

Manejo de Punteros y Memoria Dinámica



Every computer, at the unreachable memory address 0x-1, stores a secret. I found it, and it is that all humans ar-- SEGMENTATION FAULT.

Introducción

Antes de meternos a fondo en las cuestiones de la manipulación de la memoria dinámica en C, definiremos algunos conceptos básicos.

Un **proceso** es un programa en ejecución. Cuando nosotros abrimos el binario de nuestro programa, se crea la imagen del proceso y comienza a ejecutarse. Podría decirse que un proceso está dividido en cuatro segmentos^[1] básicos:

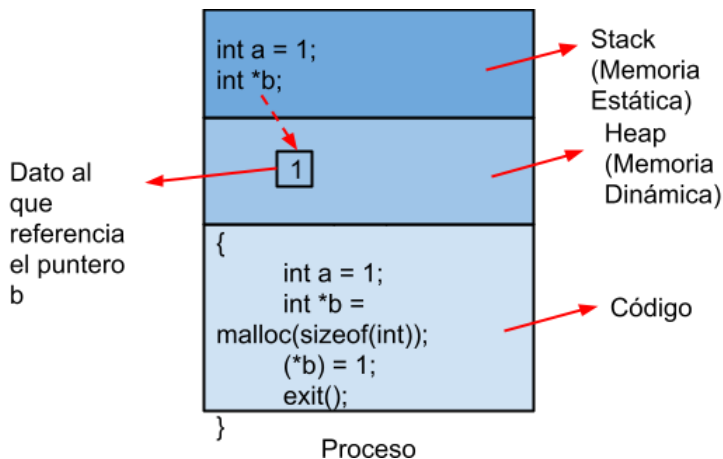
- Código
- Datos (globales)
- Heap
- Stack

La **memoria estática** es memoria que se reserva al declarar variables de cualquier tipo de dato: int, float, char, double, estructuras como así también los punteros^[2] a tipos de datos (por ejemplo: int*, char*, double*), y se aloja en el **stack** del proceso. En éste caso, el programador no podrá modificar el espacio en memoria que ocupan ni tampoco tendrá que encargarse de liberarla.

Al producirse una llamada a una función, se almacena en el stack del proceso la dirección a la que debe retornar la ejecución tras finalizar la llamada, y otros datos adicionales^[3].

La **memoria dinámica** es memoria que se reserva en *tiempo de ejecución* y se aloja en el **heap** del proceso (los datos apuntados por los punteros). En C, el programador deberá reservar dicha memoria para su uso y también tendrá la responsabilidad de liberarla cuando no la utilice más.

Una diferencia importante es que **el tamaño de la memoria dinámica se puede ir modificando durante la ejecución del programa**. ¿Qué quiere decir ésto? Que, por ejemplo, podrías ir agrandando/achicando una determinada estructura (por ejemplo, un array) a medida que lo necesitás.



Algunas ventajas que ofrece la memoria dinámica frente a la memoria estática es que podemos reservar espacio para variables de tamaño no conocido hasta el momento de la ejecución (por ejemplo, para listas o arrays de tamaños variables), o bloques de memoria que, mientras mantengamos alguna referencia a él, pueden sobrevivir al bloque de código que lo creó. Sin embargo, como una vez dijo el tío Ben a Spiderman: "Todo poder conlleva una gran responsabilidad" ^[4].

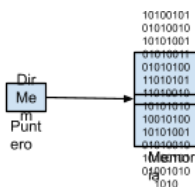
Todos los datos tienen un tiempo de vida, nada persiste para siempre. En C, hay tres tipos de duración:

- **Estática:** son aquellas variables que se crean una única vez junto con la creación del proceso y se destruyen junto con la destrucción del mismo, son únicas y generalmente pueden ser utilizadas desde cualquier parte del programa. Para generar una variable estática se la puede declarar por fuera de la función principal (arriba del `main()` por ejemplo), o bien usando el calificador `static`.
- **Automática:** son aquellas variables locales que no son declaradas con el especificador `static`. Se crean al entrar al bloque en el que fueron declaradas y se destruyen al salir de ese bloque. Por ejemplo, el tiempo de vida de las variables internas de una función es lo que tome ejecutarla.
- **Asignada:** es la memoria que se reserva de forma dinámica (en el heap) y que se explicó más arriba.

Punteros

¿Qué es un puntero?

Un puntero es la dirección de algún dato en memoria. Un puntero **NO** es el dato en sí mismo, sino su posición en la memoria. También se lo conoce como *referencia* a memoria.

