# Arquitectura de sistemas

**Abelardo Pardo**

University of Sydney
School of Electrical and Information Engineering
NSW, 2006, Australia
*Autor principal del curso de 2009 a 2012*

**Iria Estévez Ayres**

**Damaris Fuentes Lorenzo**

**Pablo Basanta Val**

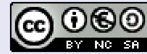
**Pedro J. Muñoz Merino**

**Hugo A. Parada**

**Derick Leony**

Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática
Avenida Universidad 30, E28911 Leganés (Madrid), España

---

© Universidad Carlos III de Madrid | Licencia Creative Commons 

# Capítulo 6. Llamadas al sistema para gestión de memoria en C

**Tabla de contenidos**

6.1. Los tipos de memoria de un programa en C
6.2. La pila y las variables locales
6.3. El *heap* y la memoria dinámica
6.4. La función `sizeof()`
6.5. Llamadas a las funciones de gestión de memoria
6.6. Tablas y punteros
6.7. Bibliografía de apoyo
6.8. Autoevaluación automática
6.9. 20 problemas de memoria dinámica
6.10. Anomalías en la gestión de memoria en C
    6.10.1. La trastienda de la gestión de memoria
    6.10.2. La "fuga" de memoria
    6.10.3. Memoria sin inicializar
    6.10.4. Sobre-escritura de memoria dinámica
    6.10.5. Acceso a memoria con un puntero corrupto
6.11. Problemas sobre fugas de memoria.

Cuando se escribe un programa, se asume que las variables se almacenan en memoria y están ahí disponibles para ser utilizadas. En principio, los detalles de cómo se almacenan y organizan los datos en memoria no son visibles a un programador. Sin embargo, como el lenguaje de programación C nos ofrece una gestión de memoria muy cercana a la RAM, para realizar un uso eficiente de la memoria es preciso conocer más de cerca cómo se organiza la memoria de un programa.

## 6.1. Los tipos de memoria de un programa en C

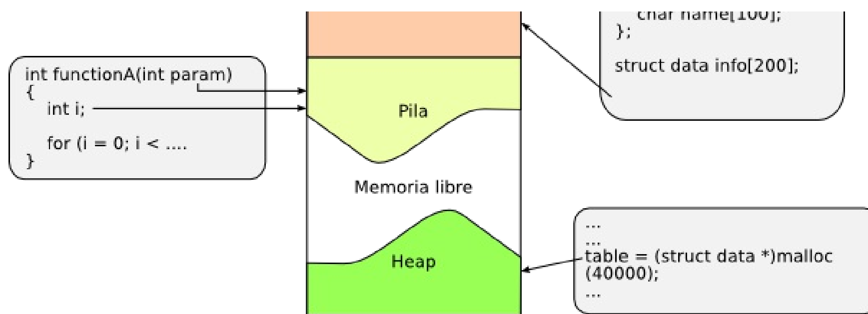
Un programa en C almacena sus datos en memoria en tres áreas diferentes:

1. Memoria global. Es el área en la que están almacenadas las variables que se declaran globales o estáticas y las constantes de tipo cadena de caracteres (por ejemplo "`Mi string`"). Es decir, en esta zona de memoria se almacenan todos aquellos datos que están presentes desde el comienzo del programa hasta que termina.

2. La pila. Es un área en la que las variables aparecen y desaparecen en un momento puntual de la ejecución de un programa. Se utiliza principalmente para almacenar variables locales a las funciones. Estas variables tienen un ámbito reducido, sólo están disponibles mientras se está ejecutando la función en la que han sido definidas. En la pila se encuentran todas estas variables, y por tanto, en esa zona se está continuamente insertando y borrando variables nuevas.

3. El *heap*. Esta zona (traducida en algunos casos como "el montón") contiene memoria disponible para que se reserve y libere en cualquier momento durante la ejecución de un programa. No está dedicada a variables locales de las funciones como la pila, sino que es memoria denominada "dinámica" para estructuras de datos que no se saben si se necesitan, e incluso tampoco se sabe su tamaño hasta que el programa está ejecutando.

Nótese que de estas tres zonas sólo la memoria global tiene un tamaño fijo y que se sabe cuando comienza la ejecución de un programa. Tanto la pila como el *heap* albergan datos cuyo tamaño no se puede saber hasta que el programa está en ejecución. La siguiente figura muestra estas tres zonas de memoria.




Como las áreas de *heap* y pila tienen un tamaño variable, el sistema operativo reserva un espacio inicial y las dos zonas crecen y decrecen dentro de ese espacio máximo.

Responde a las siguientes preguntas para ver si has entendido lo que se explica en este documento:

1. Una variable estática se almacena en
   - Memoria global
   - Memoria de pila
   - El heap
2. Un parámetro que se pasa a una función se almacena en
   - Memoria global
   - Memoria de pila
   - El heap
3. Una variable local se almacena en
   - Memoria global
   - Memoria de pila
   - El heap

## 6.2. La pila y las variables locales

El funcionamiento de la pila es como un espacio de anotación temporal. Por ejemplo, cuando se invoca una función, sus variables temporales están activas sólo durante su ejecución. Cuando la función termina, esas variables ya no existen. No tiene lógica reservar un espacio en la memoria global para estas variables. En la pila, por tanto, se crean espacios para estas variables y se destruyen al acabar la función. Esta es la razón por la que si se quiere conservar algún valor que se obtiene en una función, se ha de almacenar en una variable global (que por tanto está almacenada en la zona global), manipularse a través de un puntero, o devolverse como resultado, que se traduce en que su valor se copia a otra variable.

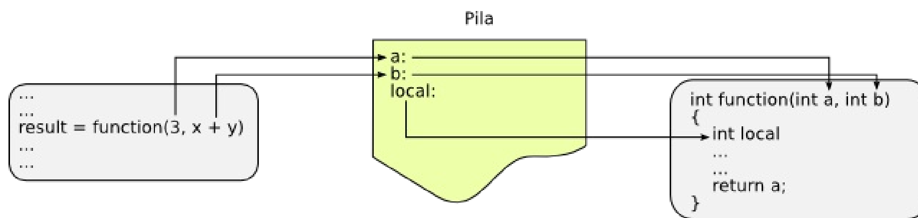
Los parámetros de una función también se consideran variables locales. Su ámbito de validez es igual al de las variables locales de una función con la salvedad de que comienzan la ejecución con un valor inicial que viene dado por el código que realiza la llamada.

