Guía de C - avanzada

Arquitectura y Organización del Computador

Estructuras

Insanity consists of building major structures upon foundations which do not exist.

¿Qué es una estructura?

Una estructura es una **colección de variables** de distintos tipos, agrupadas bajo un mismo nombre.

Las estructuras son una forma de agrupar datos relacionados en un solo tipo de dato. Por ejemplo, si quisiéramos representar un **punto** en el espacio 3D, podríamos usar una estructura que contenga tres variables de tipo `float` (una para cada coordenada). Podríamos definir una estructura `punto` de la siguiente manera:

Snippet 1: struct

```c
#include <stdio.h>

struct punto {
    float x;
    float y;
    float z;
};
int main() {
    struct punto p1;
    p1.x = 1.0;
    p1.y = 2.0;
    p1.z = 3.0;

    printf("punto: (%f, %f, %f)\n", p1.x, p1.y, p1.z);
    return 0;
}
```

Podemos acceder a los miembros de una estructura usando el operador `.` (punto) y tratar a esa variable como una variable normal. En el ejemplo, definimos una estructura llamada `punto` que tiene tres campos: `x`, `y` y `z`, todos de tipo `float`. Luego, en la función `main`, creamos una variable de tipo `punto` llamada `p1` y le asignamos valores a sus campos. Podemos decir, que `struct punto` es un **tipo de dato compuesto** que agrupa tres variables de tipo `float`. Podemos definir estructuras anidadas, es decir, estructuras dentro de otras estructuras. Por ejemplo, si

quisiéramos representar un **jugador** en un mapa, podríamos definir una estructura `player` que contenga otra estructura `punto` para representar su posición en coordenadas cartesianas:

Snippet 2: struct

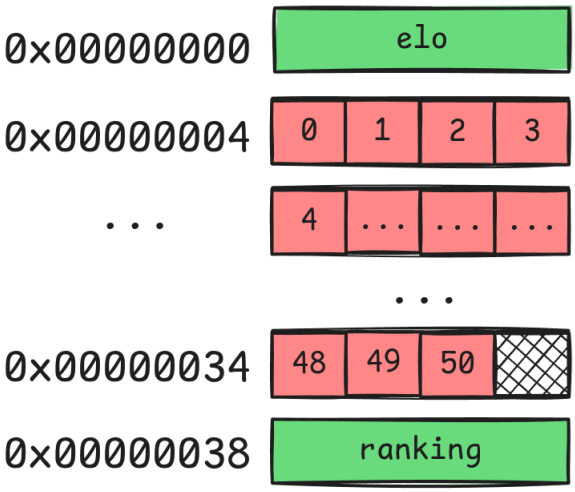
```c
#include <stdio.h>

struct punto {
    float x;
    float y;
};

struct player {
    char nombre[50];
    struct punto posicion;
};

int main() {
    struct player player1;
    player1.posicion.x = 1.0;
    player1.posicion.y = 2.0;

    printf("Posición del player1: (%f, %f)\n",
            player1.posicion.x,
            player1.posicion.y);
    return 0;
}
```

Como se ve en el ejemplo, nada impide que definamos arrays dentro de las estructuras. Por otro lado, podemos definir estructuras anónimas, es decir, estructuras que no tienen un nombre específico. Esto puede ser útil para crear estructuras temporales:

Snippet 3: struct

```c
#define NAME_LEN 50

struct {
  int elo;
  char name[NAME_LEN + 1];
  int ranking;
} player1;

int main() {
  player1.elo = 2800;
  player1.ranking = 1;

  return 0;
}
```

Esta estructura tiene tres campos: `elo` , `name` y `ranking`. En particular, podemos ver que ese struct no tiene nombre (se le suele llamar *anonymous struct*), solamente creamos una instancia de esa estructura anónima, llamada `player1`. Su layout en memoria sería algo así:



Struct en memoria

Notar el *padding* que existe entre el array y el campo `ranking`. Esto se debe a que el compilador alinea los datos en memoria para optimizar el acceso a ellos. En este caso, el compilador decidió alinear el campo `ranking` a un múltiplo de 4 bytes, lo que puede ser útil para mejorar la eficiencia del acceso a la memoria. Veremos en más detalle el tema de alineación y padding en la guia de Assembly.

Podemos inicializar una estructura al momento de declararla, usando llaves:

Snippet 4: Inicialización de struct

```c
#define NAME_LEN 50

struct player{
  int elo;
  char name[NAME_LEN + 1];
  int ranking;
};

struct player player1 = { 2800, "Magnus Carlsen", 1 },
struct player player2 = { 2700, "Fabiano Caruana", 2 };
```

En este momento, es conveniente mencionar el keyword `typedef`, que nos permite definir un nuevo tipo de dato basado en un tipo existente. El nuevo tipo también se conoce por el nombre de **type alias**. Esto es útil para simplificar la declaración de estructuras y hacer el código más legible.

Snippet 5: typedef

```c
typedef float real_t;
typedef int quantity_t;
typedef unsigned long int size_t; // en <stdint.h>
typedef uint32_t vaddr_t; // direccion virtual.
typedef uint32_t paddr_t; // direccion fisica.
```

Notar como primero viene el tipo original y luego el nuevo tipo.

En el siguiente fragmento definimos una estructura y le asignamos un alias `player_t` para poder usarlo más fácilmente en el resto del código. Además, notar el tipo de inicialización alternativa de `player3`:

Snippet 6: typedef de struct

```c
#define NAME_LEN 50

typedef struct {
  int elo;
  char name[NAME_LEN + 1];
  int ranking;
} player_t;

player_t player1 = { 2800, "Magnus Carlsen", 1 },
player_t player2 = { 2700, "Fabiano Caruana", 2 };
player_t player3 = { .name = "Hikaru Nakamura",
                     .ranking = 3,
                     .elo = 2600}; //forma alternativa
```

Por otro lado, la operación asignación `=` en `struct` produce una copia de la estructura completa.

Snippet 7: miembros

```c
player_t magnus = { 2800, "Magnus Carlsen", 1 },
player_t faustino;

printf("Elo: %d\n", magnus.elo);
printf("Name: %s\n", magnus.name);
printf("Ranking: %d\n", magnus.ranking);

magnus.elo = 2700;
magnus.ranking--;

faustino = magnus; // copia de estructura
```

Las estructuras pueden pasarse a funciones como argumentos y también pueden ser devueltas como valores de retorno.

Snippet 8: struct como argumento y valor de retorno

```c
player_t get_player(void) {
  player_t player = { 3000, "Bobby Fischer", 1 };
  return player;
}

void print_player(player_t player) {
  printf("Elo: %d\n", player.elo);
  printf("Name: %s\n", player.name);
  printf("Ranking: %d\n", player.ranking);
}
```

Notar que pasar estructuras grandes de esta manera puede ser ineficiente, ya que se copian todos los datos de la estructura. En su lugar, se suele usar punteros a estructuras, que son más eficientes. Lo veremos más adelante.

Nada impide que armemos un array de estructuras, como el siguiente:

Snippet 9: array de struct

```c
typedef struct {
  char* pais;
  int code;
} dials_code_t;

dials_code_t country_codes[] = {
  {"Argentina", 54},
  {"Brasil", 55},
  {"Chile", 56},
  {"Uruguay", 598}
};

printf("Código para Argentina: %d\n", country_codes[0].code);
```

La inicialización puede ser parcial también, como en el siguiente ejemplo:

Snippet 10: Inicialización de array de struct

```c
dials_code_t country_codes[100] = {
  [0] = {"Argentina", 54},
  [1] = {"Brasil", 55},
  [2] = {"Chile", 56},
  [3].pais = "Uruguay", [3].code = 598,
  // ... el resto se inicializa en 0
};
```

Ejercicio 1:

Definir una estructura `monstruo_t` que contenga los siguientes campos:

- `nombre` (string)
- `vida` (entero)
- `ataque` (double)
- `defensa` (double)

Luego, inicializar un array de monstruos y mostrar por pantalla el nombre y la vida de cada uno de ellos.

