Curso de programación en C moderno

(II Edición)

Neira Ayuso, Pablo Falgueras García, Carlos

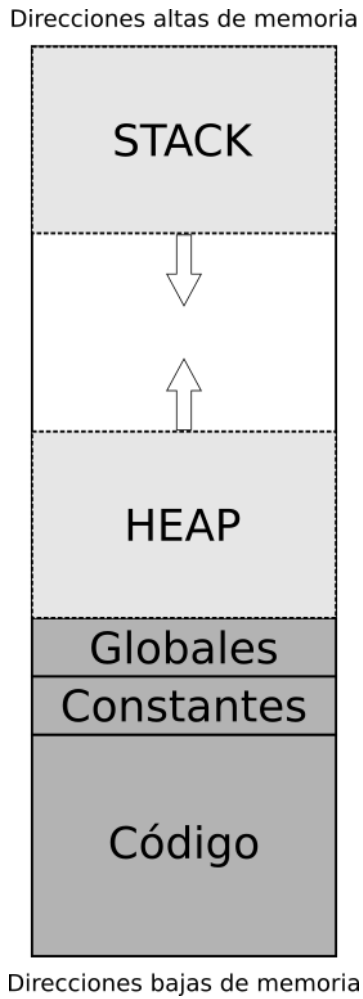
Tema 6

Reserva dinámica de memoria

Índice

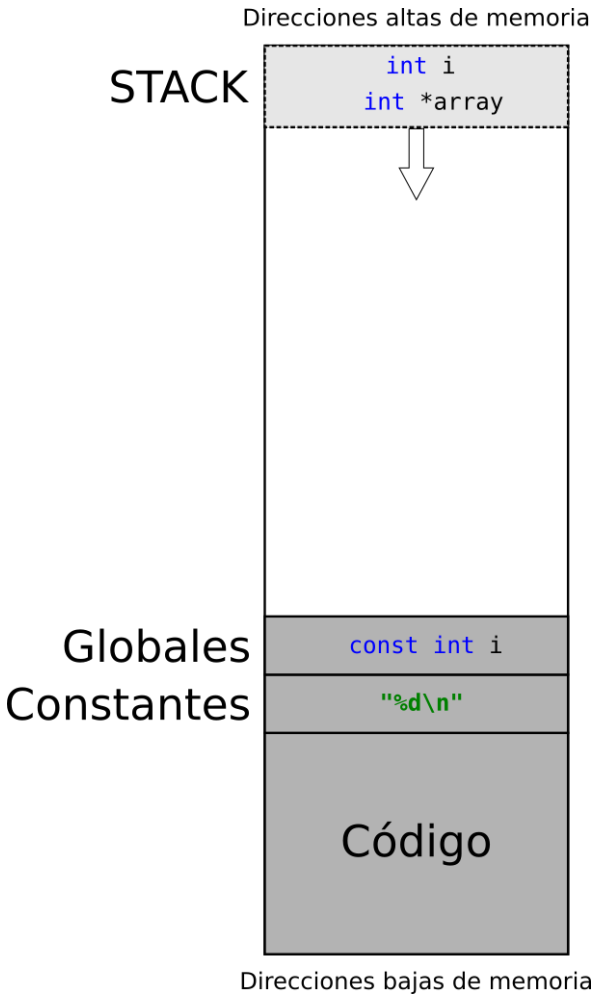
Mapa de memoria

- **Heap**: Se almacena la memoria reservada **dinámicamente** con **malloc**
- **Stack**: Se almacenan las **variables locales** de cada llamada a función
- **Globales**: Todas las variables globales
- **Constantes**: Todas las constantes (números, cadenas, etc)
- **Código**: El programa en sí



