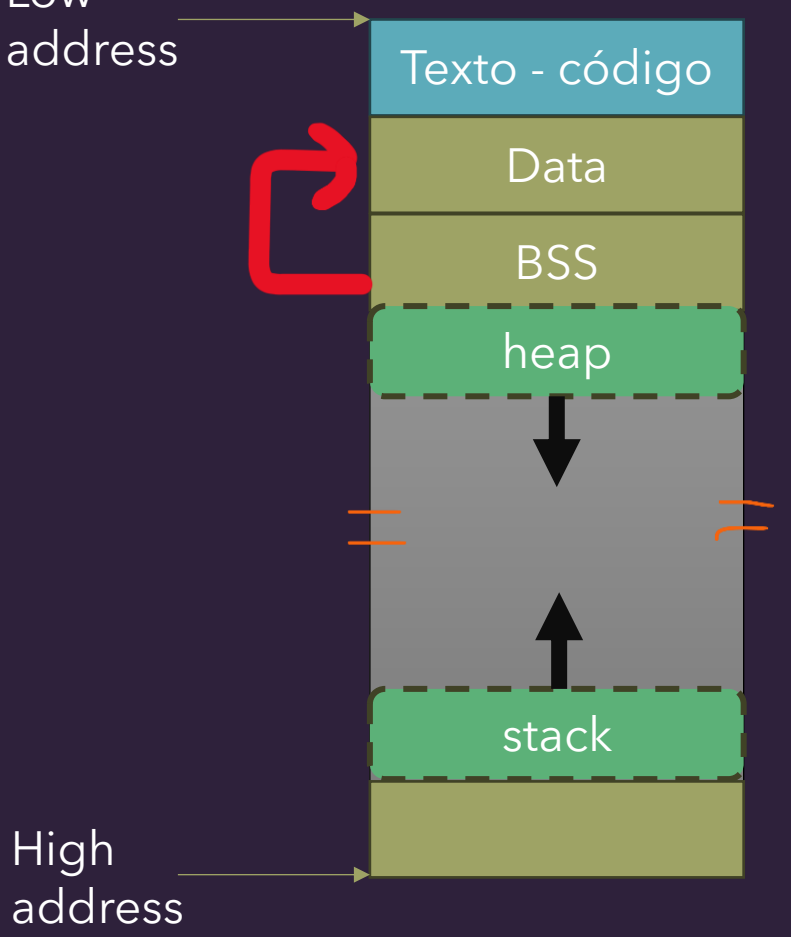
```
  GNU nano 6.2                                  mem5.c
#include <stdio.h>

int g_x = 14 ;

int main (void)
{
        static int y = 22 ;
        return 0;
}
```

Anterior (mem3)

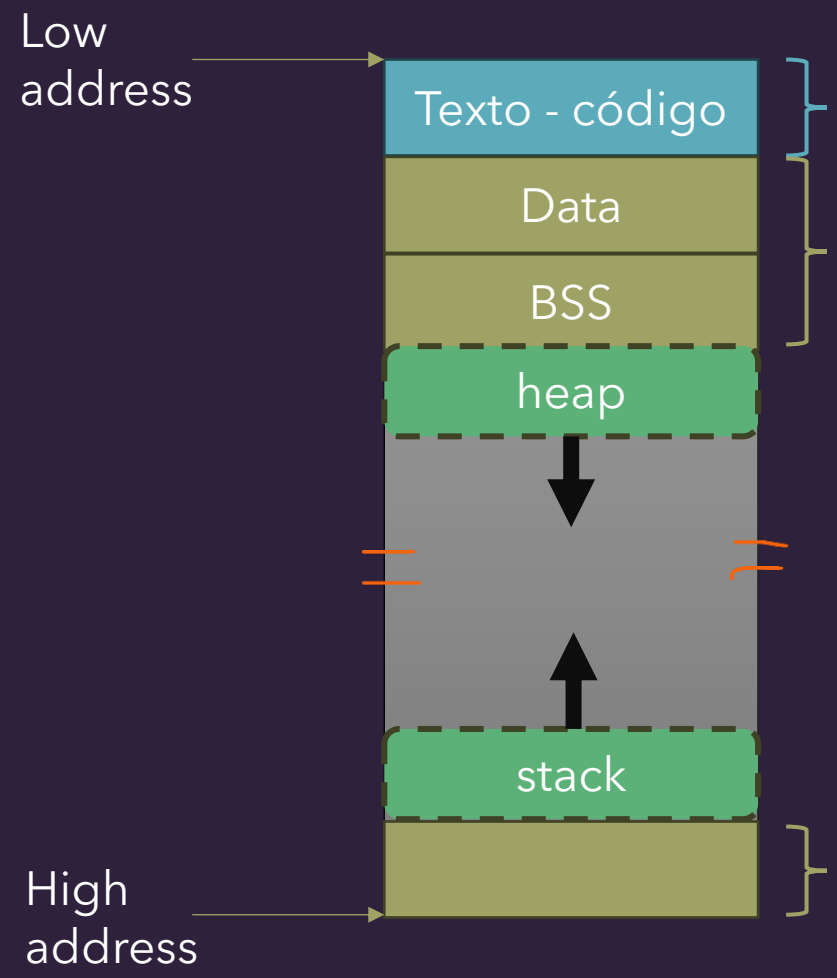
```
   text    data     bss     dec     hex filename
   1228     544      16    1788     6fc mem3
```

```
info2@info2-virtual-machine:~/Desktop/info2$ gcc mem5.c -o mem5
info2@info2-virtual-machine:~/Desktop/info2$ size mem5
   text    data     bss     dec     hex filename
   1228     552       8    1788     6fc mem5
```