Ejercicio 2:

Definir una función `evolution` que reciba un `monstruo_t` y devuelva un nuevo monstruo con los mismos atributos, pero con el ataque y defensa aumentados en 10. Luego, usar esta función para evolucionar un monstruo y mostrar por pantalla sus atributos antes y después de la evolución.

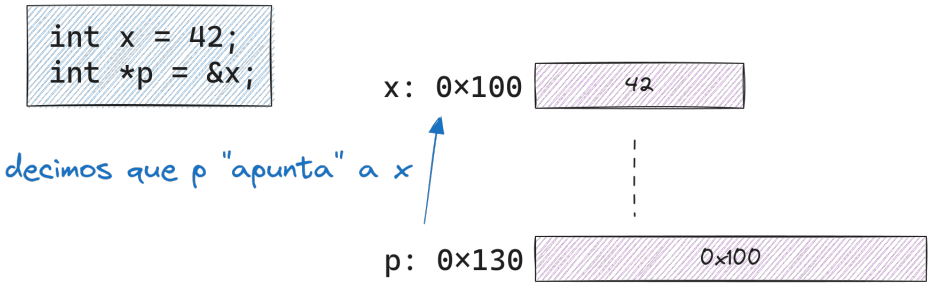
Punteros

Paradox is a pointer telling you to look beyond it. If paradoxes bother you, that betrays your deep desire for absolutes. The relativist treats a paradox merely as interesting, perhaps amusing or even, dreadful thought, educational.

¿Qué es un puntero?

Un puntero es una **variable que almacena una dirección de memoria**.

En la guía anterior mencionamos al pasar que uno de los tipos nativos de C es el **puntero**, tipado `int*`, `char*`, `void*`, etc.



Comencemos mirando el siguiente código:

Snippet 11:

```c
#include <stdio.h>

int main(){
    int x = 42;
    int *p = &x;

    printf("Direccion de x:  %p Valor: %d\n", (void*) &x, x);
    printf("Direccion de p: %p Valor: %p\n", (void*) &p, (void*) p);
    printf("Valor de lo que apunta p: %d\n", *p);
}
```

Ejercicio 3:

Sin correr el código, responder:

- ¿Cuál es la diferencia entre `x` y `p`? ¿Y entre `x` y `&x`? ¿Y entre `p` y `*p`?
- ¿Qué valores creen que se van a imprimir por terminal?

Ejercicio 4:

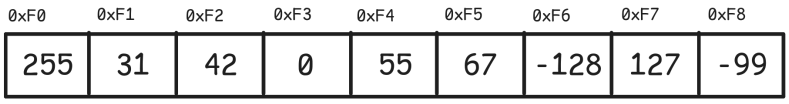
Compilar y ejecutar el código. ¿Qué valores se imprimieron? ¿Qué creen que significan?

Para responder estas preguntas, es **clave** que entendamos como se organiza la memoria en C.

Al final del día, la memoria no es más que una tira de **bits** (1/0). Pero usualmente queremos trabajar con variables con más valores que solo 0 y 1[1], por lo que hoy en día usamos a la memoria como si fuera una tira de **bytes**, que son grupos de **8 bits**.

Entonces, desde ahora, pensemos a la memoria como una **tira de bytes contiguos**. Vamos a poder acceder a cada uno de estos bytes individualmente, porque cada uno tiene una **dirección única** que lo identifica.

Llamamos **puntero** a una variable que guarda la dirección de un valor en memoria. Por ejemplo, si la memoria de la dirección `0xF0` (hexadecimal) a `0xF8` tiene esta pinta:



Podríamos declarar las siguientes variables:

Snippet 12:

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

int main(){
  uint8_t *x = (uint8_t*) 0xF0;
  int8_t *y = (int8_t*) 0xF6;

  printf("Dir de x: %p Valor: %d\n", (void*) x, *x);
  printf("Dir de y: %p Valor: %d\n", (void*) y, *y);

  //Devolverá:
  //  Dir de x: 0xF0 Valor: 255
  //  Dir de y: 0xF6 Valor: -128
}
```

Ejercicio 5:

¿Por qué `x` e `y` tienen distintos tipos? ¿Qué representan?

Si intentan ejecutar ese código, probablemente el programa falle con un error de tipo `Segmentation fault (core dumped)`. En nuestra computadora, la memoria esta siendo usada por varios componentes y es estadísticamente imposible que las direcciones `0xF0` a `0xF8` tengan los valores que

[1] Tener memorias del tamaño que utilizamos hoy, direccionables con granularidad de bit sería caro y dificultoso de implementar a nivel hardware. Este post de quora da una intuición al respecto https://www.quora.com/Why-do-memory-addresses-use-the-unit-of-byte-instead-of-bit.

usamos en este ejemplo. De hecho, las direcciones `0xF0` a `0xF8` de memoria probablemente no estén a nuestro alcance porque seguramente las esté usando otro proceso (por ende ocurre un `Segmentation Fault`).

Si queremos una tira contigua de bytes como la mostrada en la imagen, necesitamos declararla. Para eso, ¡podemos usar arrays! Como vimos en la guía anterior.

Ejercicio 6:

Completar los ?? en el siguiente código:

Snippet 13:

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

int main(){
    int8_t memoria[??] = ??;
    uint8_t *x = (uint8_t*) ??;
    int8_t *y = ??;

    printf("Dir de x: %p Valor: %d\n", (void*) x, *x);
    printf("Dir de y: %p Valor: %d\n", (void*) y, *y);
}
```

Pista: ¿Para qué sirve el operador `&`? Revisar los ejemplos de código anteriores o ver la sección Declaración vs Desreferencia.

Claramente, podemos usar el puntero, no solo para leer el valor de una dirección de memoria, sino también para escribir en ella.

Snippet 14: Escritura a través de puntero

```c
#include <stdio.h>

int main(){
    int x = 42;
    int *p = &x;

    printf("x: %d\n", x);
    *p = 200;
    printf("x: %d\n", x);
}
```

```
x: 42
x: 200
```

Declaración vs Desreferencia

Habrán visto en los ejemplos hasta ahora que al trabajar con punteros contamos con dos operadores principales: `*` y `&`. Definamos propiamente:

- `*`: se usa en dos situaciones
  - Al **declarar** un puntero: indica que la variable siendo declarada es un puntero al tipo indicado. Por ejemplo, `int *num;`[2] significa que `num` es un puntero a un número entero.

[2] es equivalente declararlo como `int* num;`. Como era de esperarse, esto lleva a innumerables debates no saldados

- Cuando acompaña a una variable ya declarada: nos permite **desreferenciar** la variable que acompaña (que debería ser un puntero), permitiéndonos acceder al valor que se encuentra en la dirección guardada en la variable. `*` en este contexto, es también llamado **operador de indirección**.

  Por ejemplo, en el Snippet 11, pueden ver que en el último `printf`, donde imprimimos el valor de lo que apunta `p`, para acceder a ese valor usamos `*p`. Efectivamente, ese `printf` resulta en que se imprima `42` a consola.

- `&`: se usa para obtener la dirección de una variable. En el Snippet 11, usamos `&x` para inicializar `p` ya que queríamos que el puntero `p` guardara la dirección de la variable `x` (es decir, la dirección donde está guardado el valor 42).

`void*`

Un caso particular de punteros es `void*`, que es un puntero genérico. Sirve para almacenar una dirección de memoria de cualquier tipo, pero no se puede desreferenciar directamente. ¿Para qué tener un puntero genérico? Bueno, es útil cuando no sabemos de antemano el tipo de dato al que va a apuntar el puntero. Por ejemplo, en funciones que pueden recibir punteros a diferentes tipos de datos, como `malloc` o `free`, que veremos más adelante.