El mecanismo de paso de parámetros y devolución de resultados de una función en C se puede ilustrar mediante las operaciones que se realizan en la zona de la pila. Supongamos que existe una función definida como `int function(int a, int b)`, y que se llama mediante la línea

```
result = function(3, x + y);
```

Cuando se ejecuta la llamada, se crea un nuevo espacio en la pila en el que se almacenan los parámetros y las variables locales de la función `function`. Sobre el espacio creado para el primer parámetro `a` se copia el valor `3`. Análogamente, sobre el espacio creado para el segundo parámetro se

copia el valor resultante de evaluar la expresión `x + y`. Tras inicializar con estos valores a los parámetros en la pila, la función pasa a ejecutarse. Nótese que las variables `x` e `y` no son necesariamente variables de la función, pero sí deben ser válidas en el código que contiene llamada. La siguiente figura ilustra estas operaciones en la pila.




A la vista del funcionamiento del paso de parámetros en C, explica por qué la siguiente función, aunque se pretende que intercambie los valores de dos variables, no lo consigue.

| Llamada a función | Definición de función |
| --- | --- |
| `...`<br>`...`<br>`x = 10;`<br>`y = 20;`<br>`swap(x, y);`<br>`printf("x = %d, y = %d\n", x, y);`<br>`...` | `void swap(int a, int b) {`<br>`    int tmp;`<br>`    tmp = a;`<br>`    a = b;`<br>`    b = tmp;`<br>`    return;`<br>`}` |

¿De qué forma redefinirías la función `swap` para que el intercambio de los valores de las variables sea visible fuera de la función?

## 6.3. El *heap* y la memoria dinámica

La memoria dinámica que se almacena en el *heap* es aquella que se utiliza para almacenar datos que se crean en el medio de la ejecución de un programa. En general, este tipo de datos puede llegar a ser casi la totalidad de los datos de un programa. Por ejemplo, supóngase un programa que abre un fichero y lee una colección de palabras. ¿Cuántas palabras y de qué tamaño hay en el fichero? Hasta que no se procese el fichero en su totalidad, no es posible saberlo.

La manipulación de memoria en C se hace con un mecanismo muy simple, pero a la vez muy propenso a errores. Los dos tipos de operaciones son la petición y liberación de memoria. El ciclo es sencillo, cuando se precisa almacenar un nuevo dato, se solicita tanta memoria en bytes como sea necesaria, y una vez que ese dato ya no se necesita la memoria se devuelve para poder ser reutilizada. Este esquema se conoce como “gestión explícita de memoria” pues requiere ejecutar una operación para pedir la memoria y otra para liberarla.

Las cuatro operaciones principales para gestionar memoria en C son:

- `void *malloc(size_t size)`. Es la función para reservar tantos bytes consecutivos de memoria como indica su único parámetro. Devuelve la dirección de memoria de la porción reservada. La memoria no se inicializa a ningún valor.
- `void *calloc(size_t nmemb, size_t size)`. Reserva espacio para tantos elementos como indica su primer parámetro `nmemb`, y cada uno de ellos con un tamaño en bytes como indica el segundo. En otras palabras, reserva `nmemb * size` bytes consecutivos en memoria. Al igual que la función anterior devuelve la dirección de memoria al comienzo del bloque reservado. Esta función inicializa todos los bytes de la zona reservada al valor cero.
- `void free(void *ptr)`. Función que dado un puntero, libera el espacio previamente reservado. El puntero que recibe como parámetro esta función tiene que ser el que se ha obtenido con una llamada de reserva de memoria. No es necesario incluir el tamaño. Una vez que se ejecuta esta llamada, los datos en esa porción de memoria se consideran basura, y por tanto pueden ser reutilizados por el sistema.
- `void *realloc(void *ptr, size_t size)`. Función para redimensionar una porción de memoria previamente reservada a la que apunta el primer parámetro al tamaño dado como

segundo parámetro. La función devuelve la dirección de memoria de esta nueva porción redimensionada, que no tiene por qué ser necesariamente igual al que se ha pasado como parámetro. Los datos se conservan intactos en tantos bytes como el mínimo entre el tamaño antiguo y el nuevo.

Responde a las siguientes preguntas para ver si has entendido lo que se explica en este documento:

1. Si tenemos que reservar una nueva porción de memoria dinámica e inicializarla a 0, utilizaremos
   - malloc
   - calloc
   - realloc
   - free

2. En un programa desde el inicio hasta su finalización, debemos realizar tantas invocaciones a "free" como la suma de invocaciones de malloc y calloc
   - Verdadero
   - Falso

## 6.4. La función `sizeof()`

Para reservar memoria se debe saber exactamente el número de bytes que ocupa cualquier estructura de datos. Tal y como se ha comentado con anterioridad, una peculiaridad del lenguaje C es que estos tamaños pueden variar de una plataforma a otra. ¿Cómo sabemos, entonces, cuántos bytes reservar para una tabla de, por ejemplo, 10 enteros? El propio lenguaje ofrece la solución a este problema mediante la función `sizeof()`.

La función recibe como único parámetro o el nombre de una variable, o el nombre de un tipo de datos, y devuelve su tamaño en bytes. De esta forma, `sizeof(int)` devuelve el número de bytes que se utilizan para almacenar un entero. La función se puede utilizar también con tipos de datos estructurados o uniones tal y como se muestra en el siguiente programa (que te recomendamos que te descargues, compiles y ejecutes):

```
#include <stdio.h>
#define NAME_LENGTH 10
#define TABLE_SIZE 100
#define UNITS_NUMBER 10

struct unit
{  /* Define a struct with an internal union */
  int x;
  float y;
  double z;
  short int a;
  long b;
  union
  { /* Union with no name because it is internal to the struct */
    char name[NAME_LENGTH];
    int id;
    short int sid;
  } identifier;
};

int main(int argc, char *argv[])
{
  int table[TABLE_SIZE];
```

```
    struct unit data[UNITS_NUMBER];

    printf("%d\n", sizeof(struct unit)); /* Print size of structure */
    printf("%d\n", sizeof(table));       /* Print size of table of ints */
    printf("%d\n", sizeof(data));        /* Print size of table of structs */

    return 0;
}
```

Con esta función puedes, por tanto, resolver cualquier duda que tengas sobre el tamaño de una estructura de datos. Basta con escribir un pequeño programa que imprima su tamaño con `sizeof()`.