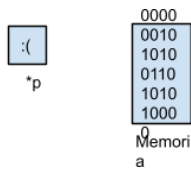


¿Cómo declaramos un puntero?

Supongamos que queremos declarar un puntero a un tipo de datos `int`. En C ésto se escribe de la siguiente manera:

```
int *p;
```

Bien, ya declaramos el puntero. Sin embargo, no está inicializado y apunta a basura.



Para que nuestro puntero no esté triste, ¡vamos a darle algo para que apunte!

¿Cómo inicializamos un puntero? (Malloc)

La memoria se puede reservar o liberar **dinámicamente**, es decir, según necesitemos. Para esto hay varias funciones estándar, que se encuentran en la biblioteca ^[5] estándar de C, en el encabezado `stdlib.h`.

Una de ellas es la función `malloc` ^[6], la cual sirve para solicitar un bloque de memoria del tamaño indicado por parámetro. Devuelve un puntero a la zona de memoria concedida:

```
void* malloc (unsigned numero_de_bytes);
```

El tamaño se especifica en bytes. `malloc` nos garantiza que la zona de memoria concedida no esté ocupada por ninguna otra variable. *Grosso, ¿no?* Eso sí, si `malloc` no puede otorgarnos dicha zona de memoria, devuelve un puntero a `NULL`. Por ende, cada vez que hacemos una llamada a `malloc` deberíamos chequear que no devuelve un puntero nulo.

Como dijimos antes, `malloc` devuelve un puntero a la zona de memoria concedida. Sin embargo, éste puntero devuelto no sabe a qué tipo de datos apunta (`void*` significa esto) ^[7].

Antes de llamar a `malloc` tenemos que saber cuántos bytes queremos reservar en memoria. Como nuestro tipo de datos va a ser un `int`, vamos a reservar 4 bytes. Entonces, la llamada debería quedar así:

```
malloc (4);
```

Entonces, ¿por cada tipo de puntero que tenga que declarar me tengo que acordar los bytes que ocupa? No necesariamente, podemos recurrir al operador `sizeof`. El operador `sizeof` recibe por parámetro un tipo de dato y devuelve el tamaño en bytes de éste. También podemos pasarle una variable y él se encargará de chequear el tipo y hacer el cálculo.

Entonces, nuestra llamada a `malloc` quedaría así:

```
malloc(sizeof(int));
```

Esta opción es altamente preferible, no sólo por su legibilidad y correctitud, sino por su declaratividad: yo no quiero reservar 4 bytes, sino los bytes necesarios para guardar un `int`. Cuántos sean esos bytes, no me interesa: es problema de la implementación, y hay alguien que lo resuelve por mí. Tal vez éste ejemplo suene muy trivial, pero si nosotros tenemos que reservar en memoria un espacio equivalente al tamaño de una determinada estructura (`struct`) tendríamos que saber cuántos bytes requiere cada

tipo de datos que la estructura contenga y sumarlos. Para ésto podemos recurrir a `sizeof(struct miEstructura)` para saber la cantidad de bytes que ocupa la misma.

Bien, ya pedimos los bytes, ahora sólo queda asignarlo a nuestro puntero:

```
int *p = malloc(sizeof(int));
```

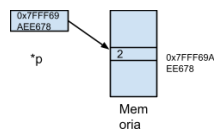
En resumen, nuestro código inicial quedaría de ésta manera:

```
1  #include <stdlib.h>
2
3  int main(void){
4      int *p = malloc(sizeof(int));
5      return 0;
6  }
```

Sin embargo, ¡falta algo más! Si bien reservamos el espacio en memoria al que va a apuntar nuestro puntero, ¿qué dato contiene ese espacio? Lo que contiene es... ¡basura!

Para asignarle un valor, usamos el operador `*` (asterisco), el operador de *desreferencia*^[8] de C:

```
1  #include <stdlib.h>
2  int main(void){
3      int *p = malloc(sizeof(int));
4      *p = 2;
5      return 0;
6  }
```



Al hacer `*p` estamos diciendo "al contenido de *p*" o "a lo apuntado por *p*". Si hiciéramos `p = 2` estaríamos modificando al puntero *p*, y no al valor apuntado por el puntero *p*.

Entonces, para inicializar un puntero, tenemos que realizar dos pasos:

- Reservar el espacio en memoria.
- Darle un valor inicial al dato contenido en ése espacio en memoria.

Operaciones de punteros

Dirección de una variable (&)

Un operador muy importante es `&` (ampersand) el cual nos devuelve la dirección en memoria de su parámetro.