La ventaja es que un `void*` puede apuntar a cualquier tipo de dato, pero la desventaja es que no podemos realizar operaciones aritméticas directamente con él. Para usar un puntero `void*`, debemos convertirlo al tipo de puntero adecuado antes de usarlo. Por ejemplo:

Snippet 15:

```c
void* ptr;
int x = 42;
ptr = &x; // ptr apunta a la dirección de x

// Convertir el puntero void* a un puntero int*
int* p = (int*)ptr;
printf("Valor: %d\n", *p); // Imprime 42
```

En este ejemplo, `ptr` es un puntero `void*` que apunta a la dirección de `x`. Luego, convertimos `ptr` a un puntero `int*` para poder desreferenciarlo y acceder al valor de `x`.

Punteros a NULL

Un puntero puede no apuntar a nada, o `NULL`. Esto es útil para indicar que el puntero no está inicializado o que no apunta a una dirección válida. `NULL` es una constante definida en `stddef.h` y se corresponde con el valor entero `0`. Podemos usar el operador `==` para comparar un puntero con `NULL` y verificar si apunta a una dirección válida o no. O también, podemos usar a un puntero en una condición lógica (ya que `NULL` es considerado **falso** en C).

en internet, por ejemplo: https://stackoverflow.com/questions/398395

Snippet 16: Puntero a NULL

```c
#include <stdio.h>

int main(){
    int *p = NULL;
    if (p == NULL) {
        printf("El puntero no apunta a nada.\n");
    } else {
        printf("El puntero apunta a: %d\n", *p);
    }
}
```

`NULL` puede ser asignado a un puntero de cualquier tipo (`char *`, `int*`, `void*`, etc), y es una buena práctica inicializar los punteros a `NULL` al declararlos. Esto ayuda a evitar errores de acceso a memoria no válida y facilita la depuración del código. Como es de esperarse, si intentamos desreferenciar un puntero que apunta a `NULL`, obtendremos un error de segmentación (segmentation fault). Muchas funciones de la biblioteca estándar de C que devuelven punteros, como `malloc`, devuelven `NULL` si no pueden asignar memoria. Por lo tanto, es importante verificar si el puntero es `NULL` antes de usarlo.

La inicialización de punteros a `NULL` es una buena práctica, ya que evita el uso accidental de punteros no inicializados, lo que puede llevar a errores difíciles de depurar.

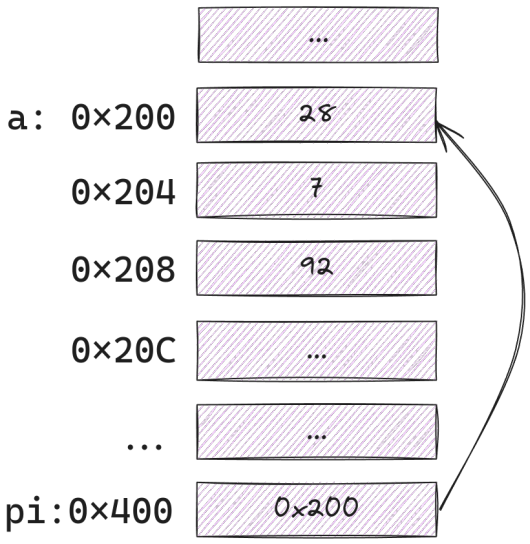
Aritmética de punteros

La aritmética de punteros es una característica poderosa de C que permite realizar operaciones matemáticas con punteros. Esto es útil para recorrer arrays y estructuras de datos dinámicas.

Veamos la siguiente secuencia:

Snippet 17:

```c
uint32_t a[] = {28,7,92};
uint32_t *pi = a;

printf("%d\n", *pi); //28
```



Notamos que el puntero `pi` apunta a la dirección de memoria de `a`, que es la dirección de memoria del primer elemento del array. Momento... entonces ¿qué es `a`? ¿Es un puntero? ¿Es un array? `a` representa un array, pero en C, un array se comporta como un puntero al primer elemento del array. Por lo tanto, `a` y `pi` apuntan a la misma dirección de memoria.

En la mayoría de los contextos, `a` se puede usar como un puntero al primer elemento del array. Sin embargo, hay algunas diferencias importantes entre arrays y punteros que debemos tener en cuenta:

- Un array tiene un tamaño fijo, mientras que un puntero puede apuntar a cualquier dirección de memoria.

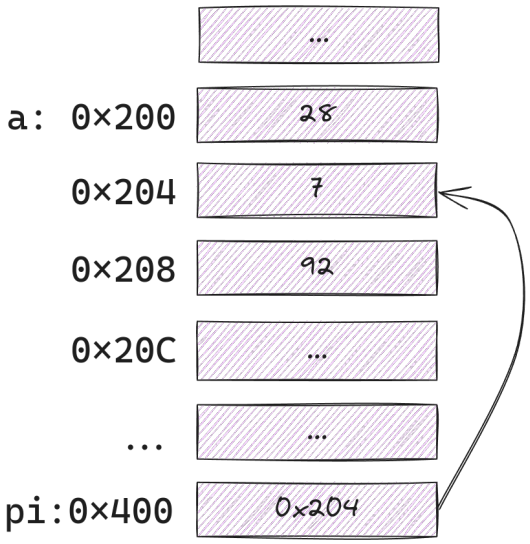
- Un array no se puede reasignar a otra dirección de memoria, mientras que un puntero sí.

Esto significa que no podemos hacer algo como `a = pi;` porque `a` es un array y no se puede reasignar. Sin embargo, podemos hacer `pi = a;` porque `pi` es un puntero y puede apuntar a cualquier dirección de memoria.

Sigamos con el ejemplo:

Snippet 18:

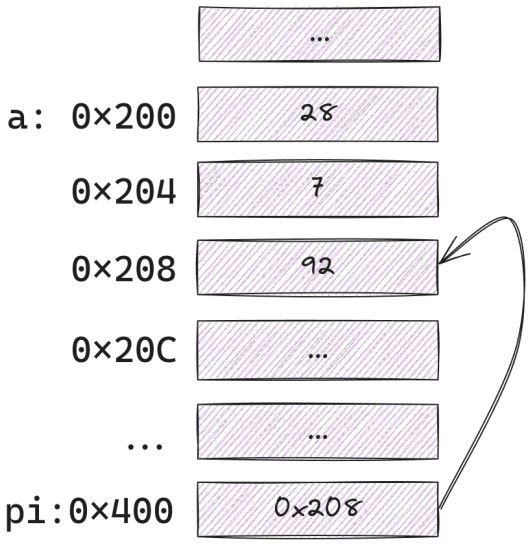
```
uint32_t a[] = {28,7,92};
uint32_t *pi = a;

printf("%d\n", *pi); //28
pi += 1;
printf("%d\n", *pi); //7
```



Bien... entonces podemos usar el puntero para movernos por el array. Algo importante a notar es que la operación de "sumar 1", en realidad no suma 1, sino que suma el tamaño del tipo al que apunta el puntero. En este caso, `pi` es un puntero a `uint32_t`, que ocupa 4 bytes. Por lo tanto, al sumar 1 a `pi`, estamos sumando 4 bytes a la dirección de memoria que contiene el primer elemento del array. Esto significa que `pi` ahora apunta al segundo elemento del array.