## 6.5. Llamadas a las funciones de gestión de memoria

La primera observación cuando se ven los tipos de datos que manipulan las funciones de gestión de memoria es que el uso de memoria dinámica y el de punteros van de la mano. Un programa puede manipular una porción de memoria dinámica porque dispone de la dirección de su primer byte, y esta se almacena en un dato de tipo puntero. Este puntero se utiliza tanto para acceder a los datos que se almacenen en la memoria reservada, como para liberar ésta cuando no sea necesaria (con la función `free()`).

Cuando se llama a la función `malloc` el valor que retorna es una dirección de memoria de tipo `void *`, es decir, puntero a cualquier dato. Pero esa dirección de memoria se almacena en un programa en un puntero de un tipo diferente. Para evitar advertencias del compilador, se antepone a la llamada a `malloc` lo que se conoce como un "*casting*". Un casting es una expresión escrita por el programador notificando al compilador de que una variable de un cierto tipo ahora debe ser considerada de otro tipo. Por ejemplo, la siguiente expresión reserva espacio para una estructura de tipo `struct cell_info` y la dirección de ese espacio se asigna al puntero `cell_ptr`.

```
#define TABLE_SIZE 10
struct cell_info
{
    int a;
    int b;
    int table[TABLE_SIZE];
};
struct cell_info *cell_ptr;

cell_ptr = (struct cell_info *)malloc(sizeof(struct cell_info));
```

Nótese que para saber el tamaño exacto de la porción de memoria a reservar se utiliza la función `sizeof()` que, dado cualquier tipo de datos o variable, devuelve el número de bytes que ocupa en memoria. El casting `(struct cell_info *)` hace que la asignación sobre la variable `cell_ptr` se haga como si la dirección de memoria fuese de este tipo, y por tanto es correcta.

Una vez obtenida la porción de memoria, el acceso se realiza normalmente utilizando el puntero obtenido, por ejemplo:

```
cell_ptr->a = 10;
cell_ptr->b = a + 20;
cell_ptr->table[5] = 0;
```

La regla general para invocar a la función `malloc` se puede enunciar de la siguiente forma. El espacio en memoria dinámica para una variable de tipo `T` se obtiene mediante

```
T *ptr = (T *)malloc(sizeof(T));
```

Una vez que una estructura de datos reservada con memoria dinámica no se necesita, se libera mediante la llamada a la función `free`. Siguiendo con el ejemplo anterior, se invoca con la línea

```
free(cell ptr)
```

Responde a las siguientes preguntas para ver si has entendido lo que se explica en este documento:

1. Las llamadas "malloc" y "calloc" en caso de que no se produzca error devuelven
   - Un puntero al inicio de la memoria reservada
   - Un puntero al final de la memoria reservada
   - Un puntero en medio de la memoria reservada
2. Si p es un puntero a entero, y realizamos p=p+1;
   - El valor de p incrementa el tamaño de un entero
   - El valor de p incrementa el tamaño de un puntero

## 6.6. Tablas y punteros

Aunque el concepto de “tabla” en un lenguaje de programación es intuitivo, en C las tablas se manipulan de una forma especial que debe ser entendida para evitar errores al combinar su uso con memoria dinámica. Para el lenguaje de programación C, una tabla se representa internamente sólo como la dirección de memoria a partir de la cual está almacenado su primer elemento. Otra forma de verlo, en C se define una tabla de 100 elementos de tipo `T` como `T tabla[100]`, pero una vez definida (y reservado su espacio correspondiente ya sea en la memoria global o en la pila), en adelante, el nombre de la tabla y la dirección de su primer elemento es exactamente lo mismo. Esta particularidad de C se puede resumir diciendo que para toda tabla en C la siguiente expresión es siempre cierta:

```
tabla == &tabla[0]
```

La principal consecuencia es que cuando es preciso reservar espacio en memoria dinámica para una tabla, la reserva devuelve un puntero, pero este puede utilizarse directamente como una tabla. Por ejemplo, el siguiente código crea una tabla de tantos elementos como indica la variable `size` y los inicializa:

```
struct element
{
  float number1;
  int number2;
  char letter;
};

struct element *table;

table = (struct element *)malloc(sizeof(struct element) * size);
for (i = 0; i < size; i++)
{
  table[i].number1 = 0.0;
  table[i].number2 = 10;
  table[i].letter = 'B';
}
```

Nótese que el acceso a los elementos de la tabla, tras incluir el índice entre corchetes, se realiza como una estructura, y no como un puntero.

Por lo tanto existe una gran relación entre punteros y tablas. Con una variable puntero que apunta al primer elemento de una tabla, se pueden realizar las mismas operaciones (y con la misma notación) que las que se pueden hacer con la tabla definida. Sin embargo, no todas las operaciones que se pueden realizar con un puntero son válidas para realizarlas con una tabla. Así a un puntero se le puede asignar otro puntero o la dirección inicial de una tabla, mientras que sin embargo a una tabla

declarada no se le puede asignar otro puntero o la dirección inicial de otra tabla. Esto es porque al declarar una tabla su dirección permanece fija, mientras que el puntero puede admitir nuevas asignaciones. El siguiente código ilustra esta situación:

```
1  #define SIZE 30
2  char cadena1[SIZE];
3  char *cadena2;
4  cadena1 = "Pulsa espacio para continuar"; /* Incorrecto */
5  cadena2 = "Pulsa espacio para continuar"; /* Correcto */
```

La línea 4 es incorrecta porque a la tabla `cadena1` no se le puede asignar una dirección nueva, ya se le asigna al ser declarada. Por el contrario, `cadena2` se declara como puntero, y sí se le puede asignar una dirección cualquiera, y en la línea 5 se le asigna la de la cadena “Pulsa espacio para continuar”.