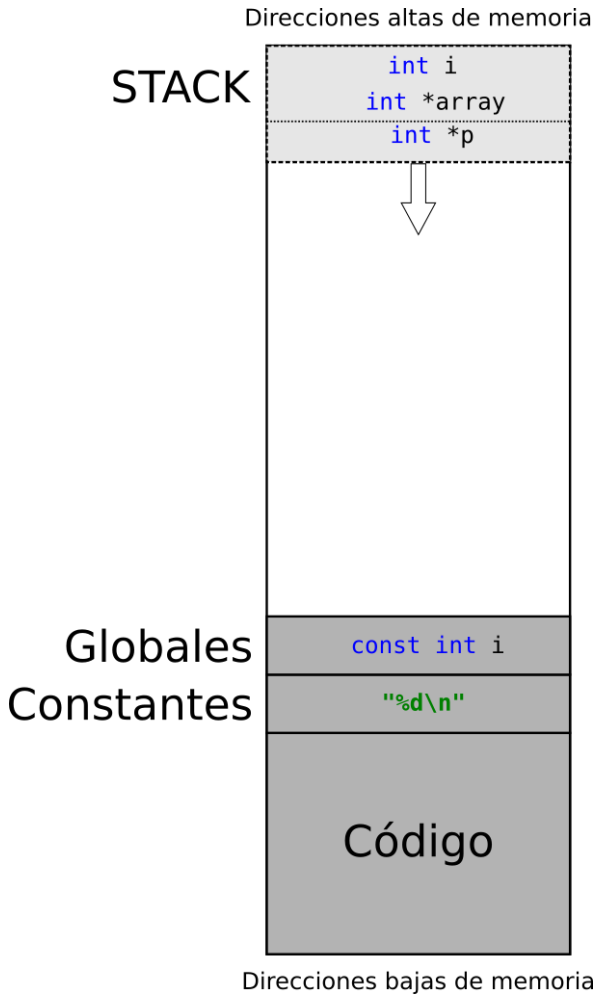
Ejemplo


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

int *array_alloc(int n)
{
  int *p = (int *)malloc(n * sizeof(int))
  return p;
}

static const int tam = 3;

int main()
{
  int i;
  int *array;

  array = array_alloc(tam);

  /* Inicializamos */
  for (i = 0; i < tam; i++)
    array[i] = i;

  /* Imprimimos */
  for (i = 0; i < tam; i++)
    printf("%d\n", array[i]);

  free(array);

  return 0;
}
```



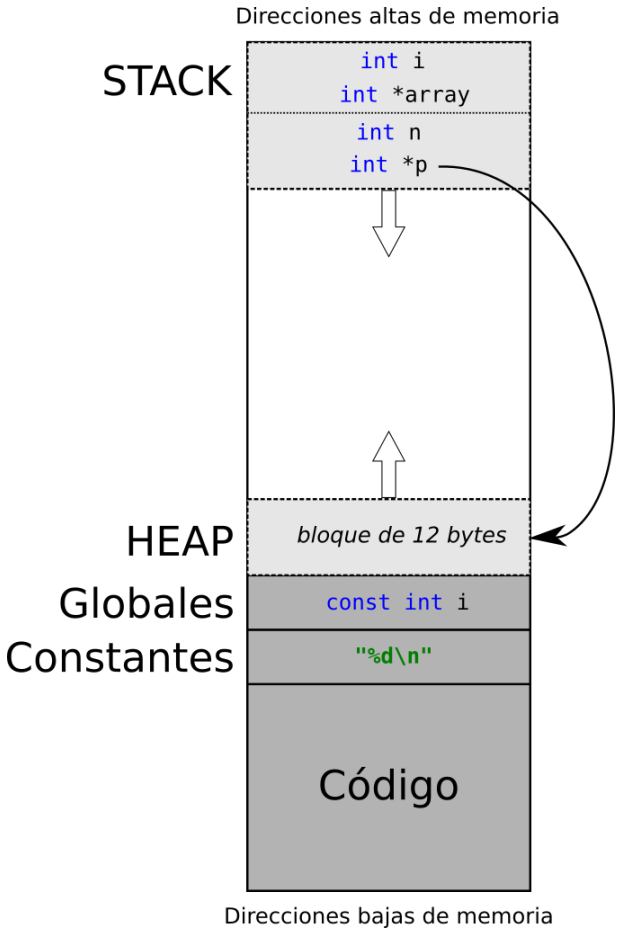
Ejemplo


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

int *array_alloc(int n)
{
  int *p = (int *)malloc(n * sizeof(int))
  return p;
}

static const int tam = 3;

int main()
{
  int i;
  int *array;

  array = array_alloc(tam);

  /* Inicializamos */
  for (i = 0; i < tam; i++)
    array[i] = i;

  /* Imprimimos */
  for (i = 0; i < tam; i++)
    printf("%d\n", array[i]);

  free(array);

  return 0;
}
```