Snippet 19:

```
uint32_t a[] = {28,7,92};
uint32_t *pi = a;

printf("%d\n", *pi); //28
pi += 1;
printf("%d\n", *pi); //7
pi += 1;
printf("%d\n", *pi); //92
```



Ahora, si seguimos sumando 1 a `pi`, llegamos al tercer elemento del array. Si seguimos sumando

1, llegaremos a la dirección de memoria que sigue al último elemento del array. Claramente, que un puntero apunte a un lugar de memoria no definido no trae ningún problema, el problema sería si queremos leer o escribir ese lugar de memoria. En ese caso, el comportamiento es indefinido, porque básicamente no sabemos que hay en esa dirección de memoria. Puede haber memoria de nuestro proceso, o puede haber memoria de otro proceso. En caso que haya memoria de otro proceso, el sistema operativo nos va a arrojar un error de `Segmentation fault`. En caso que

haya memoria de nuestro proceso, el comportamiento es indefinido: el programa puede seguir ejecutándose con errores lógicos o puede terminar abruptamente en un `Segmentation fault`, según la operación y el tipo de memoria accedida.

Veamos otro ejemplo, para que quede claro el concepto de aritmética de punteros:

Snippet 20:

```c
int arr[7] = {1,2,3,4,5,6,7};
int *p = arr;
// p apunta a la dirección del primer elemento del array

printf("%d\n", *p); // imprime 1
printf("%d\n", *(p+1)); // imprime 2
printf("%d\n", *(p+2)); // imprime 3
printf("%d\n", p[4]); // imprime 5
```

- `p` es un puntero a `int`, por lo tanto, al hacer `p+1` estamos sumando 4 bytes a la dirección de memoria de `p`
- al hacer `p+1` estamos apuntando a la dirección de memoria del segundo elemento del array
- `p[4]` es equivalente a `*(p+4)`, es decir, estamos accediendo al quinto elemento del array

Se puede pensar el operador `[]` como un operador de desreferencia. Es un *shorthand* que mueve el puntero y lo desreferencia al mismo tiempo.

Snippet 21:

```c
p[0] == *(p+0) == *p
p[1] == *(p+1)
...
p[n] == *(p+n)
```

Punteros como argumentos

Los punteros son útiles para pasar argumentos a funciones sin necesidad de copiar el valor completo. Esto es especialmente útil para estructuras grandes o arrays. Al pasar un puntero, estamos pasando la dirección de memoria del valor, lo que permite a la función modificar el valor original.

Aqui vemos la implementación típica de la función `swap`, que intercambia los valores de dos variables enteras usando punteros:

Snippet 22:

```c
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main() {
    int x = 10, y = 20;
    swap(&x, &y);
    printf("x: %d, y: %d\n", x, y);
}
```

Algo bastante común es encontrar funciones que reciben punteros como `const int*` o `const char*`. Esto significa que la función no puede modificar el valor al que apunta el puntero. Por ejemplo, si tenemos una función que recibe un puntero a un string, podemos usar `const char*` para indicar que la función no puede modificar el string original. Esto es útil para evitar errores de modificación accidental de datos y para indicar que la función no necesita modificar el valor. Notar que no es el puntero en sí que es constante, sino que a través de ese puntero no podemos modificar el valor al que apunta.

Ejercicio 7:

Explicar qué sucedería si la firma de la función `swap` fuera `void swap(int a, int b)`. En ese caso, ¿podríamos intercambiar los valores de `x` e `y`?

Strings revisited

Ahora que vimos punteros, podemos ver que los strings en C son simplemente punteros a `char`, terminados en `'\0'`. Esto significa que podemos usar punteros para manipular strings de manera eficiente. Por ejemplo, podemos usar punteros para recorrer un string y modificar sus caracteres.

Tener en cuenta que los literales de strings se escriben con comillas dobles (por ejemplo, `"hola"`, `"f"`) y los literales de `char` se escriben con comillas simples y se componen de un solo carácter (por ejemplo, `'h'`, `'f'`, `'\n'`). El especificador de conversión para chars es `%c` y el de strings es `%s`.

Veamos un ejemplo de cómo usar punteros para manipular strings:

Snippet 23:

```c
#include <stdio.h>

site_t lenght(char *str) {
    size_t len = 0;
    while (*str != '\0') {
        len++;
        str++;
    }
    return len;
}

int main() {
    char str[] = "This too shall pass";
    printf("Length of string: %zu\n", lenght(str));
  return 0;
}
```

Ejercicio 8:

Pensar si existe una diferencia entre definir un string como `str1` o como `str2` en el siguiente código:

Snippet 24:

```c
int main(){
  char *str1 = "Hola";
  char str2[] = "Hola";

  printf("%s\n", str1);
  printf("%s\n", str2);
  return 0;
}
```

¿Es lo mismo? Parte de la solución es pensar en la memoria. Veremos un poco más adelante como se organiza la memoria en C.

Ejercicio 9:

Definir una función que reciba un string y lo pase a mayúsculas. **Pista:** alcanza con sumar la diferencia entre `'A'` y `'a'` a cada letra. Sólo aplicar la conversión si lo que llega es una letra minúscula.

Ejercicio 10:

Crear programas demostrativos que utilicen las funciones de strings provistas en la librería estándar de C (en `string.h`): `strcpy`, `strcat`, `strlen`, `strcmp`, etc. Para acceder a la documentación de cada función se puede usar la línea de comando: `man strcpy`, `man strlen`, etc.
Side quest: investigar que significa `restrict` en la firma de la función `strcpy`

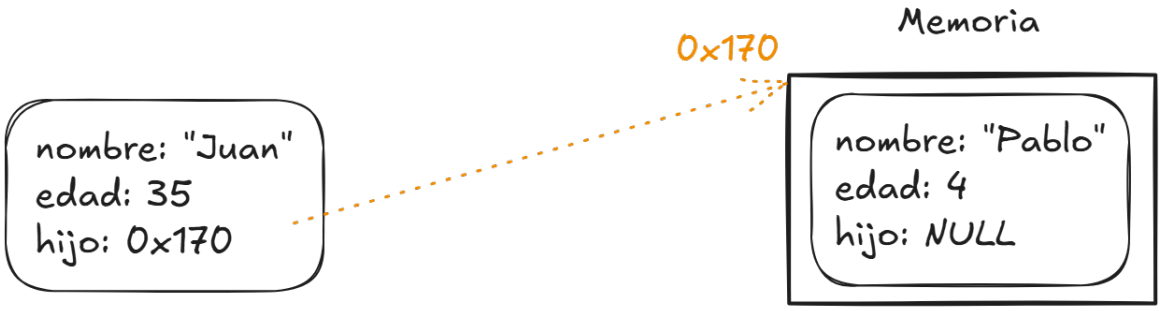
struct revisited

Es normal, al declarar estructuras, querer tener un puntero al mismo tipo de estructura que estamos definiendo. Para hacer esto, lo tenemos que hacer de la siguiente manera:

Snippet 25:

```
  #define NAME_LEN 50

  typedef struct persona_s {
    char nombre[NAME_LEN+1];
    int edad;
    struct persona_s* hijo;
} persona_t;
```

Ejemplo: Juan tiene un hijo que se llama Pablo



Juan y Pablo

Nota: usamos `NULL` para indicar que Pablo no tiene hijos

Podemos crear arreglos de structs y funciones que toman structs como parámetros (o retornan structs). En este caso tenemos una función que dado un puntero a un array de `persona_t` y la longitud, imprime nombre y edad contenido en cada struct.

Snippet 26:

```
void imprimir_personas(persona_t* personas, uint32_t longitud_array) {
    for (uint32_t i = 0; i < longitud_array; i++) {
        printf("Nombre: %s, Edad: %d\n", personas[i].nombre, personas[i].edad);
    }
}
```

Memoria

Memory never recaptures reality. Memory reconstructs. All reconstructions change the original, becoming external frames of reference that inevitably fall short

---

Tipos de memoria

En C tenemos varios tipos de memoria, y cada uno tiene sus propias características. Es importante entender estos tipos de memoria para poder usar C de manera efectiva. Lo que viene a continuación es bastante técnico. Quizás, una vez vistos los ejemplos que siguen, se entienda mejor.