Actual



Área estática



Área Estática

Se guardan las variables y constantes globales, como así también las variables locales estáticas. **[Variables de duración estática]**

Pila (stack)

La **pila** del programa es una estructura **LIFO** (Last In, First Out), típicamente ubicada en las partes más altas de la memoria. En la arquitectura estándar de computadoras PC x86, crece hacia la dirección cero.

En la segunda parte de la materia vamos a ver en detalle este tipo de estructuras.

La **pila** es donde se almacenan las variables de **duración automática**, junto con la información que se guarda cada vez que se llama a una función.

Cuando se llama a una función se le asigna espacio en la **pila** para sus **variables** de **duración automáticas**.

- Spoiler: Así es como las funciones recursivas en C pueden funcionar. Cada vez que una función recursiva se llama a sí misma, se usa un nuevo stack frame, por lo que un conjunto de variables no interfiere con las variables de otra instancia de la función.

Pila (stack)

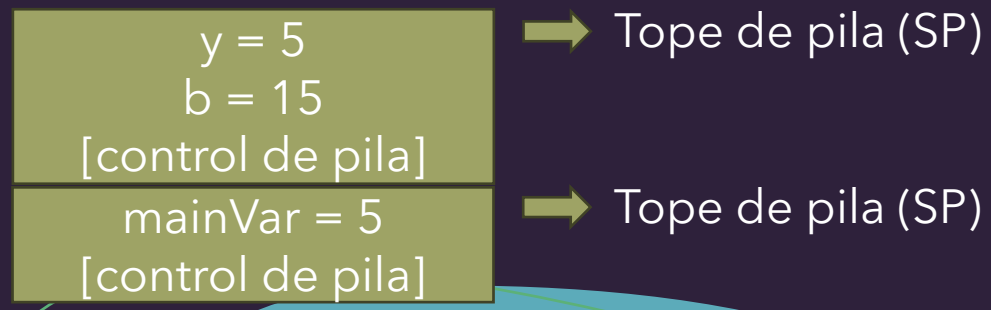
```c
#include <stdio.h>

// Prototipo de la función A
void functionA(int x);

void functionB(int y) {
    // Variable local b en el stack de functionB
    int b = y + 10;
    printf("In functionB: b = %d\n", b);
    // Llamada a functionA
    functionA(b);
}

void functionA(int x) {
    // Variable local a en el stack de functionA
    int a = x * 2;
    printf("In functionA: a = %d\n", a);
}

int main() {
    // Variable local mainVar en el stack de main
    int mainVar = 5;
    printf("In main: mainVar = %d\n", mainVar);
    functionB(mainVar);  // Llamada a functionB
    return 0;
}

```

mainVar = 5
[control de pila]

➡ Tope de pila (SP)

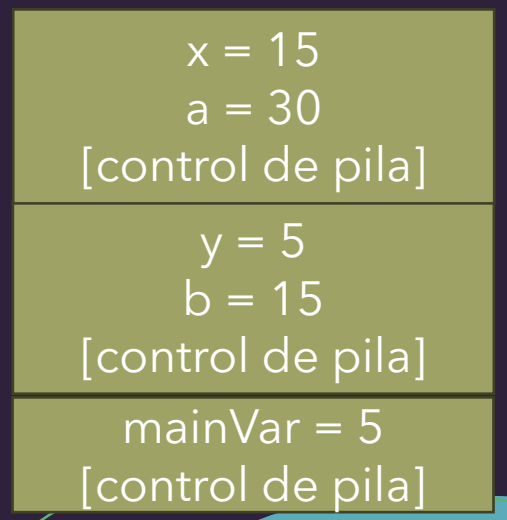
Pila (stack)

```c
#include <stdio.h>

// Prototipo de la función A
void functionA(int x);

void functionB(int y) {
    // Variable local b en el stack de functionB
    int b = y + 10;
    printf("In functionB: b = %d\n", b);
    // Llamada a functionA
    functionA(b);
}

void functionA(int x) {
    // Variable local a en el stack de functionA
    int a = x * 2;
    printf("In functionA: a = %d\n", a);
}

int main() {
    // Variable local mainVar en el stack de main
    int mainVar = 5;
    printf("In main: mainVar = %d\n", mainVar);
    functionB(mainVar);  // Llamada a functionB
    return 0;
}