## 6.7. Bibliografía de apoyo

- ***Funciones de gestión de memoria.***:
  - (Teoría) “"The GNU tutorial"”, página 212.
  - (Teoría y ejemplos) “Programación en C: Metodología, algoritmos y estructura de datos”, páginas 442-462.
  - (Teoría y ejemplos) “Practical C Programming”, páginas 252-255.
- ***Tablas y punteros (teoría y ejemplos)***: “Programación en C: Metodología, algoritmos y estructura de datos”, páginas 415-424.
- ***Problemas resueltos***: “Problemas resueltos de Programación en Lenguaje C”, problemas 4.6-4.11.

## 6.8. Autoevaluación automática

Comprueba con estas preguntas si has entendido este documento

1. Se desea convertir una función que devuelve la suma de dos enteros en una función que devuelve la suma y en un parámetro adicional, la resta. Selecciona el prototipo adecuado.

   - `int sum_subs(int a, int b, int subs);`
   - `int *sum_subs(int a, int b, int *subs);`
   - `int sum_subs(int a, int b, int *subs);`
   - `void sum_subs(int a, int b, int subs);`
   - `void sum_subs(int a, int b, int *subs);`

2. Se desea usar la función `multiple`, que multiplica dos enteros y devuelve en un parámetro adicional su multiplicación. Esta función supone que la variable resultado ya existe, es decir, NO hace un `malloc` internamente. Selecciona cómo realizar la llamada a dicha función SIN usar `malloc` en el programa llamante.

   - ```
     int a=5,b=6,resultado;
     resultado=multiple(a,b);
     ```
   - ```
     int a=5,b=6,*resultado;
     multiple(a,b,resultado);
     ```

-
  ```
  int a=5,b=6,resultado;
  multiple(a,b,&resultado);
  ```
-
  ```
  int a=5,b=6,resultado;
  multiple(a,b,*resultado);
  ```
-
  ```
  int a=5,b=6,*resultado;
  multiple(a,b,&resultado);
  ```

3. Se desea usar la función `multiple`, que multiplica dos enteros y devuelve en un parámetro adicional su multiplicación. Esta función supone que la variable resultado ya existe, es decir, NO hace un `malloc` internamente. Selecciona cómo realizar la llamada a dicha función usando `malloc` en el programa llamante.

-
  ```
  int a=5,b=6,resultado;
  resultado=(int)malloc(1);
  multiple(a,b,resultado);
  ```
-
  ```
  int a=5,b=6,*resultado;
  resultado=(int *) malloc(sizeof(int));
  multiple(a,b,resultado);
  ```
-
  ```
  int a=5,b=6,*resultado;
  resultado=(int *) malloc(sizeof(int));
  multiple(a,b,&resultado);
  ```
-
  ```
  int a=5,b=6,resultado;
  resultado=(int) malloc(sizeof(int));
  multiple(a,b,*resultado);
  ```
-
  ```
  int a=5,b=6,*resultado;
  resultado=(int *)malloc(sizeof(int));
  multiple(a,b,*resultado);
  ```

4. Se desea implementar la función `multiple`, que multiplica dos enteros y devuelve en un parámetro adicional su multiplicación. Esta función supone que la variable resultado ya existe, es decir, NO hace un `malloc` internamente. Selecciona el código adecuado para dicha función.

-
  ```
  void multiple(int a, int b, int *result)
  {
  ```

```
 result=a*b;
}
```

- ```
  void multiple(int a, int b, int *result)
  {
   return a*b;
  }
  ```
- ```
  void multiple(int a, int b, int result)
  {
   result=a*b;
  }
  ```
- ```
  void multiple(int a, int b, int *result)
  {
   &result=a*b;
  }
  ```
- ```
  void multiple(int a, int b, int *result)
  {
   *result=a*b;
  }
  ```

5. Se desea definir una función, que dado un número de alumnos, devuelva un vector de enteros inicializado a cero. Seleccione el código adecuado.

- ```
  void set_up(int number_students, int *vector)
  {
   *vector = (int *) calloc (number_students,sizeof(int));
  }
  ```
- ```
  void set_up(int number_students, int *vector)
  {
   *vector = (int *) calloc (number_students*sizeof(int));
  }
  ```
- ```
  int *set_up(int number_students)
  {
   int *vector = (int *) calloc (number_students,sizeof(int));
   return &vector;
  }
  ```
- ```
  int *set_up(int number_students)
  {
  ```

```
 int *vector = (int *) calloc (number_students,sizeof(int));
 return vector;
}
```

- ```
  int *set_up(int number_students)
  {
  int *vector = (int *) calloc (number_students*sizeof(int));
  return vector;
  }
  ```

6. Se desea definir una función, que dado un número `n`, devuelva en un vector que se le pasa como parámetro los `n` primeros números naturales, comenzando por el cero. La función deberá reservar espacio para dicho vector. Seleccione el código adecuado.

- ```
  void first_n(int number, int **vector)
  {
      int i;
      *vector = (int *)malloc (number * sizeof(int));
      for (i = 0;i < number; i++) {
          (*vector)[i]=i;
      }
  }
  ```

- ```
  void first_n(int number, int *vector)
  {
      int i;
      vector = (int *) malloc (number*sizeof(int));
      for (i = 0;i <= number; i++)
      {
         vector[i]=i;
      }
  }
  ```

- ```
  void first_n(int number, int *vector)
  {
      int i;
      vector = (int *)malloc(number, sizeof(int));
      for (i = 0; i < number; i++)
      {
          vector[i]=i;
      }
  }
  ```

- ```
  int *first_n(int number)
  {
      int i;
      int *vector = (int *)malloc(number * sizeof(int));
      for (i = 0;i < number; i++) {
          vector[i]=i;
  ```
  ```