Entender los tipos de memoria es fundamental para entender como funciona C.

- **Memoria de código**: es la memoria que se reserva para almacenar el código del programa. Su tamaño no puede cambiar durante la ejecución del programa. Es de solo lectura. Se define en el **text segment** del programa.
- **Memoria automática**: es un tipo de memoria que se reserva en el momento de la ejecución. Las variables locales se almacenan en esta memoria. Su tamaño puede cambiar durante la ejecución del programa. Es el llamado **stack** o pila.
- **Memoria estática**: es la memoria que se reserva en el momento de la compilación. Las variables globales y estáticas se almacenan en esta memoria. Su tamaño no puede cambiar durante la ejecución del programa. Se definen en el **data segment** del programa.
- **Memoria dinámica**: es la memoria que se reserva en el momento de la ejecución, pero su tamaño puede cambiar durante la ejecución del programa. Se utiliza para almacenar datos que no se conocen en tiempo de compilación. Es el llamado **heap**.

Dependiendo de dónde y cómo declaramos nuestras variables, podremos hacer uso de una u otra memoria. Ademas de duración y scope, que vimos anteriormente, cada tipo de memoria tiene un **linkage** diferente. El linkage de un símbolo (variable o función) es la forma en que es accesible desde diferentes archivos o módulos.

Existen tres tipos de linkage:

- **interno**: el símbolo es accesible solo dentro del archivo donde se declara.
- **externo**: el símbolo es accesible desde cualquier archivo que incluya su declaración.
- **no-linkage**: el símbolo no tiene linkage, es decir, no es accesible fuera de su scope.

La siguiente tabla muestra un panorama de los tipos de memoria, su scope, duración y linkage:

Scope	Lugar	Duración	Linkage
Global (a toda la aplicación)	data segment	el tiempo de vida de la aplicación (estática)	externo
Global (al archivo)	data segment	el tiempo de vida de la aplicación (estática)	interno
Local a la función o bloque (estática)	data segment	el tiempo de vida de la aplicación (estática)	no linkage
Local a la función o bloque (automática)	stack	Mientras la función (o bloque) esté en ejecución (automática)	no linkage
Dinámica	heap	Hasta que la memoria sea liberada	no linkage

El linkage de una variable definida por fuera de cualquier función lo especificamos con la palabra clave `static`. Esto significa que la variable no es accesible desde otros archivos, pero su duración es la misma que la de una variable global. Por lo tanto, el linkage de una variable estática es interno. Por defecto, el linkage de una variable global es externo. Esto también aplica a funciones que tiene por defecto un linkage externo. Si queremos que una función tenga un linkage interno, debemos declararla como `static`. Esto significa que la función no es accesible desde otros archivos.

Veamos ejemplos de lo anterior:

Snippet 27: archivo lib.h

```c
#ifndef LIB_H
#define LIB_H

// Idea: sólo exponer lo necesario a otros archivos

extern int counter;         // Solo declaración, la definición está en lib.c
void area(double radius);   // linkage externo.

#endif
```

Snippet 28: archivo lib.c

```c
#include <stdio.h>
#include "lib.h"

// definición  de sólo lectura con linkage interno
// (solo se puede usar en este archivo)
static const double pi = 3.14159;

int counter = 0; // definición con linkage externo

// definición con linkage interno (solo se puede usar en este archivo)
static int power(int x) {
    return x * x;
}

// definición con linkage interno (solo se puede usar en este archivo)
static void print(double area){
  printf("Area: %.3f\n", area);
}

void area(double radius) { // definición con linkage externo
    double area = pi * power(radius);
    print(area);
    counter++;
}
```

Snippet 29: archivo main.c

```c
#include <stdio.h>
#include "lib.h"

int main() {
    double radius = 5.0;
    area(radius); // llamada a la función con linkage externo

    // acceso a la variable con linkage externo
    printf("Counter: %d\n", counter);

    // print(3.14); // Error: no está en el header y es static en lib.c
    // power(3);    // Error: no está en el header ni tiene linkage externo
    return 0;
}
```

Más información en https://en.cppreference.com/w/c/language/storage_duration.

Hemos visto hasta ahora la memoria automática y la memoria estática. La memoria automática es la que se reserva en el *stack*, y la memoria estática es la que se reserva en el *data segment*. La memoria dinámica es la que se reserva en el *heap*, y es la que vamos a ver a continuación.

Memoria dinámica

La memoria dinámica es la que se reserva en el *heap*, y es la que se utiliza para almacenar datos que no se conocen en tiempo de compilación. La memoria dinámica se reserva y libera en tiempo de ejecución, lo que permite crear estructuras de datos dinámicas como listas enlazadas, árboles, etc.

Para reservar memoria dinámica en C, usamos las funciones `malloc`, `calloc`, `realloc` y `free`. Estas funciones son parte de la biblioteca estándar de C y se encuentran en el archivo de cabecera `stdlib.h`.

- `malloc`: reserva un bloque de memoria de un tamaño específico y devuelve un puntero a la dirección de memoria reservada. La memoria no se inicializa.
- `calloc`: reserva un bloque de memoria para un número específico de elementos de un tamaño específico y devuelve un puntero a la dirección de memoria reservada. La memoria se inicializa a cero.
- `realloc`: cambia el tamaño de un bloque de memoria previamente reservado y devuelve un puntero a la nueva dirección de memoria. Si la nueva dirección es diferente, la memoria anterior se libera automáticamente.
- `free`: libera un bloque de memoria previamente reservado.

Las declaraciones de estas funciones son las siguientes:

Snippet 30:

```c
void* malloc(size_t size);
void* calloc(size_t num, size_t size);
void* realloc(void* ptr, size_t size);
void free(void* ptr);
```

Y se encuentran en `stdlib.h`.
Analicemos un ejemplo de una función que intenta devolver un array de enteros inicializado:

Snippet 31:

```c
uint16_t *secuencia(uint16_t n){
  uint16_t arr[n];
  for(uint16_t i = 0; i < n; i++)
    arr[i] = i;
  return arr;
}
```

Ejercicio 11:

Antes de seguir avanzando, discutan qué sucede si corren el siguiente código:

Snippet 32:

```c
#include <stdio.h>

int main(){
  uint16_t *arr = secuencia(10);
  printf("%d\n", arr[0]);
  return 0;
}
```

El problema, básicamente radica en que la variable `arr` es una variable automática, y por lo tanto, se libera al salir de la función. Esto significa que la dirección de memoria a la que apunta `arr` ya no es válida al salir de la función. Por lo tanto, el puntero `arr` en la función `main` apunta a una dirección de memoria inválida. Esto es un error común en C, y es importante tener cuidado al usar variables automáticas y punteros. Para evitar este error, podemos usar memoria dinámica para reservar el array. Veamos cómo hacerlo:

Snippet 33:

```c
uint16_t *secuencia(uint16_t n){
  uint16_t *arr = malloc(n * sizeof(uint16_t));
  if (arr == NULL) {
    // Manejar el error de asignación de memoria
    return NULL;
  }
  for(uint16_t i = 0; i < n; i++)
    arr[i] = i;
  return arr;
}
```

Esta función la utilizaríamos de la siguiente manera:

Snippet 34:

```c
#include <stdio.h>

int main(){
  uint16_t *arr = secuencia(10);
  if (arr == NULL) {
    // Manejar el error de asignación de memoria
    return 1;
  }
  for(uint16_t i = 0; i < n; i++)
    printf("%d\n", arr[i]);
  free(arr); // Liberar la memoria reservada
  return 0;
}
```

Notemos que al usar `malloc`, la memoria se reserva en el *heap* y no se libera automáticamente al salir de la función. Por lo tanto, debemos liberar la memoria manualmente usando `free`. Es normal que al alocación de memoria suceda en una función y la liberación en otra. Cuando nos olvidamos de liberar la memoria, se produce una **fuga de memoria** (*memory leak*), que puede causar problemas de rendimiento y estabilidad en el programa. Por lo tanto, es importante liberar la memoria reservada cuando ya no la necesitamos.