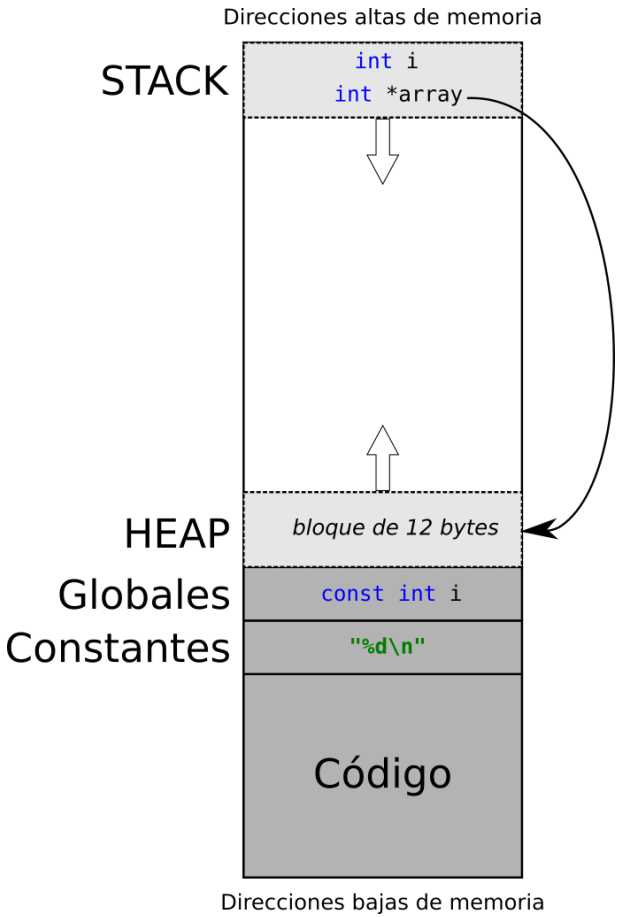
Ejemplo


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

int *array_alloc(int n)
{
  int *p = (int *)malloc(n * sizeof(int))
  return p;
}

static const int tam = 3;

int main()
{
  int i;
  int *array;

  array = array_alloc(tam);

  /* Inicializamos */
  for (i = 0; i < tam; i++)
    array[i] = i;

  /* Imprimimos */
  for (i = 0; i < tam; i++)
    printf("%d\n", array[i]);

  free(array);

  return 0;
}
```



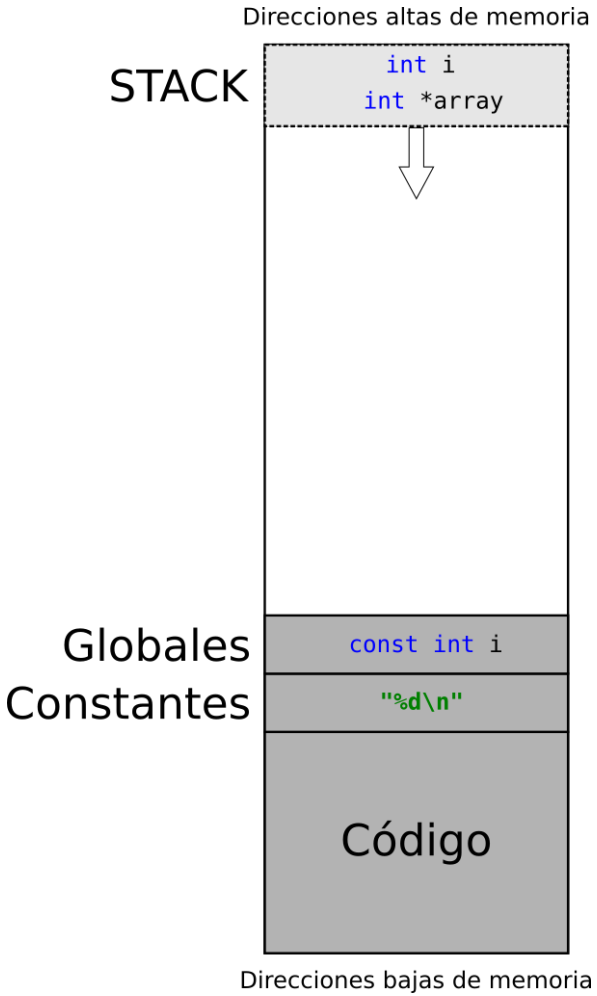
Ejemplo


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

int *array_alloc(int n)
{
  int *p = (int *)malloc(n * sizeof(int))
  return p;
}

static const int tam = 3;

int main()
{
  int i;
  int *array;

  array = array_alloc(tam);

  /* Inicializamos */
  for (i = 0; i < tam; i++)
    array[i] = i;

  /* Imprimimos */
  for (i = 0; i < tam; i++)
    printf("%d\n", array[i]);

  free(array);

  return 0;
}
```