```



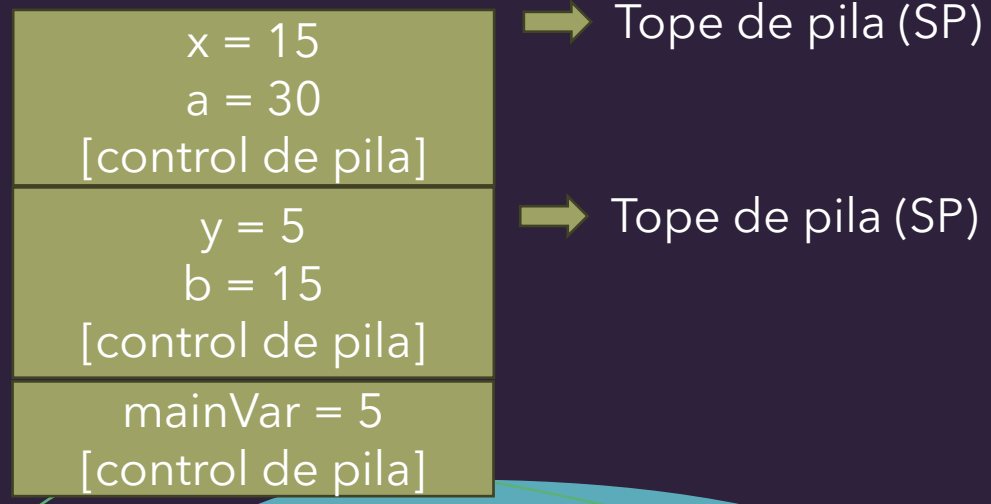
Pila (stack)

```c
#include <stdio.h>

// Prototipo de la función A
void functionA(int x);

void functionB(int y) {
    // Variable local b en el stack de functionB
    int b = y + 10;
    printf("In functionB: b = %d\n", b);
    // Llamada a functionA
    functionA(b);
}

void functionA(int x) {
    // Variable local a en el stack de functionA
    int a = x * 2;
    printf("In functionA: a = %d\n", a);
}

int main() {
    // Variable local mainVar en el stack de main
    int mainVar = 5;
    printf("In main: mainVar = %d\n", mainVar);
    functionB(mainVar);  // Llamada a functionB
    return 0;
}
```



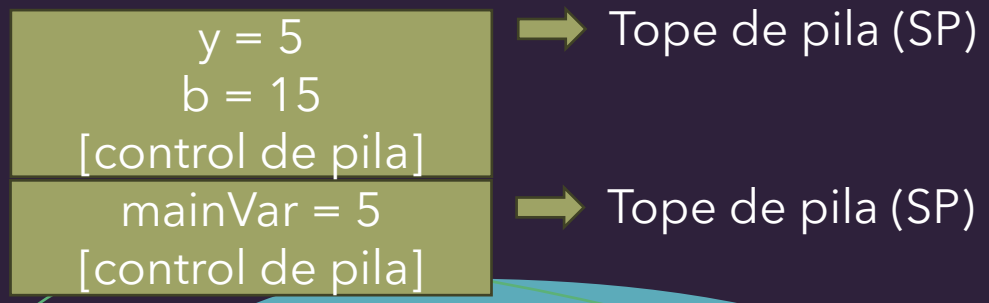
Pila (stack)

```c
#include <stdio.h>

// Prototipo de la función A
void functionA(int x);

void functionB(int y) {
    // Variable local b en el stack de functionB
    int b = y + 10;
    printf("In functionB: b = %d\n", b);
    // Llamada a functionA
    functionA(b);
}

void functionA(int x) {
    // Variable local a en el stack de functionA
    int a = x * 2;
    printf("In functionA: a = %d\n", a);
}

int main() {
    // Variable local mainVar en el stack de main
    int mainVar = 5;
    printf("In main: mainVar = %d\n", mainVar);
    functionB(mainVar);  // Llamada a functionB
    return 0;
}
```