```
    }
    return vector;
}
```

## 6.9. 20 problemas de memoria dinámica

Los siguientes 20 problemas asumen que conoces las llamadas al sistema para reservar y liberar memoria dinámica. Se suponen los siguientes tamaños de los tipos de datos básicos:

| Tipo | Tamaño (bytes) |
|---|---|
| `char`, `unsigned char` | 1 |
| `short int`, `unsigned short int` | 2 |
| `int`, `unsigned int`, `long int`, `unsigned long int` | 4 |
| `float` | 4 |
| `double`, `long double` | 8 |
| Puntero de cualquier tipo | 4 |

Para la resolución de algunos de los siguientes problemas se considera el siguiente programa en C de ejemplo:

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

#define MAC_LENGTH 6

struct bluetooth_info
{
    char *name;
    unsigned int strength;
    char mac[MAC_LENGTH];
};

struct bluetooth_info info, *new_info;

struct bluetooth_info *duplicate(struct bluetooth_info *);

int main(int argc, char **argv)
{
    struct bluetooth_info *info_ptr;
    int i;

    info_ptr = &info;
    info_ptr->name = "My phone";
    info_ptr->strength = 100;
    for (i = 0; i < MAC_LENGTH; i++)
    {
      info_ptr->mac[i] = 10;
    }
    new_info = duplicate(info_ptr);
    free(new_info);

    return 0;
}

struct bluetooth_info *duplicate(struct bluetooth_info *src_ptr)
{
    struct bluetooth_info *result;
    int i;
```

```
    result = (struct bluetooth_info *)malloc(sizeof(struct bluetooth_info));
    result->name = (char *)malloc(strlen(src_ptr->name) + 1);
    strcpy(result->name, src_ptr->name);
    result->strength = src_ptr->strength;
    for (i = 0; i < MAC_LENGTH; i++)
    {
      result->mac[i] = src_ptr->mac[i];
    }
    return result;
}
```

Se recomienda que descargues este programa en tu área de trabajo para poder hacer cambios puntuales, compilarlo y ejecutarlo. Algunos de los siguientes problemas se pueden resolver de esta forma.

1. ¿Qué tamaño tiene la estructura que se define en las líneas 6 a 11?
2. ¿Por qué crees que se pone la línea 15? Prueba a comentarla y compila el programa.
3. En la línea 23 se realiza una asignación de una cadena de texto, ¿en qué tipo de memoria (global, heap, pila) crees que está esa cadena?
4. Reescribe las líneas 23 y 24 pero en lugar de utilizar la variable `info_ptr` utiliza directamente `info`. ¿Puedes escribir la función `main` para que haga exactamente lo mismo pero suprimiendo la variable `info_ptr`?
5. Busca qué hace exactamente la función `strlen` que se utiliza en la línea 41. ¿Qué resultado devuelve para el caso del programa? ¿Cuánta memoria se está reservando en esa invocación a `malloc`?
6. Busca qué hace la función `strcpy` de la línea 42 y explica qué memoria se está modificando y dónde está para el caso del programa de ejemplo.
7. ¿Cómo accederías a la tercera letra de la cadena del campo `name` de la estructura a la que apunta `result` en la línea 43? (Puedes responder a esta pregunta insertando una línea que imprima la respuesta y ejecutando el programa)
8. ¿Dónde se copia el valor del puntero `result` al ejecutar la línea 48?
9. ¿Cuántos bytes de memoria dinámica se utilizan en el programa de ejemplo?
10. Clasifica las siguientes variables del programa ejemplo.

| Línea | Variable | Memoria global | Heap | Pila | Sin memoria |
|---|---|---|---|---|---|
| 6-11 | `struct bluetooth_info` | | | | |
| 13 | `info` | | | | |
| 13 | `new_info` | | | | |
| 19 | `info_ptr` | | | | |
| 20 | `i` | | | | |
| 23 | `info_ptr->name` | | | | |
| 24 | `info_ptr->strength` | | | | |
| 35 | `src_ptr` | | | | |
| 37 | `result` | | | | |
| 38 | `i` | | | | |
| 40 | `malloc(sizeof(struct bluetooth_info))` | | | | |
| 43 | `result->strength` | | | | |

11. La función `duplicate` definida en las líneas 35 a 49, como su nombre indica, crea una estructura que es un duplicado de la que apunta el parámetro dado. En la línea 29 del `main` se obtiene ese duplicado y se almacena en `new_info`. ¿Cuál sería el efecto de, en lugar de llamar a `duplicate` simplemente hacer `new_info = info_ptr`?

12. La duplicación del campo `name` de la estructura a la que apunta el parámetro se realiza en las líneas 41 y 42. ¿Por qué no basta con poner simplemente `result->name = src_ptr->name`?

13. ¿Cómo reservarías espacio para una tabla de 200 elementos de la estructura `struct bluetooth_info` y a la vez inicializar su contenido todo al valor cero? Asigna la dirección de ese espacio al puntero `ptr`. Escribe la porción de código equivalente pero utilizando `malloc`.

14. En un programa en C hay que reservar espacio en memoria dinámica para una tabla de 100 estructuras de datos previamente definidas, pero no es preciso inicializarlas a ningún valor en particular. ¿Cuál de las dos funciones `malloc` y `calloc` utilizarías? ¿Por qué?

15. Un programa define la siguiente estructura de datos en la que almacena una tabla de enteros y una tabla de letras pero de tamaño desconocido:

```
struct two_tables
{
    int *int_table;
    char *char_table;
};
```

    Escribe una función que recibe dos enteros, reserva una estructura de este tipo y con tablas de tamaños igual a los valores de los enteros y devuelve el puntero a esta estructura. No es preciso inicializar ningún dato. Pista: se requiere más de una llamada a `malloc`.

    Escribe la función contraria, es decir, dado un puntero a una estructura de este tipo, libera la memoria que ocupa la estructura y las tablas.

    ¿Cómo reservarías espacio en memoria para una tabla de 100 estructuras de tipo `struct two_tables`? ¿Cuánto espacio ocupa en memoria?

16. Una aplicación gráfica mantiene una tabla con un conjunto de puntos que representa como números enteros. El número de puntos fluctua en un rango entre cero y un millón de puntos. Para no tener reservada memoria para un millón de puntos, la aplicación comienza por reservar espacio sólo para 1000 puntos con la línea:

```
points = (int *)malloc(1000 * sizeof(int));
```

    La aplicación lleva la cuenta de los puntos que tiene almacenados, y al cabo de un rato ejecutando necesita almacenar más de 1000 puntos. ¿Qué operación sobre memoria dinámica necesitas ejecutar? ¿Qué parámetros utilizarías? Justifica tu respuesta.