Ejemplo

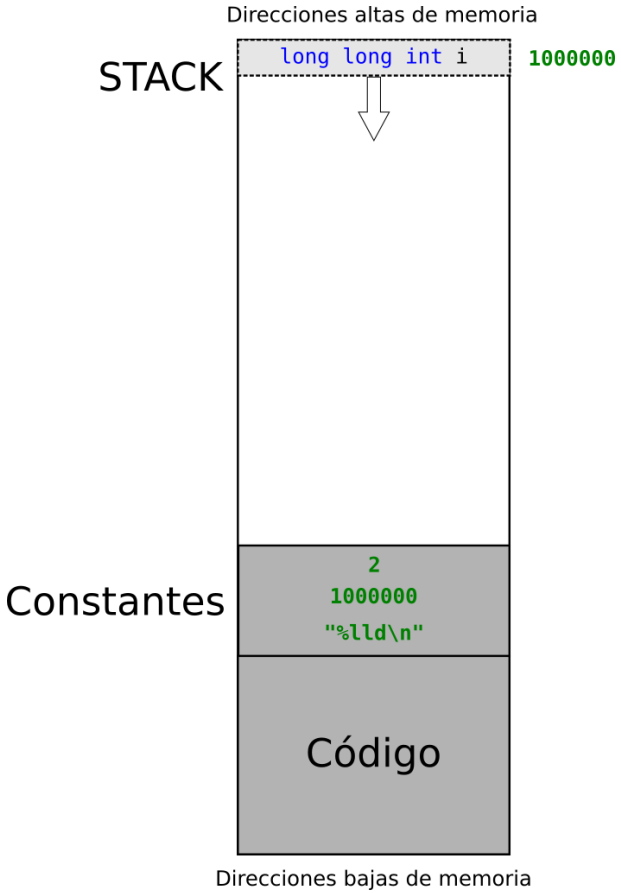

```c
#include <stdio.h>
#include <stdlib.h>

int *array_alloc(int n)
{
  int *p = (int *)malloc(n * sizeof(int))
  return p;
}

static const int tam = 3;

int main()
{
  int i;
  int *array;

  array = array_alloc(tam);

  /* Inicializamos */
  for (i = 0; i < tam; i++)
    array[i] = i;

  /* Imprimimos */
  for (i = 0; i < tam; i++)
    printf("%d\n", array[i]);

  free(array);

  return 0;
}
```



Stack Overflow

El uso de funciones recursivas puede ocasionar un agotamiento de la pila

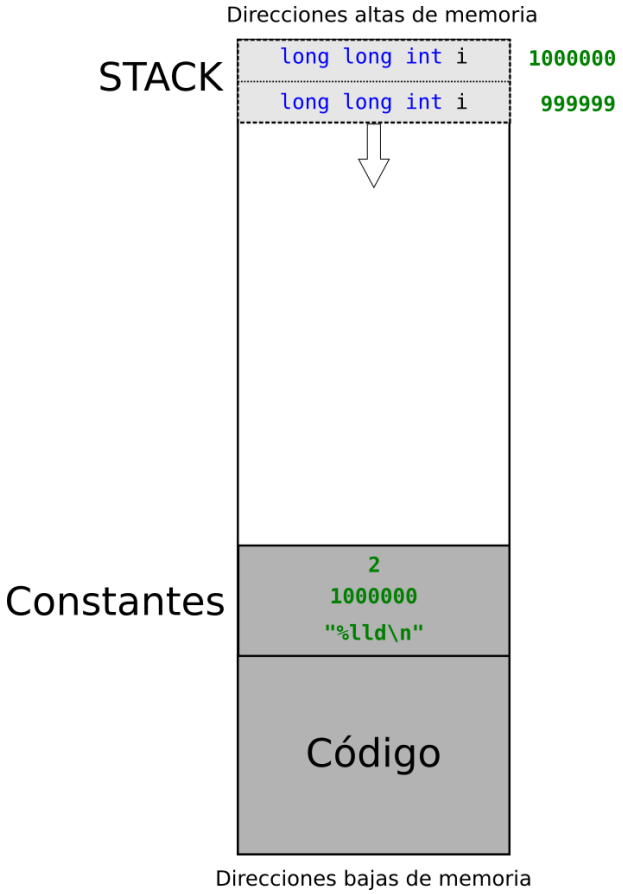
```c
#include <stdio.h>

long long int sum_all(long long int i)
{
  if (i >= 2)
    return i + sum_all(i - 1);
  else
    return i;
}

int main()
{
  printf("%lld\n", sum_all(1000000));

  return 0;
}
```