Pila (stack)

```c
#include <stdio.h>

// Prototipo de la función A
void functionA(int x);

void functionB(int y) {
    // Variable local b en el stack de functionB
    int b = y + 10;
    printf("In functionB: b = %d\n", b);
    // Llamada a functionA
    functionA(b);
}

void functionA(int x) {
    // Variable local a en el stack de functionA
    int a = x * 2;
    printf("In functionA: a = %d\n", a);
}

int main() {
    // Variable local mainVar en el stack de main
    int mainVar = 5;
    printf("In main: mainVar = %d\n", mainVar);
    functionB(mainVar);  // Llamada a functionB
    return 0;
}

```



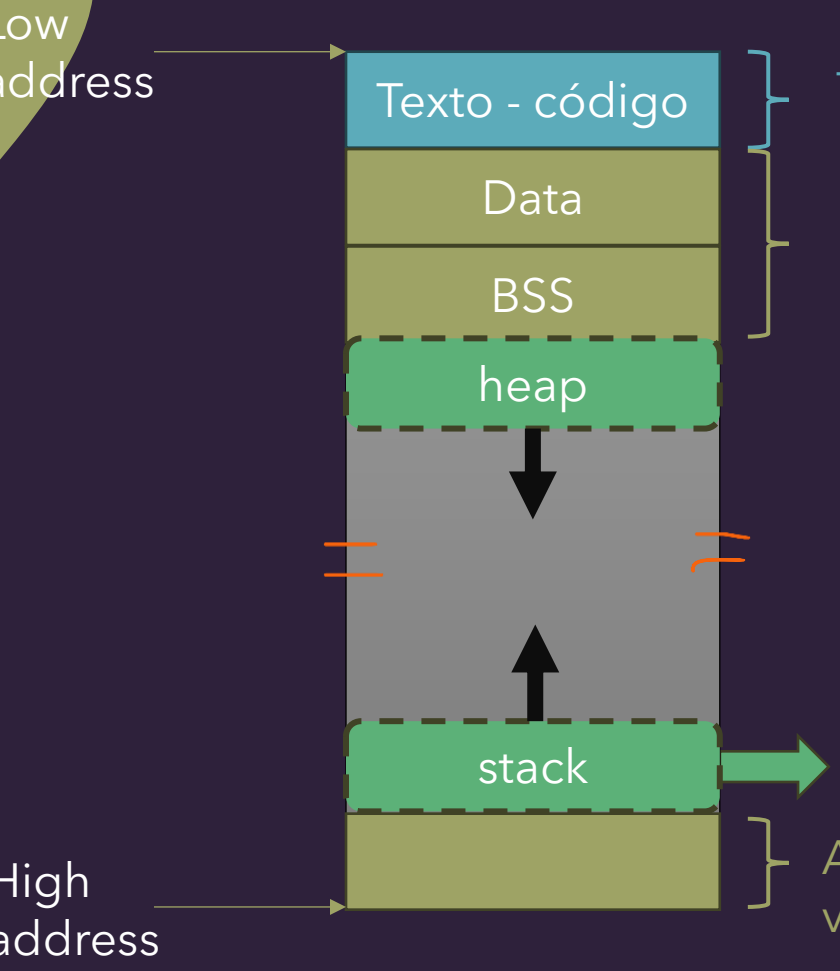
Pila (stack)

```c
#include <stdio.h>

// Prototipo de la función A
void functionA(int x);

void functionB(int y) {
    // Variable local b en el stack de functionB
    int b = y + 10;
    printf("In functionB: b = %d\n", b);
    // Llamada a functionA
    functionA(b);
}

void functionA(int x) {
    // Variable local a en el stack de functionA
    int a = x * 2;
    printf("In functionA: a = %d\n", a);
}

int main() {
    // Variable local mainVar en el stack de main
    int mainVar = 5;
    printf("In main: mainVar = %d\n", mainVar);
    functionB(mainVar);  // Llamada a functionB
    return 0;
}
```

mainVar = 5
[control de pila]

➡ Tope de pila (SP)

Distribución de memoria de los programas en C



Text segment

Data segment

Área estática:
Se guardan las variables y constantes globales, como así también las variables locales estáticas. [Variables de duración estática]

Stack:
Se guardan parámetros de las funciones y variables locales [variables de duración automática]

Argumentos de línea de comandos y variables de entorno.

Distribución de memoria de los programas en C

- Tanto la pila como el área de memoria estática tienen dos cosas en común:

1. El tamaño de la variable (vector, estructura, etc) debe conocerse en el momento de la compilación.
2. La asignación de la memoria, tanto así como su "destrucción" ocurren automáticamente.

No podemos "meter la pata" en la asignación de memoria. **Por ahora....**

- En C existe otra forma de asignar memoria, que es la asignación dinámica.

Es útil cuando no sabemos en tiempo de compilación la cantidad de memoria que vamos a necesitar. Ejemplo:

Distribución de memoria de los programas en C

Cree un programa que permita ingresar el nombre de un grupo de alumnos y sus dos notas.