17. Una aplicación almacena la información sobre contactos en una tabla con 100 estructuras como la siguiente:

```
struct contact
{
    char *name;
    char *lastname;
};
```

    La variable `table` almacena estos elementos y su espacio ha sido reservado de forma dinámica (con `malloc` o `calloc`). De forma análoga, para cada elemento, el espacio al que apuntan los campos `name` y `lastname` también ha sido reservado de forma dinámica. Escribe la función que recibe como parámetro un puntero a una tabla de este tipo y un entero con su número de elementos, y que libera toda la memoria ocupada por la tabla.

```
void nuke(struct contact *table, int size)
```

```
{
...
...
}
```

18. Supóngase la siguiente definición de estructura de datos, variables y código:

```
struct unit
{
    struct unit *next;
} *unit1, *unit2, *unit3;

unit1 = (struct unit *)malloc(sizeof(struct unit));
unit2 = (struct unit *)malloc(sizeof(struct unit));
unit3 = (struct unit *)malloc(sizeof(struct unit));
unit1->next = unit2;
unit2->next = unit3;
unit3->next = NULL;
```

    Escribe la función `void borra(struct unit *ptr)` tal que si se invoca como `borra(unit1)` libera la memoria reservada de las tres estructuras. ¿Tienes que cambiar algo en tu función para que haga la misma operación para una longitud arbitraria de la cadena de estructuras?

19. 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:

    1. ¿Qué valor debería tener justo antes de terminar la ejecución?
    2. Y si un programa termina su ejecución y esta variable vale cero, ¿es esto suficiente para concluir que hace una gestión de memoria correcta?

20. Un programa en C manipula un número muy elevado de elementos de las dos siguientes estructuras de datos:

```
struct picture_info
{
    char *name;
    char *location;
    char *note;
    int size;
    int latitude;
    int longitude;
};
struct video_info
{
    char *location;
    int *encoding
    int length;
};
```

    Todos los elementos se crean utilizando memoria dinámica y el código de la aplicación está perfectamente dividido en dos partes, cada una para manipular los elementos de una de las estructuras.

    Al terminar la ejecución, el programa tiene porciones de memoria que se han reservado pero no liberado. Se utiliza una herramienta que supervisa esta ejecución y al final emite un informe en el que consta que 2345 porciones de memoria, todas ellas de 12 bytes, han sido reservadas pero no liberadas. ¿En qué parte del código empezarías a buscar la anomalía en la gestión de memoria y por qué?

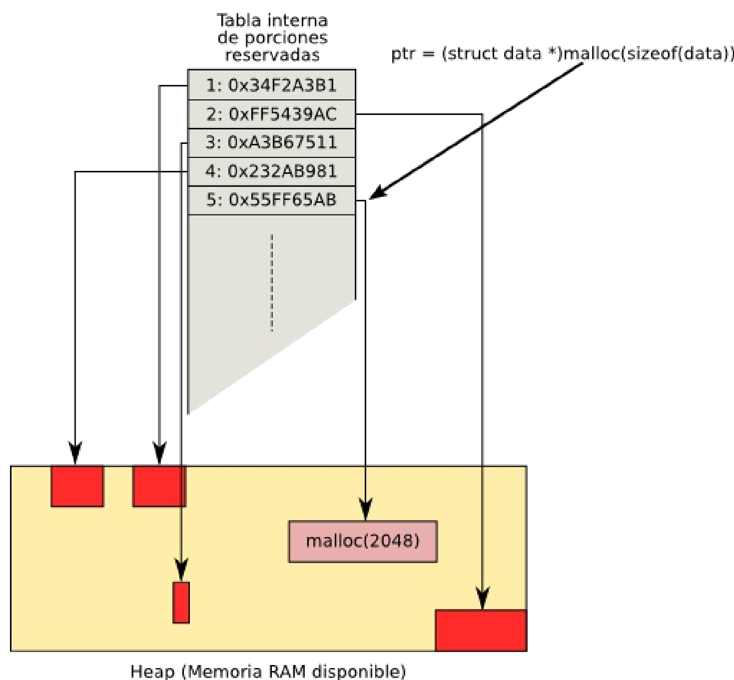
## 6.10. Anomalías en la gestión de memoria en C

La gestión de memoria en C se describe como "explícita" porque tanto las operaciones de reserva como las de liberación han de aparecer explícitamente en el código. En otros lenguajes de programación, como por ejemplo Java, el entorno de ejecución se encarga de recuperar aquellas porciones de memoria que ya no se utilizan, liberando al programador de escribir este código. De la gestión explícita de memoria se derivan varias posibles anomalías cuando fragmentos de memoria no se pueden liberar porque se ha perdido cualquier referencia a ellos. Es lo que se conoce como "fuga de memoria". Esa porción de memoria permanece reservada pero inaccesible para el resto de la ejecución de un programa.

### 6.10.1. La trastienda de la gestión de memoria

Al comenzar la ejecución de un programa, su memoria se divide en tres zonas: la pila, memoria global y el "*heap*". El heap se utiliza para la reserva y liberación de porciones de memoria durante la ejecución del programa. Pero ¿cómo se gestiona esta memoria?

El sistema operativo mantiene una tabla interna en la que apunta qué fragmentos del heap están ocupados y qué punteros se han devuelto como resultado de la petición de reserva. Cuando un programa ejecuta la función `malloc` para pedir un nuevo fragmento, el sistema busca una porción del tamaño solicitado, si existe devuelve su dirección de comienzo, y se apunta ese bloque como ocupado. De manera análoga, cuando se llama a la función `free` para liberar un fragmento, el sistema busca en la tabla ese fragmento (que debe estar apuntado previamente como reservado) y libera el espacio para usos futuros. En la siguiente figura se ilustra este funcionamiento.




A la petición de reserva de una porción de 2048 bytes, el gestor de memoria responde con la dirección de un bloque que previamente marca como ocupado. La llamada a `free` es análoga, pero se recibe una dirección de memoria de un bloque previamente reservado, se busca en la tabla, y si existe, se marca de nuevo como disponible.

Este esquema de gestión de memoria hace que los programas deban ceñirse a unas pautas muy concretas para garantizar el correcto uso de la memoria y sacar el mayor rendimiento a un programa. Por ejemplo, si un programa utiliza una cantidad muy alta de datos dinámicos (esto es, que se almacenan en la memoria solicitada al gestor mediante `malloc`) y no libera esa memora en cuanto puede, corre el riesgo de agotar la memoria y no terminar la ejecución.