Stack Overflow

El uso de funciones recursivas puede ocasionar un agotamiento de la pila

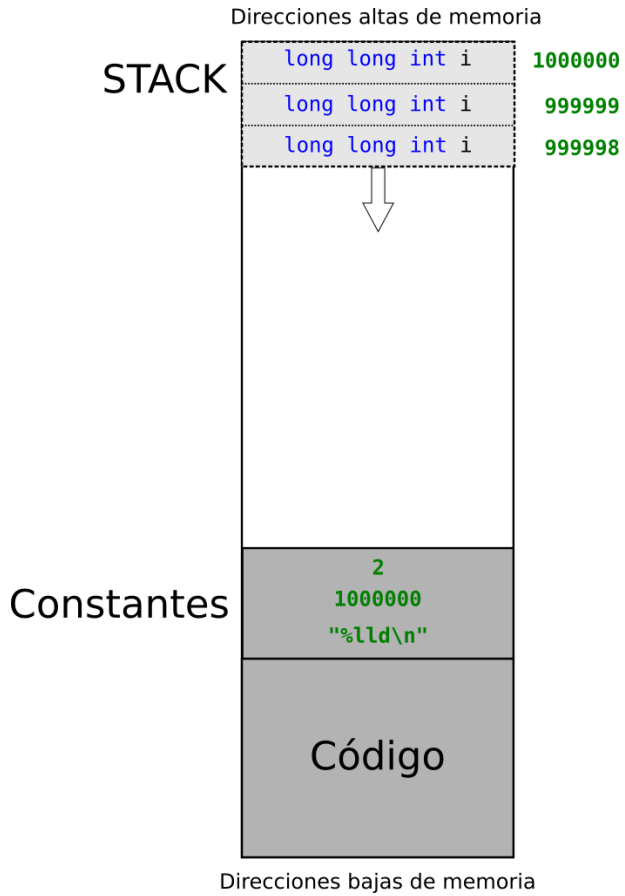
```
1  #include <stdio.h>
2
3  long long int sum_all(long long int i)
4  {
5    if (i >= 2)
6      return i + sum_all(i - 1);
7    else
8      return i;
9  }
10
11 int main()
12 {
13   printf("%lld\n", sum_all(1000000));
14
15   return 0;
16 }
```



Stack Overflow

El uso de funciones recursivas puede ocasionar un agotamiento de la pila

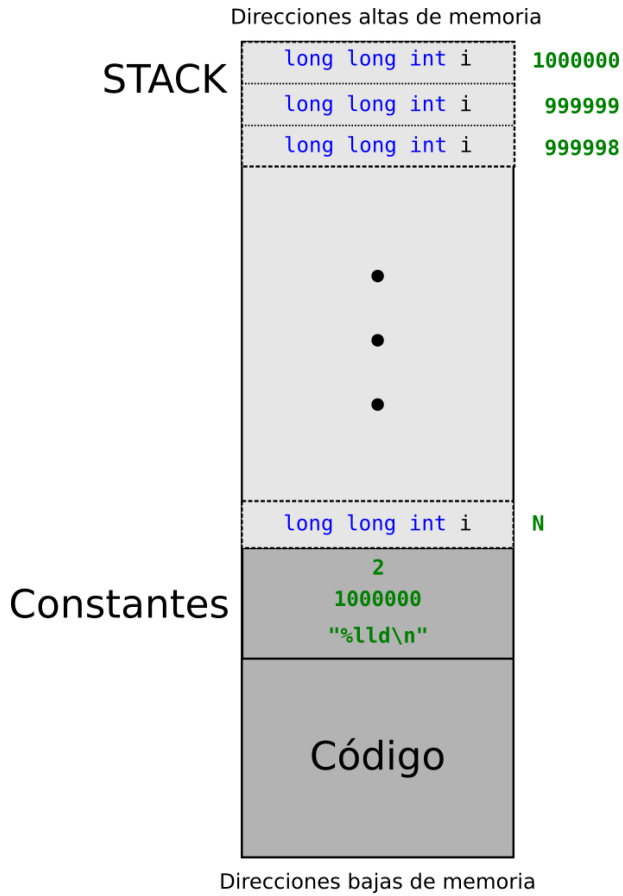
```
1  #include <stdio.h>
2
3  long long int sum_all(long long int i)
4  {
5    if (i >= 2)
6      return i + sum_all(i - 1);
7    else
8      return i;
9  }
10
11 int main()
12 {
13   printf("%lld\n", sum_all(1000000));
14
15   return 0;
16 }
```



Stack Overflow

El uso de funciones recursivas puede ocasionar un agotamiento de la pila

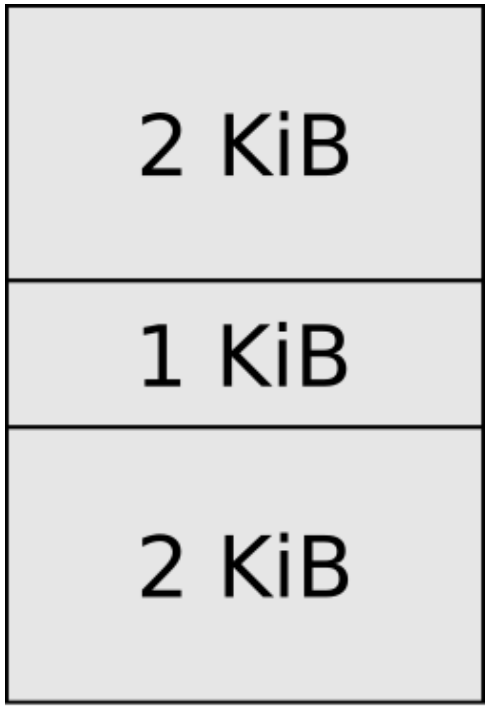
```
1  #include <stdio.h>
2
3  long long int sum_all(long long int i)
4  {
5    if (i >= 2)
6      return i + sum_all(i - 1);
7    else
8      return i;
9  }
10
11 int main()
12 {
13   printf("%lld\n", sum_all(1000000));
14
15   return 0;
16 }
```