Podés comprobar ésto ejecutando el siguiente código:

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(void){
5      int *p = malloc(sizeof(int));
6      *p = 1;
7      printf("p = %d\n", p);
8      printf("&p = %p\n", &p);
9      return 0;
10 }

```

- `%d` y `%p` son especificadores de formato.
- `%d` está diciendo que en esa parte de la línea va a imprimir un valor entero.
- `%p` está diciendo que en esa parte de la línea va a imprimir la dirección en memoria del dato, por ejemplo `0x7fff78c088d8`.

Ojo! El operador `&` se puede usar con todo tipo de variables ya que todo está contenido en la memoria.

Desreferencia (*)

Veamos el siguiente código:

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(void){
5      int i = 1;
6      int *p;
7
8      printf("Antes i vale: %d\n", i);
9
10     p = &i; //p apunta a i
11     *p = 2; //se le asigna a donde este apuntando p (i) el valor 2
12
13     printf("Ahora i vale: %d y el contenido de p vale: %d\n", i, *p);
14     return 0;
15 }

```

Ejecutamos el código y veremos en consola lo siguiente:

Antes i vale 1.

Ahora i vale 2 y el contenido de p vale 2.

- Declaramos una variable `i` de tipo `int` con el valor 1.

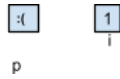
```
int i = 1;
```

1
i

- Declaramos un puntero `p` a un tipo de dato `int`.

```
int *p;
```

c



- Imprimimos por pantalla el valor de `i` , mostrará 1.

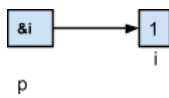
```
printf("Antes i vale: %d\n", i);
```

c

- Le asignamos al puntero `p` la dirección de memoria de `i`.

```
p = &i;
```

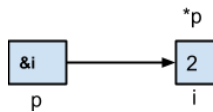
c



- Le asignamos a la porción de memoria a la que apunta `p` (con el paso anterior hicimos que apunte a `i`) el valor 2. A esto se lo conoce como **desreferenciar**.

```
1 | *p = 2;
```

c



- Imprimimos los valores de ambos. Ambos valen 2.

```
1 | printf("Ahora i vale: %d y el contenido de p vale: %d\n", i, *p);
```

c

¡Ambos tienen el mismo valor! Y eso es porque el puntero `p` está apuntando a la misma porción de memoria que `i` tiene asignada. Por ende, se puede manipular dicho dato desde el puntero `p` .

Con el operador `*` (asterisco) podemos acceder al contenido al que apunta nuestro puntero.

Lo que acabamos de hacer fue manipular una porción de memoria ajena a través de un puntero.

¿Y para qué me sirve? Veamos el siguiente ejemplo:

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void sumarUno(int unaVariable){
5      unaVariable = unaVariable + 1;
6      printf("Dentro de la funcion, i vale: %d\n", unaVariable);
7  }
8
9  int main(void){
10     int i = 1;
11
12     printf("Antes de ejecutar la funcion, i vale: %d\n", i);
13     sumarUno(i);
14     printf("Despues de ejecutar la funcion, i vale: %d\n", i);
15
16     return 0;
17 }

```

c

Tenemos una función `sumarUno` que le suma 1 a la variable que le haya sido pasada por parámetro y la imprime en pantalla.

Ejecutamos el código y veremos en consola lo siguiente:

Antes de ejecutar la funcion, i vale: 1Dentro de la funcion, i vale: 2Despues de ejecutar la funcion, i vale: 1

En C, cuando nosotros llamamos a una función y le pasamos parámetros, los valores que recibe son copiados en una dirección de memoria distinta y son operados desde allí hasta que el bloque de código termina de ejecutarse. A esto se le conoce como **parámetros por valor**. En otras palabras, no podremos modificar variables desde funciones para que persistan luego de que finalice la ejecución de la función.

Si nosotros quisiéramos que el cambio persista luego de la ejecución de la función, tendríamos que decirle, a nuestra función, que lo que va a recibir como parámetro es una **referencia a memoria**, también conocida como... ¡puntero!

Entonces, en vez de `void sumarUno(int unaVariable)` vamos a tipear `void sumarUno(int *unaVariable)`. Donde antes teníamos `unaVariable = unaVariable + 1;` vamos a tener que poner `(*unaVariable) = (*unaVariable) + 1;`. ¿Por qué? **Porque lo que deseamos modificar es el dato al que apunta.** Ésto es lo que vimos antes como *desreferenciar*.

Sin embargo, si corremos esto como está, no va a funcionar porque ahora cuando llamamos a la función tenemos que pasarle un puntero o *una dirección de memoria*. Como vimos antes, podemos utilizar el operador `&` en éste caso.

Entonces, en vez de llamar a la función de ésta manera: `sumarUno(i);` lo hacemos de esta manera: `sumarUno(&i);`.

Así, nuestro código quedará de la siguiente forma:

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void sumarUno(int *unaVariable){
5      (*unaVariable) = (*unaVariable) + 1;
6      printf("Dentro de la funcion, i vale: %d\n", *unaVariable);
7  }
8
9  int main(void){
10     int i = 1;
11
12     printf("Antes de ejecutar la funcion, i vale: %d\n", i);
13     sumarUno(&i);
14     printf("Despues de ejecutar la funcion, i vale: %d\n", i);
15
16     return 0;
17 }