Ejercicio 12:

Definir una función `crearPersona` que reciba un nombre y una edad, y devuelva un puntero a una estructura `persona_t` que contenga esos datos. La función debe reservar memoria dinámica para la estructura y devolver el puntero. Luego, en la función `main`, crear una persona y liberar la memoria reservada. El string que se pasa por parámetro debe ser copiado a la estructura.

Ejercicio 13:

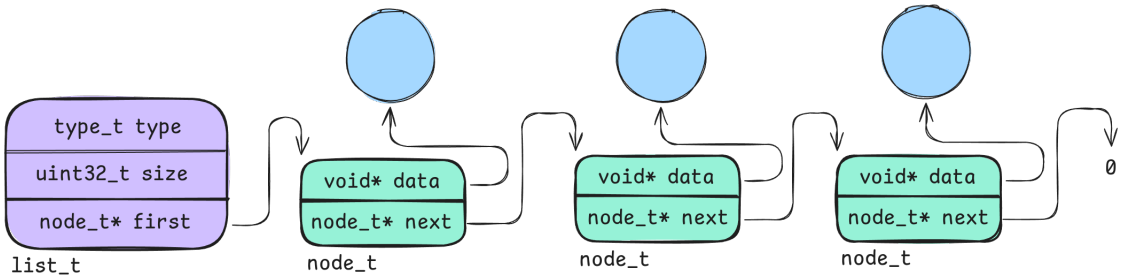
Definir una función `eliminarPersona` que reciba un puntero a una estructura `persona_t` y libere la memoria reservada para la estructura.

Creando una lista simplemente enlazada

Ahora, veamos un ejemplo un poco más realista. Supongamos que queremos crear una lista simplemente enlazada de nodos, donde cada nodo contiene una dirección que apunta a un archivo

en un sistema de archivos determinado. La lista puede contener un número variable de nodos, con la restricción de que una lista solo puede tener nodos que apunten al mismo tipo de sistema de archivos.

Podemos ver la estructura que queremos crear en el siguiente diagrama:



Lista simplemente enlazada

Tenemos una primera estructura que representa la lista y que tiene un puntero al primer nodo de la lista. Luego cada nodo, referencia al nodo siguiente. El último nodo apunta a `NULL` para indicar el final de la lista. Cada nodo tiene un puntero a un archivo, especificado como `void *data`. La lista también tiene un `size` que indica la cantidad de nodos en la lista. La lista tiene un `type` que indica el tipo de sistema de archivos que puede manejar esa lista.

Veamos primero la declaración es estas estructuras:

Snippet 35: list.h

```c
#include "type.h"

typedef struct node {
  void* data;
  struct node* next;
} node_t;

typedef struct list {
  type_t type;
  uint8_t size;
  node_t* first;
} list_t;

list_t* listNew(type_t t);
void  listAddFirst(list_t* l, void* data); //copia el dato
void* listGet(list_t* l, uint8_t i); //se asume: i < l->size
void* listRemove(list_t* l, uint8_t i); //se asume: i < l->size
void  listDelete(list_t* l);
```

También vemos en este archivo que se declaran las funciones para crear la lista, agregar nodos, obtener nodos, eliminar nodos y eliminar la lista.

- `listNew` crea una nueva lista y la inicializa con el tipo de sistema de archivos especificado. La función devuelve un puntero a la lista creada.
- `listAddFirst` agrega un nuevo nodo al principio de la lista. La función recibe un puntero a la lista y un puntero a los datos que se van a agregar. La función copia los datos en el nuevo nodo.
- `listGet` obtiene un nodo de la lista en la posición especificada. La función recibe un puntero a la lista y un índice. La función devuelve un puntero a los datos del nodo.

- `listRemove` elimina un nodo de la lista en la posición especificada. La función recibe un puntero a la lista y un índice. La función devuelve un puntero a los datos del nodo eliminado.
- `listDelete` elimina la lista y libera la memoria reservada para los nodos y los datos. La función recibe un puntero a la lista.

El archivo `type.h` tiene la definición de `type_t` y de las funciones de manejo de archivos para cada sistema:

Snippet 36: type.h

```c
// type.h
typedef enum e_type {
  TypeFAT32 = 0,
  TypeEXT4 = 1,
  TypeNTFS = 2
} type_t;

fat32_t* new_fat32();
ext4_t* new_ext4();
ntfs_t* new_ntfs();
fat32_t* copy_fat32(fat32_t* file);
ext4_t* copy_ext4(ext4_t* file);
ntfs_t* copy_ntfs(ntfs_t* file);
void rm_fat32(fat32_t* file);
void rm_ext4(ext4_t* file);
void rm_ntfs(ntfs_t* file);
```

Se usa un `typedef` para definir el tipo `type_t`, que es un `enum` que representa los diferentes tipos de sistemas de archivos. Recordemos que un `enum` es un tipo de dato que permite definir un conjunto de constantes enteras con nombre. En este caso, `TypeFAT32`, `TypeEXT4` y `TypeNTFS` son los nombres de las constantes que representan los diferentes tipos de sistemas de archivos.

Las funciones de tipo `new_` crean archivos, y las funciones de tipo `copy_` copian archivos y las funciones de tipo `rm_` eliminan archivos. No nos interesa en este ejemplo, la definición de esas funciones, así que vamos a suponer que se encuentran en alguna biblioteca externa.

En la implementación haremos uso del operador `->` y del operador `sizeof`. El operador `->` se usa para acceder a los miembros de una estructura a través de un puntero. Por ejemplo, `l->type` accede al miembro `type` de la estructura `list_t` a través del puntero `l`. Esto es equivalente a `(*l).type`, pero es más legible y conveniente. El operador `sizeof` es una función que devuelve el tamaño en bytes de un tipo de dato o de una variable. Por ejemplo, `sizeof(uint16_t)` devuelve el tamaño en bytes del tipo `uint16_t`. Esto es útil para reservar memoria dinámica, ya que necesitamos saber cuántos bytes necesitamos reservar.

Recordar que `malloc` reserva un bloque de memoria de un tamaño específico y devuelve un puntero a la dirección de memoria reservada. La memoria no se inicializa, por lo que es importante inicializarla antes de usarla.

Veamos la implementación de las funciones en el archivo `list.c`:

Snippet 37: list.c

```c
list_t* listNew(type_t t) {
  list_t* l = malloc(sizeof(list_t));
  l->type = t;  // l->type es equivalente a (*l).type
  l->size = 0;
  l->first = NULL;
  return l;
```

```
}

void listAddFirst(list_t* l, void* data) {
  node_t* n = malloc(sizeof(node_t));
  switch(l->type) {
    case TypeFAT32:
      n->data = (void*) copy_fat32((fat32_t*) data);
      break;
    case TypeEXT4:
      n->data = (void*) copy_ext4((ext4_t*) data);
      break;
    case TypeNTFS:
      n->data = (void*) copy_ntfs((ntfs_t*) data);
      break;
  }
  n->next = l->first;
  l->first = n;
  l->size++;
}

//se asume: i < l->size
void* listGet(list_t* l, uint8_t i){
  node_t* n = l->first;
  for(uint8_t j = 0; j < i; j++)
    n = n->next;
  return n->data;
}

//se asume: i < l->size
void* listRemove(list_t* l, uint8_t i){
  node_t* tmp = NULL;
  void* data = NULL;
  if(i == 0){
    data = l->first->data;
    tmp = l->first;
    l->first = l->first->next;
  }else{
    node_t* n = l->first;
    for(uint8_t j = 0; j < i - 1; j++)
      n = n->next;
    data = n->next->data;
    tmp = n->next;
    n->next = n->next->next;
  }
  free(tmp);
  l->size--;
  return data;
}

void listDelete(list_t* l){
  node_t* n = l->first;
  while(n){
    node_t* tmp = n;
```

```
    n = n->next;
    switch(l->type) {
      case TypeFAT32:
        rm_fat32((fat32_t*) tmp->data);
        break;
      case TypeEXT4:
        rm_ext4((ext4_t*) tmp->data);
        break;
      case TypeNTFS:
        rm_ntfs((ntfs_t*) tmp->data);
        break;
    }
    free(tmp);
  }
  free(l);
}
```