Fragmentación

Fragmentos pequeños de memoria libre entre bloques de memoria reservada.

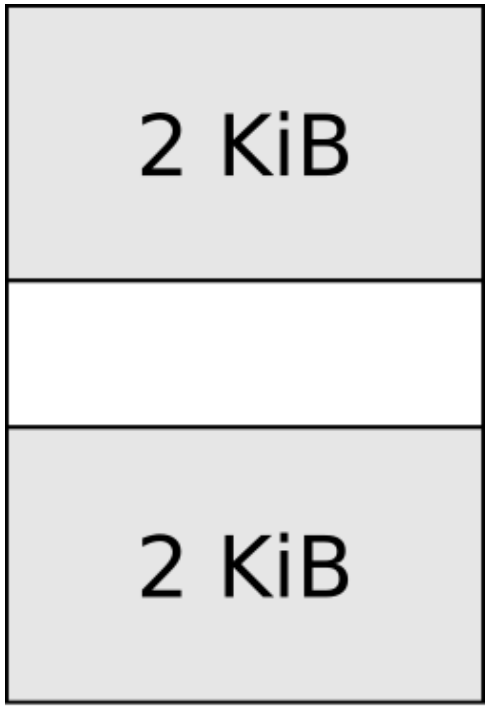
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   void *p1 = malloc(2 * 1024);
   void *p2 = malloc(1 * 1024);
   void *p3 = malloc(2 * 1024);

   void *p;

   printf("%ld\n", (long int)p1 / 1024);
   printf("%ld\n", (long int)p2 / 1024);
   printf("%ld\n", (long int)p3 / 1024);

   free(p2);
   p = malloc(2 * 1024);

   printf("\n%ld\n", (long int)p / 1024);

   return 0;
}
```



Fragmentación

Fragmentos pequeños de memoria libre entre bloques de memoria reservada.

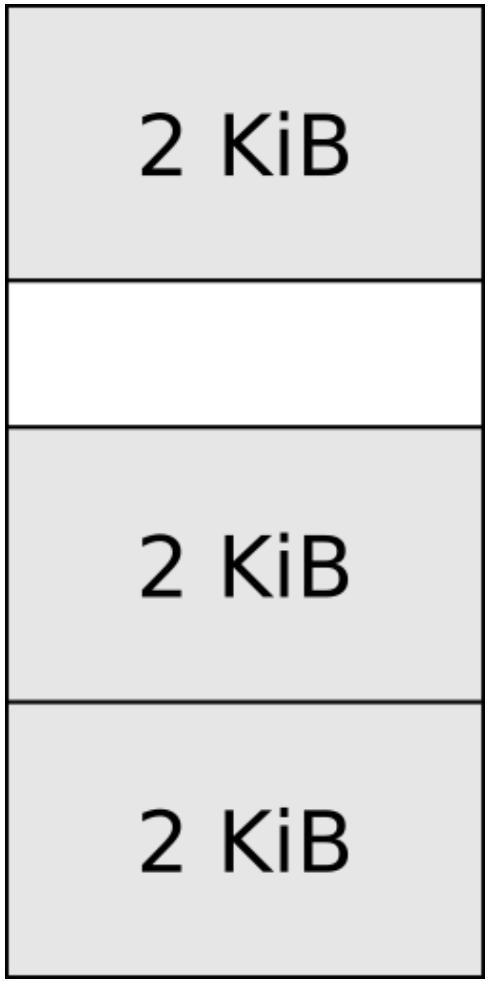
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   void *p1 = malloc(2 * 1024);
   void *p2 = malloc(1 * 1024);
   void *p3 = malloc(2 * 1024);

   void *p;

   printf("%ld\n", (long int)p1 / 1024);
   printf("%ld\n", (long int)p2 / 1024);
   printf("%ld\n", (long int)p3 / 1024);

   free(p2);
   p = malloc(2 * 1024);

   printf("\n%ld\n", (long int)p / 1024);

   return 0;
}
```



Fragmentación

Fragmentos pequeños de memoria libre entre bloques de memoria reservada.