```

Y al correrlo, en consola leeremos lo siguiente:

Antes de ejecutar la funcion, i vale: 1Dentro de la funcion, i vale: 2Despues de ejecutar la funcion, i vale: 2

Desreferencia en estructuras (->)

Este operador ofrece una sintaxis alternativa al acceder a los datos de un puntero hacia un tipo de datos estructura.

Ejemplo: Supongamos que tenemos un tipo `t_persona` definido de la siguiente manera:

```

1  typedef struct
2  {
3      char[20] nombre;
4      char[20] apellido;
5      int edad;
6  } t_persona;

```

Queremos crear un puntero hacia una estructura `t_persona` que contenga como nombre "Esteban", apellido "Trabajos" y edad 56.

Si quisiese acceder al nombre de un `t_persona`, tendría primero que acceder a la estructura y luego a sus datos. Vamos a hacer ésto utilizando el operador `*`.

```

1  t_persona *p = malloc(sizeof(t_persona));
2  (*p).nombre = "Esteban";
3  (*p).apellido = "Trabajos";
4  (*p).edad = 56;

```

Sin embargo, C ofrece una alternativa a la sintaxis `(*p)` mediante el operador `->` ("flechita"). En éste caso para que quede más "limpio" el código.


```

1  t_persona *p = malloc(sizeof(t_persona));
2  p->nombre = "Esteban";
3  p->apellido = "Trabajos";
4  p->edad = 56;

```

El operador `->` [9] se puede utilizar si tenemos estructuras anidadas. Supongamos que al campo `t_persona` le agregamos un campo más para que haga referencia a un hijo:

```

1  typedef struct
2  {
3      char[20] nombre;
4      char[20] apellido;
5      int edad;
6      t_persona* hijo;
7  } t_persona;

```

Si quisiéramos acceder al nombre del hijo, bastaría con tipear: `p->hijo->nombre` .

Funciones que retornan punteros

Como vimos antes, el pasarle un puntero como argumento a una función resulta de mucha utilidad si queremos cambiar el contenido de esa variable dentro de la función, o si no queremos copiar toda la información devuelta, pero... ¿Qué hay de las funciones que devuelven punteros? Tomemos este ejemplo:

```

1  char* copiar(char* palabra){
2      char* tmp = malloc(sizeof(char) * strlen(palabra) + 1);
3      memcpy(tmp, palabra, strlen(palabra));
4      tmp[strlen(palabra)] = '\0';
5      return tmp;
6  }

```

Esta función trivial nos va a ayudar a comprender la utilidad de este concepto. Desglosémosla:

- La función recibe un string, y devuelve un puntero a una posición en memoria donde se encuentra una copia de esta palabra.
- Primero aloca un espacio consecutivo del tamaño de la palabra (+1 porque todos los strings terminan con un `\0`)

```
malloc(sizeof(char) * strlen(palabra) + 1);
```

- y asocia este espacio a una variable "tmp"

```
char* tmp =
```

- Luego copia la palabra por argumento al segmento reservado

```
memcpy(tmp, palabra, strlen(palabra));[11]
```

- Inserta el `\0` faltante en la última posición, usando el conjunto de bytes como si fuera un array.

```
tmp[strlen(palabra)] = '\0';[12]
```

- y retorna el puntero al nuevo sector de memoria con la copia de la palabra

```
return tmp;
```

Es notable mencionar que la función es la que crea el segmento en memoria, y que si se llamara `n` veces, crearía `n segmentos de memoria`, cada uno con una copia de la palabra, por lo tanto el que reciba el segmento de memoria tiene la responsabilidad de liberarlo cuando no lo necesite más.