Podríamos usar la lista de la siguiente manera:

Snippet 38: main.c

```c
#include <stdio.h>
#include "list.h"

int main() {
  list_t* l = listNew(TypeFAT32);
  fat32_t* f1 = new_fat32();
  fat32_t* f2 = new_fat32();
  listAddFirst(l, f1);
  listAddFirst(l, f2);
  listDelete(l);
  rm_fat32(f1);
  rm_fat32(f2);
  return 0;
}
```

Estudien este ejemplo con detenimiento. Podemos hacer algunas optimizaciones usando punteros a función, que veremos más adelante. Por ahora, lo importante es entender cómo funcionan los punteros y la memoria dinámica en C.

Ejercicio 14:

Implementen el ejemplo anterior, implementando funciones dummy para `new_`, `copy_` y `rm_` .Pueden por ejemplo hacer usar `typedef uint32_t fat23_t` para definir el tipo de dato `fat32_t`. Luego, en las funciones `new_` y `copy_` reservar y asignar un valor a la variable. En la función `rm_` simplemente liberar la memoria.

Ejercicio 15:

Agregar una función a la lista que permita intercambiar el orden de dos nodos. Para esto, se debe tener en cuenta que la lista puede estar vacía o tener un solo nodo. En caso de que la lista tenga un solo nodo o esté vacía, no se debe hacer nada.

Ejercicio 16:

Extender el ejemplo para que la lista sea doblemente enlazada. Esto significa que cada nodo tendrá un puntero al nodo anterior y al siguiente. Esto permitirá recorrer la lista en ambas direcciones. Para esto, se debe agregar una función para agregar un nodo al final de la lista. Mantener un puntero `last` en la lista que apunte al último nodo. Esto permitirá agregar nodos al final de la lista de manera eficiente.

Organización de la memoria

La memoria de un programa en C se organiza en diferentes segmentos. Cada segmento tiene un propósito específico y se utiliza para almacenar diferentes tipos de datos. La siguiente imagen muestra la organización típica de la memoria de un programa en C:

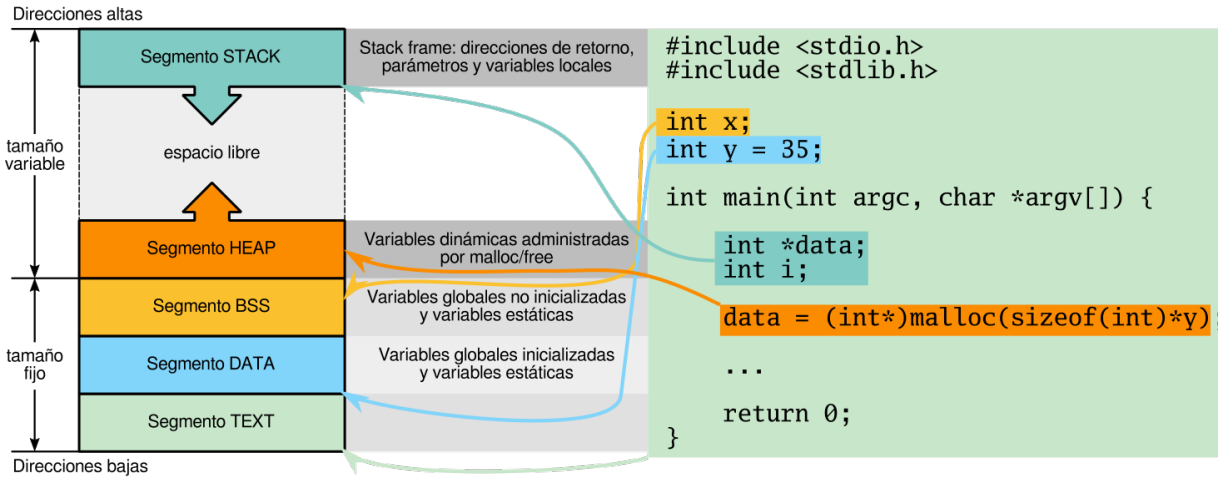


Imagen de memoria

Cuando decimos variables globales no inicializadas, nos referimos a variables globales que no tienen un valor asignado al momento de la declaración por parte del programador. Esas variables, a diferencia de lo que sucede con las variables locales, tenemos garantías que serán inicializadas a 0. Esto es porque el compilador reserva espacio para esas variables en el segmento *bss* y las inicializa a 0. Esto no sucede con las variables locales, que se reservan en el *stack* y no tienen un valor asignado al momento de la declaración. Por lo tanto, su valor es indefinido. Si inicializamos las variables globales, el compilador reserva espacio para esas variables en el *data segment* y las inicializa con el valor que les asignamos. Esto es lo que sucede con las variables globales inicializadas. Las variables estáticas no globales también van a parar al *data segment* o al *bss* según si están inicializadas o no. Vemos además que la pila y el heap crecen en direcciones opuestas. Su tamaño es variable y depende del curso de ejecución que haya tomado el programa. Notar que en el diagrama tenemos el puntero `data` que vive en el stack pero los datos a los que apunta, viven en el heap. Esto suele ser bastante común.

Problemas con memoria dinámica

La memoria dinámica es muy útil, pero también puede ser peligrosa si no se usa correctamente. Algunos problemas comunes son:

- **fugas de memoria (*memory leak*)**: esto sucede cuando se reserva memoria dinámica pero no se libera. Esto puede causar problemas de rendimiento y estabilidad en el programa. Para evitar esto, es importante liberar la memoria reservada cuando ya no la necesitamos.
- **acceso a memoria no válida (*dangling pointer*)**: esto sucede cuando se accede a un puntero que apunta a una dirección de memoria inválida. Esto puede causar errores de segmentación y otros problemas.

- ***double free***: esto sucede cuando se libera la misma dirección de memoria dos veces.
- ***use after free***: esto sucede cuando se accede a un puntero después de haber liberado la memoria a la que apunta.

La herramienta `valgrind` es muy útil para detectar estos problemas. `valgrind` es una herramienta de depuración que se utiliza para detectar errores de memoria en programas. Para usar `valgrind`, es recomendable compilar el programa con la opción `-g` para incluir información de depuración. Luego, se puede ejecutar el programa con `valgrind ./programa`. Luego de ejecutar el comando, `valgrind` mostrará información sobre las fugas de memoria, accesos a memoria no válida y otros problemas de memoria. Más info en `https://valgrind.org/docs/manual/quick-start.html`.