Al momento de escribir el código no sabemos cuantos alumnos son. Hasta ahora nuestra solución fue imaginar una cantidad, y usar por ejemplo:

```
#define N 100
```

Esta es una mala solución. Quizás eran solo 2 alumnos y estamos siendo pocos eficientes, o peor aun nos quedamos cortos y eran mas de 100 alumnos.

Este es un caso de uso donde usar la memoria dinámica es la mejor opción.

Otro problema conocido:

```c
#include <stdio.h>

int main()
{
    int vec[1000000];
    printf("Hello World!\n");
    getchar();
    return 0;
}
```

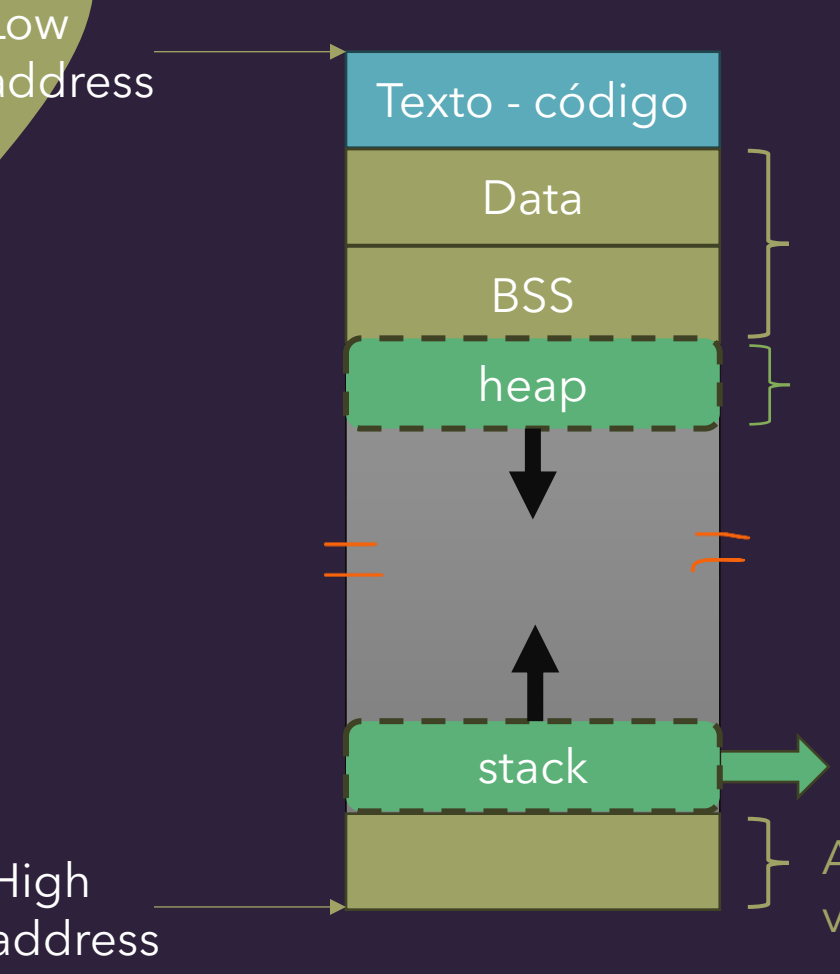
```
qtc.process_stub: Inferior error:  QProcess::Crashed "Process crashed"
```

Se produce un desborde de pila

Stack Overflow

El **stack** es una zona de memoria limitada, cuyo tamaño depende tanto del compilador como del **sistema operativo**, que impone restricciones por seguridad y estabilidad. Podemos hacernos una imagen mental y aproximarlo a unos **4 MB**.
El **heap** (también llamado "la parva" o "el montón") está gestionado completamente por el sistema operativo, y en nuestras computadoras modernas puede alcanzar **varios GB**.

Distribución de memoria de los programas en C



Área estática: Se guardan las variables y constantes globales, como así también las variables locales estáticas. [Variables de duración estática]

Área dinámica: Se gestiona en tiempo de ejecución con las funciones:

- malloc ()
- calloc ()
- realloc ()
- free ()

<stdlib.h>

Stack:
Se guardan parámetros de las funciones y variables locales [variables de duración automática]