"Tu programa chorea memoria!" (A.K.A Memory Leaks)

Al hacer el `malloc` [10], uno reserva un segmento continuo de memoria de el tamaño que se le indica, y un poquito más. Este extra contiene información sobre el segmento, como el tamaño, y otra metadata que el sistema operativo crea conveniente.

Cuando nuestro proceso finaliza, el SO se encarga de liberar toda la memoria asociada a nuestro proceso. Sin embargo, durante la ejecución de nuestro proceso, es responsabilidad nuestra liberar la memoria asignada dinámicamente para poder reutilizarla. De no hacer ésto, nuestros procesos estarían ocupando más memoria de la que requieren realmente.

Seguramente dirás: "yo tengo 4GB de memoria, ¿qué me importa si consumo un toque más de memoria o no?, de última me compro 4GB más que están 200p man". En parte tenés razón y en parte no, ¿por qué?, porque si nuestro proceso no finaliza nunca (es decir, es un while gigante) y se encarga de hacer mucho procesamiento, creeme que vas a empezar a ocupar memoria a lo loco a tal punto que en un momento se va a ver afectada la performance del proceso y te vas a quedar sin memoria. Y cuando te quedás sin memoria, el SO prioriza su vida y mata el proceso que la está pidiendo.

Generalmente, las fugas de memoria o memory leaks se dan cuando perdemos la referencia de un puntero en algún punto. Retomemos el ejemplo anterior:

```
1 char* copiar(char* palabra){
2     char* tmp = malloc(sizeof(char) * strlen(palabra) + 1);
3     memcpy(tmp, palabra, strlen(palabra));
4     tmp[strlen(palabra)] = '\0';
5     return tmp;
6 }
```

Si nosotros no devolviésemos el puntero al que le estamos asignando un espacio en memoria dinámica, estaríamos perdiendo la referencia al bloque ese y, por ende, pasaría un memory leak (¿cómo liberaríamos el bloque? ¿cómo adivinamos cuál de TODAS las posiciones de memoria que hay en nuestra computadora le corresponde a nuestro bloquecito?). Esto implica que ese espacio que reservamos nunca más vamos a poder volver a utilizarlo durante la ejecución del proceso.

Y si no me creéis, probá esto:

```
1  #include <string.h>
2  #include <stdlib.h>
3
4  void copiar(char* palabra){
5      char* tmp = malloc(sizeof(char) * strlen(palabra) + 1);
6      memcpy(tmp, palabra, strlen(palabra));
7      tmp[strlen(palabra)] = '\0';
8  }
9
10 void main(void){
11     while(1){
12         copiar("no liberar la memoria dinamica que reservamos cuando ya no necesitamos de
13     }
14 }
```

Por suerte, existe una solución fácil, el glorioso `free` :

```
free(unPuntero);
```

Esta simple función se encarga de buscar el segmento que habíamos reservado, y marcarlo como libre para que otro o nosotros lo volvamos a usar (notar que no limpia la información que había en el segmento).

Dos cosas a tener en cuenta: Si tenemos una función que reserve memoria, podemos hacer `free` fuera de ella, siempre y cuando tengamos alguna referencia al espacio en memoria. Por ejemplo:

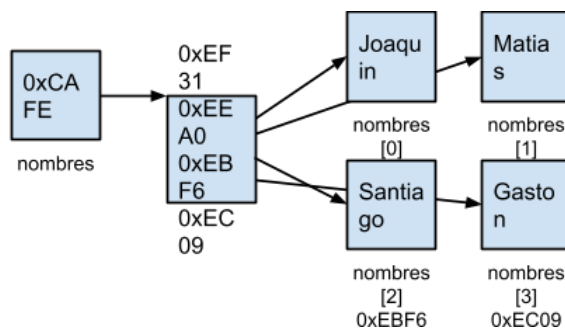
```
1  #include <stdlib.h>
2  char* reservarMemoria(int n){
3      return malloc(n*sizeof(char));
4  }
5
6  int main(void){
7      char* array;
8      array = reservarMemoria(3); //Reserva 3 char's consecutivos
9      free(array);
10 }
```

Si tenemos estructuras con punteros, el orden de liberación es muy importante. Consideremos el ejemplo previamente visto:

```

1 char* copiar(char* palabra){
2     char* tmp = malloc(sizeof(char) * strlen(palabra) + 1);
3     memcpy(tmp, palabra, strlen(palabra));
4     tmp[strlen(palabra)] = '\0';
5     return tmp;
6 }
7
8 int main(void){
9     char** nombres;
10    //Grabo espacio para 4 punteros a nombres
11    nombres = malloc(sizeof(char*) * 4);
12
13    //Grabo cada una de las palabras
14    nombres[0] = copiar("Joaquin"); //7 + 1 chars
15    nombres[1] = copiar("Matias"); //6 + 1 chars
16    nombres[2] = copiar("Santiago"); //8 + 1 chars
17    nombres[3] = copiar("Gaston"); //6 + 1 chars
18
19    free(nombres);
20 }