### 6.10.2. La "fuga" de memoria

Una de las anomalías más comunes cuando se gestiona la memoria de forma explícita es lo que se conoce como "fuga de memoria". Esta situación ocurre cuando un programa obtiene memoria dinámica, y el valor del puntero que devuelve el sistema, por error, se pierde. En tal caso, ya no es posible invocar a la función `free` con ese puntero, y la porción de memoria se queda reservada por lo que resta de ejecución. Como ejemplo de fuga de memoria analicemos el siguiente fragmento de

que resta de ejecución. Como ejemplo de fuga de memoria analicemos el siguiente fragmento de código.

```
char *string;
  string = (char *)malloc(100);
  string = NULL;
```

La primera línea declara un puntero a carácter. En la segunda se reserva un espacio de 100 bytes. El gestor de memoria devuelve un puntero al comienzo de ese bloque y se almacena en la variable `string`. En ese momento, la dirección de ese bloque no está almacenada en ningún otro sitio. La línea siguiente asigna el valor `NULL` al mismo puntero. ¿Qué ha sucedido con la dirección de memoria de la porción que se acaba de reservar? Se ha perdido y no hay forma alguna de recuperarla, porque `string` era la única copia de ese valor. Como consecuencia, la porción de memoria reservada seguirá marcada como ocupada por el resto de ejecución del programa. La memoria se ha fugado.

La principal consecuencia de una fuga de memoria, por tanto, es que esa porción no se puede utilizar, se ha perdido. Esto es equivalente a que la memoria disponible para la ejecución del programa se haya reducido. Los efectos de una fuga de memoria dependen del lugar en el código en el que se produzca. Si en un programa se fuga una única porción de unos cuantos bytes, es posible que su efecto pase desapercibido. Sin embargo, si la pérdida de memoria se produce en un lugar que se ejecuta un número muy elevado de veces, el efecto puede ser mucho más notorio. Fíjate en el siguiente fragmento de programa:

```
#define MILLION 1000000

char *table[MILLION];
for (i = 0; i < MILLION; i++) {
  table[i] = (char *)malloc(100);
  table[i] = NULL;
}
```

La fuga de memoria se produce en un lugar que forma parte de un bucle que se ejecuta un millón de veces. En este bucle se fugan casi 100 Megabytes de memoria.

Las fugas de memoria no se producen en situaciones tan obvias como las descritas anteriormente, sino que aparecen en lugares del código inesperados debidos a despistes en la manipulación de punteros. El problema de las fugas de memoria en C es tan complicado de solventar que han aparecido herramientas especializadas, tanto comerciales como de código libre, especialmente concebidas para analizar un programa y detectar fugas.

Una situación típica de fuga de memoria es cuando se manipulan estructuras de datos encadenadas. En una estructura se almacenan punteros obtenidos mediante llamadas a `malloc` y en ellos a su vez se almacenan más punteros obtenidos de esta manera. La liberación de la memoria que ocupan estas estructuras de datos ha de programarse con sumo cuidado. El siguiente fragmento de código ilustra este problema.

```
struct contact_information
{
  char *name, *lastname;
  int age;
};

struct contact_information *agenda;
int i;

agenda = (struct contact_information *)calloc(100, sizeof(struct contact_information));
for (i = 0; i < 100; i++) {
  agenda[i].name = (char *)malloc(10);
  agenda[i].lastname = (char *)malloc(30);
  agenda.age = 0;
}
free(agenda);
```

La variable agenda se reserva con espacio suficiente para almacenar 100 estructuras del tipo struct contact_information. En el bucle, los dos primeros campos de cada una de las estructuras se inicializa con dos punteros que se obtienen mediante malloc. Al terminar el bucle, la llamada free(agenda) libera el espacio reservado para la tabla, pero no el que se ha reservado para las cadenas de texto de cada uno de sus elementos. La forma correcta de liberar la estructura es igualmente con un bucle que atraviese la tabla y libere cada campo por separado con una llamada a free.

Las dos reglas a respetar en cualquier programa en C en lo referente a la gestión dinámica de memoria son:

1. Toda porción reservada de forma dinámica (con malloc, calloc o realloc) debe ser liberada mediante una llamada a free.

2. Si un programa llega a su última instrucción y tiene bloques de memoria dinámica sin liberar, se considera que el programa es erróneo.

Desafortunadamente, no hay una técnica concreta para evitar las fugas de memoria, pero sí hay herramientas que dado un programa lo analizan para ofrecerte un informe sobre qué memoria se ha fugado (si ha habido alguna). Para darte una idea de la dificultad de este problema, cuando las primeras herramientas de detección de fugas aparecieron, se utilizaron para analizar aplicaciones que se consideraban sólidas y maduras, y para sorpresa de sus diseñadores, se detectaron fugas que hasta el momento ningún programador había detectado.

### 6.10.3. Memoria sin inicializar

Otra característica de la gestión dinámica de memoria en C es que la inicialización de la memoria se realiza sólo si así se solicita mediante la llamada a la función calloc. En otras palabras, cuando se reserva una porción de memoria mediante una llamada a malloc, esa porción es visible al programa con su contenido intacto. Es decir, que no se inicializa a ningún valor en particular. Lo más probable es que contenga restos de la información que se ha almacenado previamente.

Este comportamiento está pensado para poder obtener el mayor rendimiento de un programa. A menudo hay porciones de memoria que se solicitan, pero que a continuación se inicializan desde el propio programa a unos valores concretos. En este caso, si malloc inicializase la memoria, se haría esta tarea dos veces, con la consiguiente pérdida de tiempo. Por este motivo, sólo la función calloc realiza esta tarea. Como ejemplo, en la siguiente porción de código se intenta mostrar por pantalla como cadena de texto la basura que haya quedado almacenada en esa zona de memoria.

```
char *string;
string = (char *)malloc(100);
printf("%s\n", string);
```

### 6.10.4. Sobre-escritura de memoria dinámica

El manejo de arrays en C se hace sin comprobación alguna de que el índice utilizado para acceder a un elemento esté en los límites correctos. De este comportamiento se deriva que los punteros y los arrays son, a efectos del compilador, lo mismo, una dirección de memoria sobre la que se puede utilizar entre corchetes un índice para acceder a un elemento. Este comportamiento se mantiene para el caso de la memoria dinámica, es decir, si se reserva espacio en memoria dinámica para un puntero o un array y en su acceso se rebasa el tamaño de su porción de memoria, la ejecución continua sin ningún tipo de comprobación. El siguiente fragmento de código ilustra esta situación.

```
struct point_info
{
  int x;
  int y;
};

struct point_info *points;
```

```
points = (struct point_info *)malloc(100 * sizeof(struct point_info));
points[356].x = 10;
points[356].y = 20;
```

Como el índice que se utiliza para el acceso de las dos últimas líneas está fuera de los límites, se está accediendo a una porción del heap que contiene otros datos que pueden estar reservados o no. El efecto es imprevisible, pero el programa no realiza ninguna comprobación.