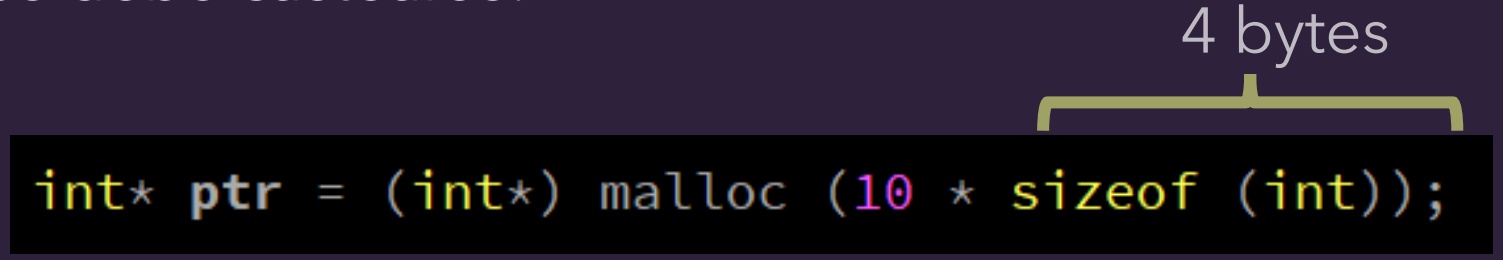
Argumentos de línea de comandos y variables de entorno.

“Memory allocation” -> malloc ()

Se utiliza para asignar dinámicamente un único bloque grande de memoria con el tamaño especificado. Su prototipo es:

void* malloc (size_t tamaño)

Devuelve un puntero de tipo void que puede convertirse en un puntero de cualquier forma. Por ese debe castearse.



malloc devuelve la dirección de memoria de inicio del bloque de 40 bytes en caso de que todo haya salido bien.

Si el espacio es insuficiente, la asignación de memoria falla y malloc retorna el puntero NULL.

No se inicializa la memoria. Por lo que el bloque tienen basura inicialmente.

“Memory allocation” -> malloc ()

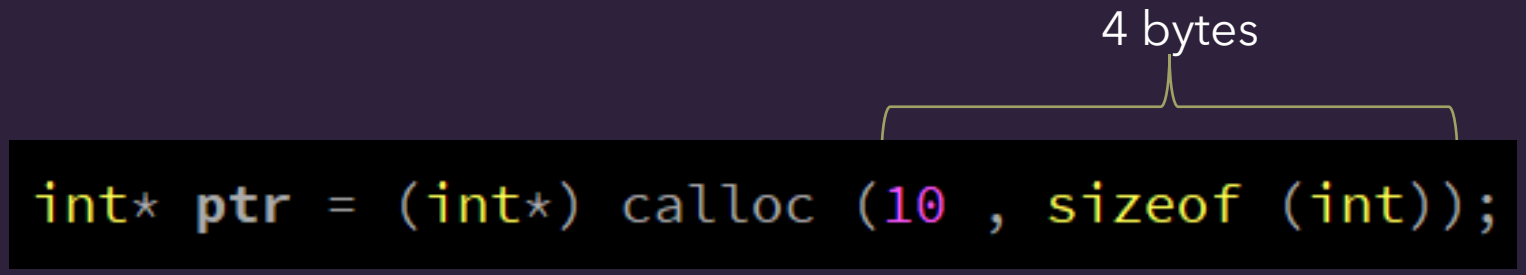
Ir a ejemplo en QT - malloc

“Contiguous allocation” -> calloc ()

Se utiliza para asignar dinámicamente el número especificado de bloques de memoria del tipo especificado.
Es muy similar a malloc(), pero tiene dos diferencias y estas son:

1. Inicializa cada bloque con un valor predeterminado ‘0’.
2. Tiene dos argumentos.

Su prototipo es: **void* calloc (size_t numero, size_t tamaño)**



calloc devuelve la dirección de memoria de inicio del bloque de 40 bytes en caso de que todo haya salido bien.

Si el espacio es insuficiente, la asignación de memoria falla y calloc retorna el puntero NULL.

“Contiguous allocation” -> calloc ()

Ir a ejemplo en QT - calloc

free()

La función free() libera un bloque de memoria gestionado anteriormente, a fin de poder asignarlo nuevamente. Su formato es el siguiente:

void free (void*)

El argumento entregado a la función free() debe provenir de una asignación dinámica anterior.

free()

Ir al ejemplo de QT-> free

Uso de Free

Con ***free*** vamos a liberar memoria que previamente hayamos asignado con ***malloc*** y **volverá a estar disponible por el SO.**

No liberar memoria cuando ya no se utiliza mas es un error común y se conoce como **memory leak.**

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *ptr = (int*) malloc (5* sizeof(int));
    *ptr = 14;
    printf("ptr: %p \t *ptr: %d \n", ptr, *ptr);
    free (ptr);
    printf("\nVamos otra vez:\n");
    printf("ptr: %p \t *ptr: %d \n", ptr, *ptr);
    getchar();
    return 0;