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

int main()
{
   void *p1 = malloc(2 * 1024);
   void *p2 = malloc(1 * 1024);
   void *p3 = malloc(2 * 1024);

   void *p;

   printf("%ld\n", (long int)p1 / 1024);
   printf("%ld\n", (long int)p2 / 1024);
   printf("%ld\n", (long int)p3 / 1024);

   free(p2);
   p = malloc(2 * 1024);

   printf("\n%ld\n", (long int)p / 1024);

   return 0;
}
```



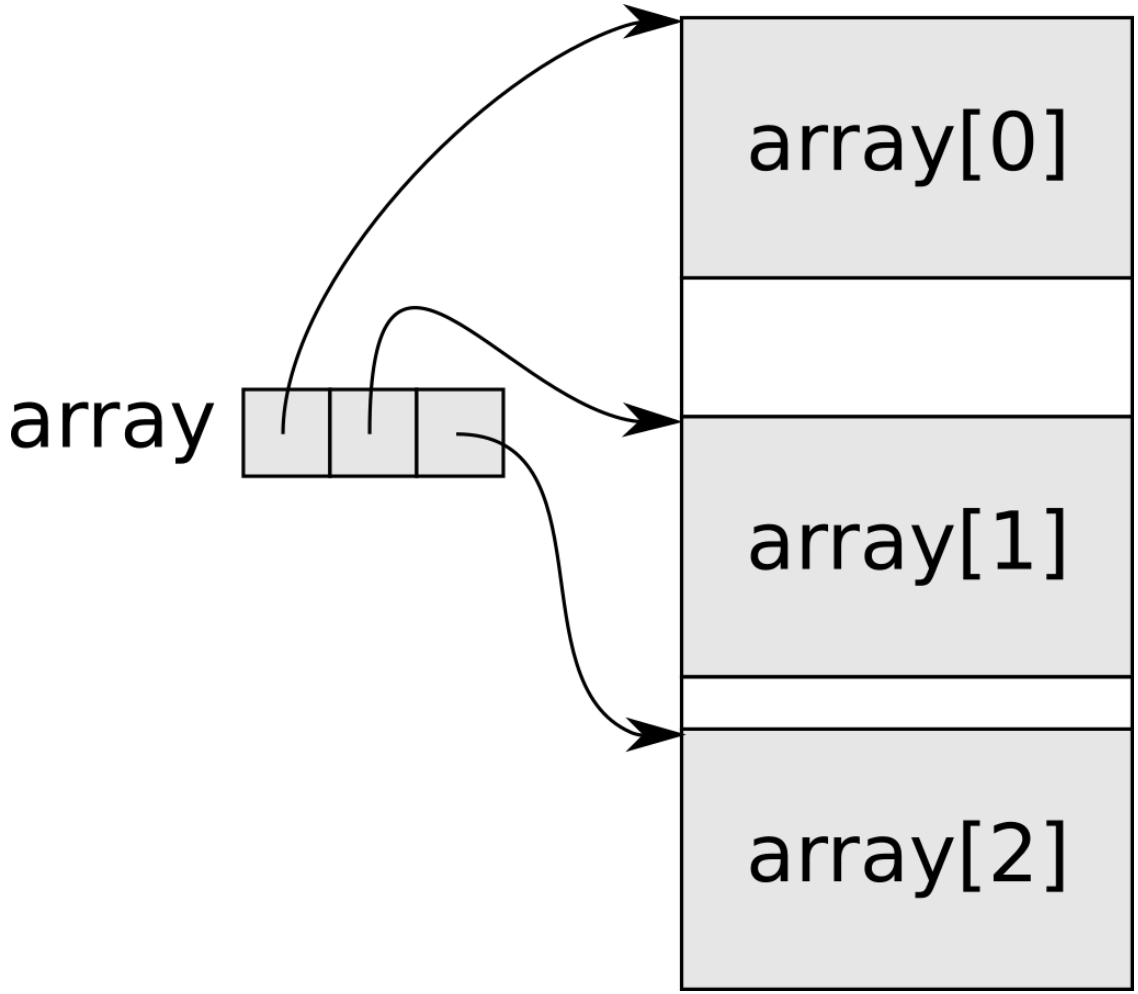
Forma 1 (la mala)

Un malloc por cada fila del array (array de punteros)