```



El programa reserva espacio para 4 punteros, y despues carga cada uno de esos punteros con los nombres "Joaquin", "Matias", "Santiago" y "Gastón". Si yo libero la variable nombres, entonces no tengo forma de liberar los restantes 35 chars que reservé para las letras, por lo que perdí (leak'ie) 35 bytes de memoria. La forma correcta de terminar el programa sería:

```

1 int i;
2 for(i=0; i<4; ++i)
3     free(nombres[i]);
4 free(nombres);

```

Hacerlo al revés, primero liberar `nombres` y luego intentar liberar los `nombres[i]` tampoco funcionaría.

La regla de oro de la memoria dinámica es que tiene que existir un `free` por cada `malloc`.

Aclaración: si bien `free()` no borra los datos y el puntero sigue estando, lo que hace internamente es dejar libre (valga la redundancia) ése bloque de memoria para el resto de los procesos. Éso quiere decir que otro proceso puede pedir reservar un espacio de memoria y casualmente se le asigne ese. Nosotros, conservando el puntero, podríamos seguir manipulando ése bloque pero sería arriesgarnos a que se den inconsistencias ya que ahora no hay un proceso sólo tocando dicho espacio, sino dos. Por ende, pensar en utilizar en dichas condiciones sería pésimo.

La importante de que reservemos memoria es que nos aseguramos de que ningún proceso más acceda a ése espacio (salvo que le demos permiso, pero éso es otra historia).

Aritmética de Punteros

El lenguaje nos permite sumar o restar cantidades enteras al puntero para que apunte a direcciones de memoria distintas, a ésto se lo conoce como aritmética de punteros. Sirve mucho a la hora de hacer manejos de memoria y es una sintaxis alternativa a la del acceso de elementos de un array.

Con un puntero, las expresiones $p+1$, $p+2$, $p+3$, $p+n$ tienen sentido. La expresión $p+n$ es un puntero que apunta a la dirección de p sumándole n veces el espacio ocupado por un elemento del tipo al que apunta. Es decir, la expresión sumada NO es el número de bytes que se suman a la dirección, es el número de elementos del tipo al que apunta el puntero.

Utilicemos el ejemplo anterior.

```
1 char** nombres;
2 //Grabo espacio para 4 punteros a nombres
3 nombres = malloc(sizeof(char*) * 4);
4
5 //Grabo cada una de las palabras
6 nombres[0] = copiar("Joaquin");
```

¡Ambos tienen el mismo valor!. Y eso es porque el puntero p está apuntando a la misma porción de memoria que i tiene asignada. Por ende, se puede manipular dicho dato desde el puntero p . Entonces, si quisiéramos acceder al segundo elemento del array de strings (`nombres[1]`), la sintaxis equivalente sería `*(nombres + 1)` . Teniendo en cuenta ésto, en un array, `*nombres` sería el primer elemento del array, entonces quedaría:

```
1 *(nombres+1) = copiar("Matias"); //6 + 1 chars
2 *(nombres+2) = copiar("Santiago"); //8 + 1 chars
3 *(nombres+3) = copiar("Gaston"); //6 + 1 chars
```

Otros alloc

`calloc(n, bytes)` : reserva memoria para un array de n elementos que ocupan un tamaño de x bytes cada uno, además inicializa los bytes con un `\0` . Por ejemplo, supongamos que queremos reservar memoria para un array de 5 enteros, entonces:

```
int *arrayEnteros = calloc(5, sizeof(int))
```

El equivalente, con la función `malloc` , sería:

```
int *arrayEnteros = malloc(5*sizeof(int))
```

`realloc(*unPuntero, bytes)` : cambia el tamaño del bloque de memoria apuntado por `unPuntero` a uno de x bytes. Devuelve un puntero al bloque con el tamaño indicado. Es importante saber que los datos no

son alterados y se guardan en el nuevo bloque siempre y cuando le hayamos reasignado un tamaño mayor o igual al del bloque anterior. Los bytes agregados (es decir, si el tamaño total que le pasamos por parámetro es mayor al tamaño del bloque apuntado por unPuntero) no están inicializados. Debemos tener cuidado en los parámetros que le pasemos porque: Si unPuntero es `NULL`, la función se comporta como un `malloc(bytes)`. Si unPuntero no es `NULL` y `bytes = 0`, la función se comporta como un `free(unPuntero)`.