}
```

https://www.youtube.com/watch?v=5VnDaHBi8dM

¿Qué borra **free**?

nada

```
ptr: 00000250632514c0    *ptr: 14

Vamos otra vez:
ptr: 00000250632514c0    *ptr: 1663392768
```

Puntero colgante, Dangling pointer o Puntero loco

Uso de free + NULL

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *ptr = (int*) malloc (5* sizeof(int));
    *ptr = 14;
    printf("ptr: %p \t *ptr: %d \n", ptr, *ptr);
    free (ptr);
    ptr = NULL ;
    printf("\nVamos otra vez:\n");
    printf("ptr: %p \t *ptr: %d \n", ptr, *ptr);
    getchar();
    return 0;

}
```

```
ptr: 00000239005a14c0    *ptr: 14

Vamos otra vez:
qtc.process_stub: Inferior error:  QProcess::Crashed "Process crashed"

Process exited with code: -1073741819
```

Puntero Nulo

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *ptr = NULL ;

    if (!ptr)
    {
        ptr = (int*) malloc (5* sizeof(int));
        *ptr = 14;
    }

    printf("ptr: %p \t *ptr: %d \n", ptr, *ptr);
    free (ptr);
    ptr = NULL ;
    getchar();
    return 0;

}
```

Inicializar los punteros con **NULL** puede ser una buena práctica. Evitamos tener punteros colgantes y además podes utilizarlos condicionalmente como vemos en el ejemplo.

Memory leaks [perdidas de memoria]

```c
#include <stdlib.h>

void asignar (int) ;

int main()
{
    int x;
    printf ("Ingrese un valor: ");
    scanf ("%d", &x);
    asignar (x);
    return 0;

}

void asignar (int aX)
{
    int *ptr = (int*) malloc (aX* sizeof(int));
    *ptr = 14;
    printf("ptr: %p \t *ptr: %d \n", ptr, *ptr);
    getchar();
```

El puntero **ptr** en la función **asignar** tiene duración automática.
El programa al llegar a la llave de cierre de la función, **ptr** deja de existir.
En ese momento perdimos el acceso a la memoria que le pedimos prestada al SO.
Pero el SO sigue reservando esa memoria para nosotros.

"NO PODEMOS SIQUIERA USAR FREE"

Memory leaks [perdidas de memoria]

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int x = 10;
    int *ptr = (int*) malloc (4* sizeof(int));
    ptr = &x;
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int x = 10;
    int *ptr = (int*) malloc (4* sizeof(int));
    ptr = (int*) malloc (5* sizeof(int));
    return 0;
}
```

Hay varias formas de meter la pata. Prestar atención no siempre son solo 3 líneas de código.

“re-allocation” -> realloc()

Se utiliza para cambiar la asignación de memoria dinámica de un bloque previamente asignado.

En otras palabras, si la memoria previamente asignada con la ayuda de malloc o calloc es insuficiente, realloc puede usarse para re-asignar dinámicamente la memoria.

La re-asignación de memoria mantiene el valor ya presente y los nuevos bloques se inicializarán con el valor de basura predeterminado.

void* realloc (void* , unsigned int tamaño)

```
int* ptr = (int*) malloc (5 * sizeof (int));
ptr = realloc (ptr, 10*sizeof(int));
```

En caso de no poder reasignar el bloque, realloc() retorna el puntero NULL.

“re-allocation” -> realloc()

Ir al ejemplo de QT-> realloc

Errores típicos

1. Fugas de Memoria (Memory Leaks):

- **Descripción: Ocurre cuando la memoria asignada dinámicamente no se libera con free antes de que se pierdan todas las referencias a ella.**
- **Consecuencia: El programa consume más memoria de la necesaria, lo que puede llevar a un agotamiento de la memoria disponible y, eventualmente, a la caída del programa.**

2. Asignación Incorrecta de Tamaño:

- **Descripción**: Se asigna una cantidad de memoria incorrecta, ya sea menos de la necesaria o más de lo necesario.
- **Consecuencia**: Asignar menos memoria de la necesaria puede llevar a desbordamientos de búfer, mientras que asignar más de la necesaria puede resultar en un uso ineficiente de la memoria.

3. No Comprobación de Éxito de Asignación:

- **Descripción**: No se comprueba si **malloc** o **calloc** devuelve NULL, lo que indica que la asignación de memoria ha fallado.
- **Consecuencia**: Intentar usar un puntero NULL puede llevar a fallos del programa y a comportamiento indefinido.

4. Fragmentación de Memoria:

- **Descripción**: La memoria se divide en muchos pequeños bloques libres y ocupados, lo que puede dificultar futuras asignaciones de grandes bloques de memoria.
- **Consecuencia**: Reducción de la eficiencia de la memoria, potencialmente llevando a fallos de asignación incluso si hay suficiente memoria total disponible.