```
1  int main()
2  {
3    int i, j;
4    int **array;
5
6    /* Reservamos */
7    array = (int **)malloc(ROWS * sizeof(int *));
8    for (i = 0; i < ROWS; i++)
9      array[i] = (int *)malloc(COLS * sizeof(int)); // Falta comprobar
10
11   /* Inicializamos */
12   for (i = 0; i < ROWS; i++)
13     for (j = 0; j < ROWS; j++)
14       array[i][j] = i * 10 + j;
15
16   /* Imprimimos */
17   for (i = 0; i < ROWS; i++) {
18     for (j = 0; j < ROWS; j++) {
19       printf("%02d ", array[i][j]);
20     }
21     printf("\n");
22   }
23
24   /* Liberamos */
25   for (i = 0; i < ROWS; i++)
26     free(array[i]);
27
28   return 0;
29 }
```

Forma 1 (la mala)

No se garantiza continuidad entre los bloques reservados



Forma 2 (la buena)


```c
#define ROWS 3
#define COLS 3

int main()
{
  int i, j;
  int *array;

  /* Reservamos */
  array = (int *)malloc(ROWS * COLS * sizeof(int));
  if (!array) {
    printf("Can't allocate the array\n");
    return -1;
  }

  /* Inicializamos */
  for (i = 0; i < ROWS; i++)
    for (j = 0; j < COLS; j++)
      *(array + i * COLS + j) = i * 10 + j;

  /* Imprimimos */
  for (i = 0; i < ROWS; i++) {
    for (j = 0; j < COLS; j++) {
      printf("%02d ", *(array + i * COLS + j));
    }
    printf("\n");
  }

  free(array); /* Liberamos */
  return 0;
}
```

Uso tras liberación


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

struct person
{
  char name[200];
  char surname[200];
  unsigned char age;
};

int main()
{
  struct person *p;

  p = (struct person *)malloc(sizeof(struct person));

  strcpy(p->name, "Bob");
  strcpy(p->surname, "Smith");
  p->age = 20;

  free(p);

  printf("name    = %s\n", p->name);
  printf("surname = %s\n", p->surname);
  printf("age     = %d\n", p->age);

  return 0;
}
```

Depuración con valgrind

- Valgrind es una herramienta que nos avisa de los errores cometidos en el manejo de memoria
- Un **leak** o fuga de memoria es un error de programación que hace que la memoria reservada no se libere cuándo ya no se usa. Un programa que no libere correctamente la memoria reservada puede acabar consumiendo toda la memoria del sistema y dejarlo inutilizado.
- Para que valgrind nos muestre más información podemos hacer dos cosas:
  1. Compilar con símbolos de depuración (`gcc -g`)
  2. Ejecutar valgrind con el flag `-leak-check=full`

Ejemplo


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

int main()
{
  int *p = (int *)malloc(20);

  if (!p)
    return EXIT_FAILURE;

  printf("La direccion de p es %p\n", p);

  /* free(p) */

  return 0;
}
```

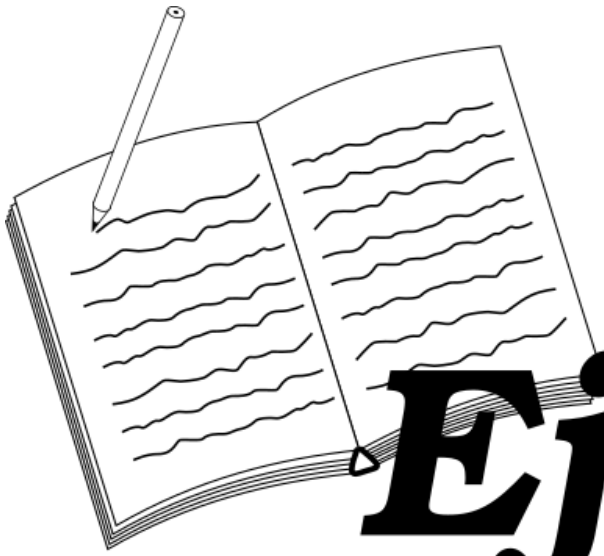
Ejemplo


```
gcc -g valgrind.c -o valg_ej
```

```
valgrind --leak-check=full valg_ej
```

```
1  Memcheck, a memory error detector
2  ....
3  Command: valg_ej
4
5  La direccion de p es 0x51d7040
6
7  HEAP SUMMARY:
8      in use at exit: 20 bytes in 1 blocks
9    total heap usage: 1 allocs, 0 frees, 20 bytes allocated
10
11 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
12    at 0x4C2ABD0: malloc (in /usr/lib/...)
13    by 0x4004F7: main (valgrind.c:6)
14
15 LEAK SUMMARY:
16    definitely lost: 20 bytes in 1 blocks
17    indirectly lost: 0 bytes in 0 blocks
18      possibly lost: 0 bytes in 0 blocks
19    still reachable: 0 bytes in 0 blocks
20         suppressed: 0 bytes in 0 blocks
21
22 For counts of detected and suppressed errors, rerun with: -v
23 ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Ejercicios: Reserva dinámica de memoria



Ejercicios