### 6.10.5. Acceso a memoria con un puntero corrupto

Cuando la memoria dinámica se reserva, el sistema marca esa porción como ocupada y por tanto sus datos se mantienen. Sin embargo, cuando la memoria se libera, su contenido ya no está garantizado, y depende del uso interno que de ella haga el sistema operativo.

Esta observación es importante porque la función `free` que recibe como parámetro un puntero, libera su contenido pero no evita que se pueda volver a acceder a él, en lo que se conoce como un problema de acceso a un “puntero corrupto”. La siguiente porción de código muestra un ejemplo de este problema.

```
struct list_element
{
  int;
  struct list_element *next;
};

void destroy(struct list_element *l)
{
    while (l != NULL)
    {
      free(l);
      l = l->next;
    }
    return;
}
```

La línea que avanza por la cadena de punteros `l = l ->next` accede a una porción de memoria apuntada por `l` que ha sido liberada previamente, por lo tanto, su contenido no está garantizado y puede que el campo `next` ya no contenga el dato esperado. Una forma de resolver este problema es copiar ese puntero en un lugar en el que no pase esto, por ejemplo, una variable local.

## 6.11. Problemas sobre fugas de memoria.

Los siguientes problemas asumen que conoces las llamadas al sistema para reservar y liberar memoria dinámica.

1. Considera el siguiente programa en C:

```
#define SIZE 100
int main(int argc, char *argv[])
{
  int i, j;
  int *ptr;

  for (i = 0; i < SIZE; i++)
  {
    ptr = (int *)malloc(SIZE * sizeof(int));
    for (j = 0; j < SIZE; j++)
    {
      ptr[j] = j;
    }
  }
  ptr = NULL;
```

```
}
```

¿Cuántas fugas de memoria tiene este fragmento de código? ¿Cuánta memoria en total se fuga?

2. Un programador trabaja con una lista encadenada de palabras. Cada elemento de la lista tiene un puntero a una cadena y el puntero a la siguiente. La lista se representa por el puntero a su primer elemento. El código de la función de borrado de una lista es el siguiente:

```
struct list_node
{
  char *word;
  struct list_node *next;
};

void delete(struct list_node *l)
{
  struct list_node *tmp;
  while (l->next != NULL)
  {
    tmp = l;
    free(l->word);
    l = l->next;
    free(l);
  }
}
```

Tras utilizar un programa de detección de fugas, el informe dice que hay bloques de memoria sin liberar. El programador ha detectado que el problema está en esta función, pero ¿dónde exactamente? (Pista: Dibuja primero un ejemplo de lista encadenada y de cómo esta función realiza la liberación de memoria)

¿Qué sucede si se intenta liberar una lista vacía? Es decir, que `l` es `NULL`.

3. Un programa necesita manipular un árbol binario (todo nodo tiene dos y sólo dos hijos). La estructura que se utiliza para esto es la siguiente:

```
struct tree_node
{
  int value;
  struct tree_node *left;
  struct tree_node *right;
};
```

Un nodo sin hijos se representa con los dos punteros a NULL.

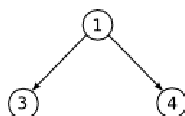
1. Escribe una función que dado un entero devuelve un nodo terminal (sin hijos) con ese valor almacenado. La función debe crear ese nodo. El prototipo de la función es el siguiente:

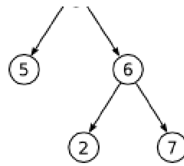
```
struct tree_node *node(int v);
```

2. Escribe una función que dado un entero y dos punteros a nodos (ya creados), devuelve un nuevo nodo con el entero almacenado como valor y los dos punteros como hijos. El prototipo de la función es el siguiente:

```
struct tree_node *create_node(int v, struct tree_node *left, struct tree_node *right);
```

3. Escribe ahora la función `main` que utilice las dos anteriores para crear el árbol que se muestra en la siguiente figura:






4. Escribe una función que dado un puntero a un nodo, libera el espacio que ocupa. No se realiza operación alguna sobre los punteros de los hijos, se asume que ya se han liberado. El prototipo de la función es:

```
void delete(struct tree_node *t);
```

5. Utilizando la función anterior escribe una función que, dado un puntero a un nodo, libere ***todos los nodos de ese árbol*** (cuidado, esta función no es trivial, si no logras escribirla, consúltanos). El prototipo de la función es:

```
void delete_tree(struct tree_node *t);
```

4. Una aplicación que ejecuta en tu móvil tiene una estructura creada basada en las siguientes definiciones:

```
struct vendor
{
  char *vendor_name;
  int installation_date;
  int price;
};
struct program
{
  char *path;
  struct stats
  {
    int uses;
    int last_use;
  } statistics;
  struct vendor *vendor_info;
};
struct program *installed_applications;
int number_of_installed_applications;
```

El programa ha reservado dinámicamente (a través de `malloc`) un array para tantas estructuras de tipo `program` como contiene la variable `number_of_installed_applications` y ha almacenado ese puntero en `installed_applications`. Cada elemento del tipo `struct program` tiene sus campos `path` y `vendor_info` reservados de forma dinámica. Así mismo cada elemento del tipo `struct vendor` tiene su campo `vendorname` reservado de forma dinámica

Escribe una función que recibe un puntero de tipo `struct program *` a una tabla como la que hay almacenada en `installed_applications` y su tamaño como un entero y libera todo el espacio que ocupa la estructura de datos. El prototipo de esta función es:

```
void delete(struct program *table, int size);
```