El problema del tipo de dato "int"

Sin embargo, con el tipo de datos `int` hay un tema muy importante a considerar. Dependiendo de la arquitectura, sistema operativo, y del compilador en sí mismo, el tipo de dato `int` va a tener un tamaño u otro. Generalmente `int` tiene un tamaño equivalente al tamaño de la palabra del procesador. Por ende, si estamos en una arquitectura de 32 bits, la palabra tendrá un tamaño de 32 bits, `int` tendrá un tamaño de 32 bits o 4 bytes (8 bits = 1 byte). Si estamos en una arquitectura de 64 bits, la palabra tendrá un tamaño de 64 bits, `int` tendrá un tamaño de 64 bits o 8 bytes.

Entonces, si nosotros reservamos (malloc) 4 bytes para un tipo `int` en una máquina con un procesador de 32 bits no vamos a tener ningún problema pero si lo hacemos en una de 64 bits va a volar todo por los aires.

Para solucionar este problema podemos considerar dos opciones:

- Recurrir a un tipo de datos que no dependa de la arquitectura de nuestro procesador, es decir, que tengan un tamaño fijo lo corramos donde lo corramos. Por ejemplo, `int32_t` o `int64_t` ^[1].
- Utilizar el operador el `sizeof()` para que se acople a la arquitectura en la que esté corriendo. Ej: `malloc(sizeof int);`

Normalmente vamos a optar por usar el operador `sizeof`, los tipos de dato entero de tamaño fijo los vamos a dejar para aquellos momentos en que no podemos dejar el tamaño del entero al criterio del sistema operativo. Un uso común para estos tipos de datos es cuando queremos realizar un intercambio de datos entre dos computadoras diferentes, pero eso lo veremos en el proximo capitulo.

EXTRA: Punteros a Funciones

Un puntero a una función es una variable que almacena la dirección en memoria de una función que luego podrá ser invocada desde dicho puntero. Los punteros a funciones se declaran de una manera similar a los punteros que conocemos hasta ahora, con la diferencia de que hay que aclarar el tipo de valor que retorna y los tipos de datos de los parámetros que acepta. Al fin y al cabo, ¡como una función!. Por ejemplo:

```
void (*f)(int,int);
```

c

Con ésto estamos declarando un puntero `f` a una función que recibirá por parámetro dos enteros (`int,int`) y no retorna ningún valor.

En el siguiente código veremos cómo un posible uso:

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void imprimirValor(int x)
5  {
6      printf("%d\n", x);
7  }
8
9  int main()
10 {
11     void (*punteroAFuncion)(int);
12     punteroAFuncion = &imprimirValor;
13     punteroAFuncion(1);
14     return 0;
15 }

```

¿Qué sucede en el main?

- Declaramos un puntero a una función que recibe un entero y no retorna ningún valor.

```
void (*punteroAFuncion)(int);
```

- Le asignamos al puntero la dirección en memoria de la función imprimirValor.

```
punteroAFuncion = &imprimirValor;
```

- Llamamos a la función mediante el puntero, nótese que la sintaxis es idéntica a la llamada de una función cualquiera.

```
punteroAFuncion(1);
```

Los punteros a funciones nos pueden servir para, por ejemplo, reutilizar código en funciones genéricas. Para ser más claros, supongamos que tengo una lista de alumnos en la que de cada alumno se conoce su nombre, apellido, curso y notas de cada parcial.

```

1  typedef struct
2  {
3      char *nombre;
4      char *apellido;
5      int  curso;
6      int  notaPrimerParcial;
7      int  notaSegundoParcial;
8      t_alumno *siguiente;
9  } t_alumno;

```

El último campo se utiliza para anidar los distintos alumnos, es decir, para formar una lista. Una operación común en las listas es realizar un filtrado, es decir, a partir de una lista obtener otra que cumple con unas determinadas condiciones. Por ejemplo, todos los alumnos del curso 3020, todos los alumnos que aprobaron los dos parciales (es decir, `notaPrimerParcial >= 4 && notaSegundoParcial >= 4`) o todos los alumnos cuyo nombre empieza con la letra 'A'.

Nosotros, como programadores, decimos "ah, quiero obtener todos los alumnos cuyo nombre empiezan con la letra 'A', pero capaz mañana quiero saber quiénes son los que empiezan con la letra 'Z', entonces me adelanto y hago que le pase la letra con la que empieza el nombre por parámetro". Lo mismo con el filtrado por curso.

Entonces, programamos unas funciones cuya definición sería:

```
1  t_alumno *filtrarPorCurso(t_alumno *listaAlumnos, int curso);  
2  t_alumno *filtrarPorLetraInicialNombre(t_alumno *listaAlumnos, char c);
```

Una posible implementación de `filtrarPorCurso` sería:

```
1  t_alumno *filtrarPorCurso(t_alumno *listaAlumnos, int curso)  
2  {  
3      t_alumno *aux = listaAlumnos;  
4      t_alumno *listaFiltrada = crearListaAlumnos();  
5  
6      while(aux != NULL)  
7      {  
8          if (aux->curso == curso)  
9          {  
10             agregarALista(listaFiltrada, aux);  
11         }  
12         aux = aux->siguiente;  
13     }  
14  
15     return listaFiltrada;  
16 }
```

Una posible implementación de `filtrarPorLetraInicialNombre` sería:

```
1  t_alumno *filtrarPorLetraInicialNombre(t_alumno *listaAlumnos, char c)  
2  {  
3      t_alumno *aux = listaAlumnos;  
4      t_alumno *listaFiltrada = crearListaAlumnos();  
5  
6      while(aux != NULL)  
7      {  
8          if (aux->nombre[0] == c)  
9          {  
10             agregarALista(listaFiltrada, aux);  
11         }  
12         aux = aux->siguiente;  
13     }  
14  
15     return listaFiltrada;  
16 }
```

¿En qué cambia la función con respecto a la otra? Solamente en el criterio de filtro, el resto de la lógica - crear una lista aparte, recorrer la lista original y agregar a la lista nueva los que cumplan con el criterio - es exactamente la misma.

Repetir código es una mala práctica pero no es el objetivo de éste documento abordar esas cuestiones, sino de introducir una posible utilidad real que le puedas dar a los punteros a funciones.

En éste caso, podríamos hacer una función filtrar, genérica, que en base a un criterio determinado me devuelva la lista de aquellos que cumplen dicho criterio. El criterio será una función por parámetro.

Por lo visto anteriormente, el filtrado quedaría de la siguiente manera:

```
1  t_alumno *filtrarPorCriterio(t_alumno *listaAlumnos, bool (*criterio)(t_alumno*))  
2  {  
3      t_alumno *aux = listaAlumnos;  
4      t_alumno *listaFiltrada = crearListaAlumnos();  
5  
6      while(aux != NULL)  
7      {  
8          if (criterio(aux))  
9          {  
10             agregarALista(listaFiltrada, aux);  
11          }  
12          aux = aux->siguiente;  
13      }  
14  
15      return listaFiltrada;  
16  }
```



En negrita está marcado lo que cambió con respecto a la versión anterior. De ésta manera, sólo tendríamos que definir funciones criterio que respeten la definición dada en filtrarPorCriterio.

Y se usaría de la siguiente manera:

```
1  int main(int argc, char **argv) {  
2      char inicial = 'a';  
3      bool nombreEmpiezaCon(t_alumno *alumno){  
4          return alumno->nombre[0] == inicial;  
5      }  
6      //... inicializar lista ...  
7      t_alumno *filtrados = filtrarPorCriterio(alumnos, nombreEmpiezaCon);  
8  
9      return 0;  
10 }
```

Material recomendado

- [Binky Pointer C](#) (en inglés)
- [You will never ask about pointers again after watching this video](#) (en inglés)
- [Punteros en la guía Beej](#) (en inglés)
- [Tutorial de Valgrind](#) (para resolver memory leaks)

2. En el próximo capítulo se definirá el concepto de puntero. ↩
3. Por ejemplo, registros de la CPU al momento de la ejecución del proceso. ↩
4. En el comic original, el narrador es quien dice esta frase al final del primer volumen /NERD. ↩
5. La traducción correcta de library es biblioteca y NO "librería". ↩
6. `malloc()` es una abreviatura de "memory allocate". ↩
7. En C89 estábamos obligados, cada vez que llamábamos a malloc, a castear el puntero retornado por malloc al tipo de dato que estemos usando. En C99 esto ya no es necesario. ↩
8. **Dereference operator**  ↩
9. Ojo, son dos caracteres: - y > (menos y mayor). No busques un caracter que sea la flecha 😊 ↩
10. En realidad, cualquier operación de reserva de memoria dinámica, como calloc(), malloc() o realloc().
↩
11. Más info **acá** . ↩