Errores típicos

1. **Fugas de Memoria (Memory Leaks):**
   - Descripción: Ocurren cuando la memoria asignada dinámicamente no se libera con **free** antes de que se pierdan todas las referencias a ella.
   - Consecuencia: El programa sigue consumiendo memoria innecesariamente, lo que puede agotar la memoria disponible y llevar a una caída del sistema o una degradación del rendimiento.
2. **Asignación Incorrecta de Tamaño:**
   - Descripción: Se reserva una cantidad incorrecta de memoria, ya sea menor o mayor a la necesaria.
   - Consecuencia:
     - Asignar menos memoria puede causar desbordamientos de búfer (vulnerabilidad crítica).
     - Asignar más memoria de la necesaria genera desperdicio y contribuye a la fragmentación del heap.
3. **No Comprobación del Éxito en la Asignación:**
   - Descripción: No se verifica si malloc, calloc o realloc devuelven NULL, lo cual indica error en la asignación.
   - Consecuencia: Usar un puntero NULL como si fuera válido provoca fallos del programa o comportamiento indefinido.
4. **Fragmentación de Memoria:**
   - Descripción: La memoria dinámica se divide en muchos pequeños bloques libres y ocupados, lo que dificulta la asignación de bloques grandes.
   - Consecuencia: Se reduce la eficiencia del uso de memoria, y el programa puede fallar al solicitar grandes bloques, incluso si la memoria total disponible parece suficiente.

Errores típicos

5. **Desbordamiento de Búfer (Buffer Overflow) [tema scanf]:**
   - Descripción: Se escribe más allá del límite de un arreglo, sobrescribiendo memoria contigua.
   - Consecuencia: Puede alterar otras variables, provocar caídas del programa o permitir ejecución de código malicioso.
6. **Desbordamiento de Heap (Heap Overflow):**
   - Descripción: Ocurre cuando se escribe más allá del límite de un bloque de memoria dinámica (asignado con malloc, calloc, etc.).
   - Consecuencia: Puede corromper otras estructuras del heap, afectando el comportamiento del programa o introduciendo vulnerabilidades de seguridad.
7. **Uso Después de Liberar (Use-After-Free):**
   - Descripción: Se accede a un bloque de memoria después de haber sido liberado con free.
   - Consecuencia: El programa puede fallar, comportarse de forma impredecible o incluso permitir que se ejecute código no deseado.
8. **Doble Liberación (Double Free):**
   - Descripción: Se llama free más de una vez sobre el mismo puntero.
   - Consecuencia: Puede causar corrupción del heap, errores de seguridad o abortos inmediatos del programa.

Errores típicos

9. **Corrupción de Punteros:**
   - Descripción: Un puntero es sobrescrito accidentalmente, por ejemplo, debido a un desbordamiento o mal manejo de memoria.
   - Consecuencia: Puede alterar la lógica del programa, provocar accesos inválidos o generar vulnerabilidades difíciles de detectar.
10. **No Anular el Puntero Después del Free:**
    - Descripción: Se deja un puntero apuntando a memoria que ya fue liberada.
    - Consecuencia: Aumenta el riesgo de uso posterior indebido (use-after-free), con consecuencias impredecibles.

En C, la gestión manual de la memoria dinámica puede llevar a errores de programación que se traducen, o pueden traducirse, en vulnerabilidades de seguridad.
Estos problemas surgen cuando no se controla correctamente el acceso a la memoria, permitiendo corromper datos o incluso ejecutar código malicioso.

Ejercitación

1. Supongamos que a lo largo de la ejecución de un programa en C, cada vez que se invoca a malloc o calloc se incrementase una variable global entera a modo de contador inicializada a cero, y cada vez que se invocase free se decrementase: ¿Qué valor debería tener justo antes de terminar la ejecución? Y si un programa termina su ejecución y esta variable vale cero, ¿es esto suficiente para concluir que hace una gestión de memoria correcta?

2. Ingresar por teclado un entero que represente la cantidad de elementos que debe crearse un vector. Crear el vector en forma dinámica, cargar e imprimir sus datos. Hacer todo en main.

3. Se tiene la siguiente declaración de registro:

```
struct producto {
    int codigo;
    char descripcion[41];
    float precio; };
```

Definir un puntero de tipo producto y luego mediante la función malloc crear un registro en la pila dinámica. Cargar el registro, imprimirlo y finalmente liberar el espacio reservado mediante la función free.

4. Pedir ingresar por teclado cuantas letras tiene una palabra. Seguidamente crear un vector en forma dinámica que reserve el espacio mínimo para ingresar dicha palabra.
   Cargar por teclado la palabra, mostrarla y finalmente liberar el espacio requerido.

FIN