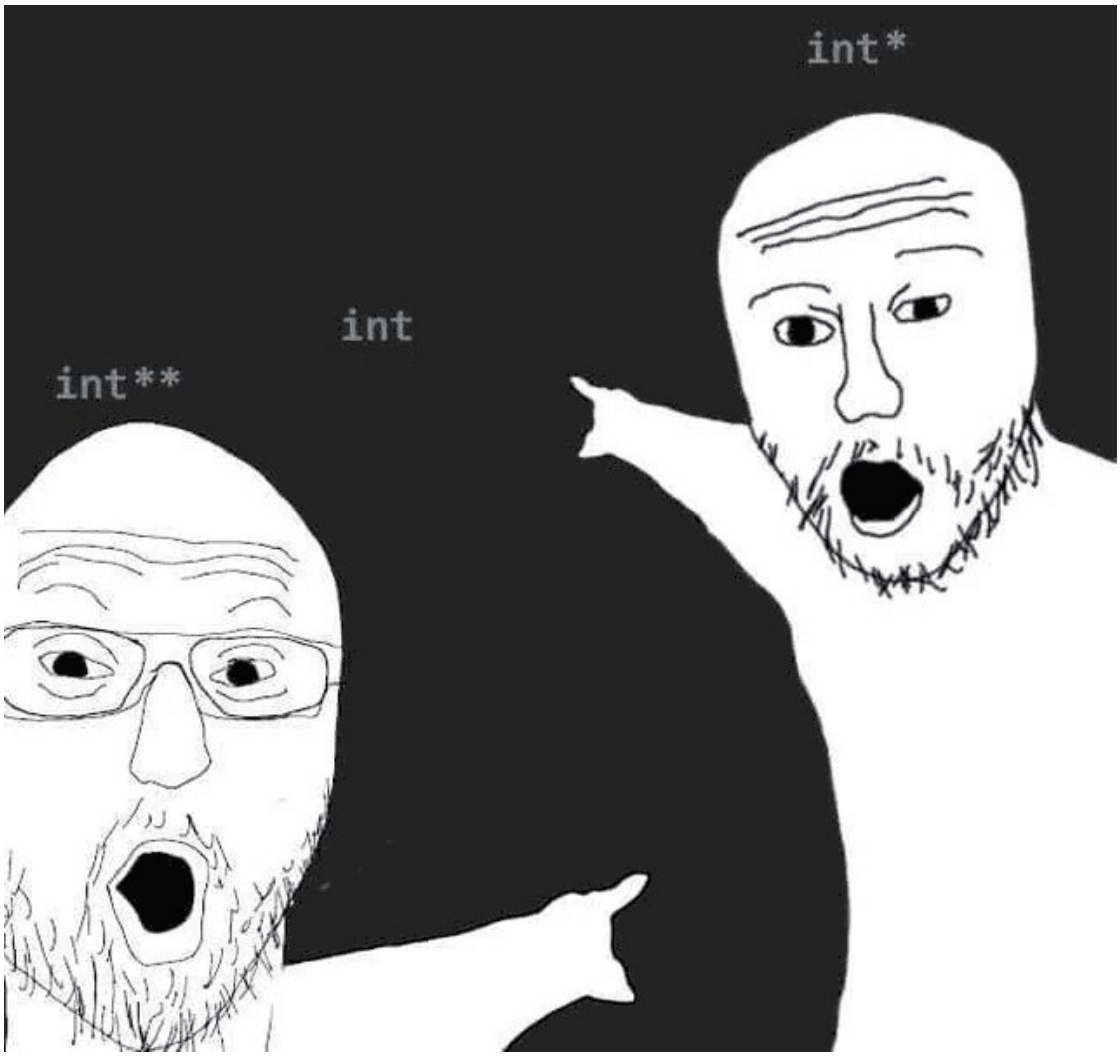
Ejercicio 17:

Forzar un error de memoria en en el ejemplo de la lista enlazada, por ejemplo, al eliminar un nodo, liberar la memoria del nodo pero no liberar la memoria de los datos. Luego correr el programa con `valgrind` y ver qué errores se producen.

Punteros a punteros

Not knowing what you said, you said it.

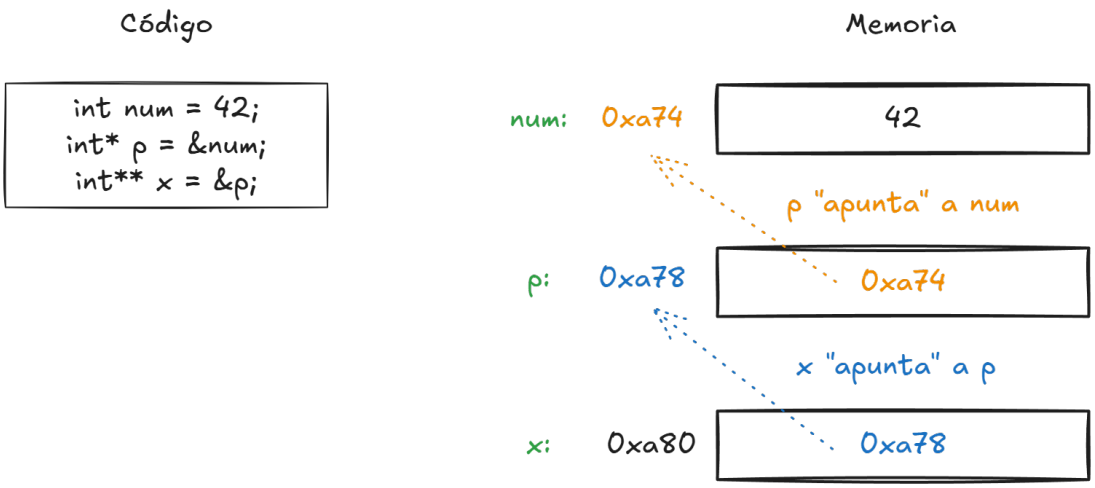
Hasta ahora solo usamos punteros a valores concretos, pero dado que un puntero no es más que la dirección de memoria de algún valor de algún tipo, podemos tener **punteros a cualquier cosa**. ¡Incluso podemos tener punteros a punteros!



Algunos ejemplos:

- `int**`: puntero (`int**`) a un puntero a un entero (`int**`)
- `void**`: puntero a puntero a un tipo **desconocido**. No se puede desreferenciar sin antes especificar su tipo mediante un **casteo**.
- `char* argv[]`: array de punteros a `char`. Esto es lo que se usa para pasar argumentos a la función `main`. En este caso, `argv` es un puntero a un array de punteros a `char`. Es decir que cada puntero apunta a un string (un array de `char`). Es lo mismo escribirlo como `char** argv`[3]

[3]Más info en https://en.cppreference.com/w/c/language/main_function



Analicemos el siguiente ejemplo:

Snippet 39:

```c
void allocateArray(int *arr, int size, int value) {
  arr = (int*)malloc(size * sizeof(int));
  if(arr != NULL) {
    for(int i=0; i<size; i++) {
      arr[i] = value;
    }
  }
}

// Uso
int *vector = NULL;
allocateArray(vector,5,45);
for(int i = 0; i < 5; i++)
  printf("%d\n", vector[i]);
free(vector);
```

Ejercicio 18:

La idea de la función es alocar un array en el heap e inicializar sus valores con el valor pasado por parámetro ¿Qué sucede si corremos el código anterior?

Lo que está pasando acá es que la función `allocateArray` recibe un puntero a un entero, pero al asignar memoria a `arr` dentro de la función, solo se está modificando el puntero local. Esto significa que el puntero `vector` en la función `main` no se ve afectado por la asignación de memoria en la función `allocateArray`. Por lo tanto, al intentar acceder a `vector` después de llamar a `allocateArray`, se produce un error de segmentación.

La implementación correcta sería la siguiente:

Snippet 40:

```c
void allocateArray(int **arr, int size, int value) {
  *arr = (int*)malloc(size * sizeof(int));
  if(*arr != NULL) {
    for(int i=0; i<size; i++) {
      (*arr)[i] = value;
    }
  }
}

// Uso
int *vector = NULL;
allocateArray(&vector,5,45);
for(int i = 0; i < 5; i++)
  printf("%d\n", vector[i]);
free(vector);
```

Arrays multidimensionales

Los arrays multidimensionales son arrays de arrays. En C, los arrays multidimensionales se representan como punteros a punteros. Por ejemplo, un array de dos dimensiones se representa como un puntero a un puntero.

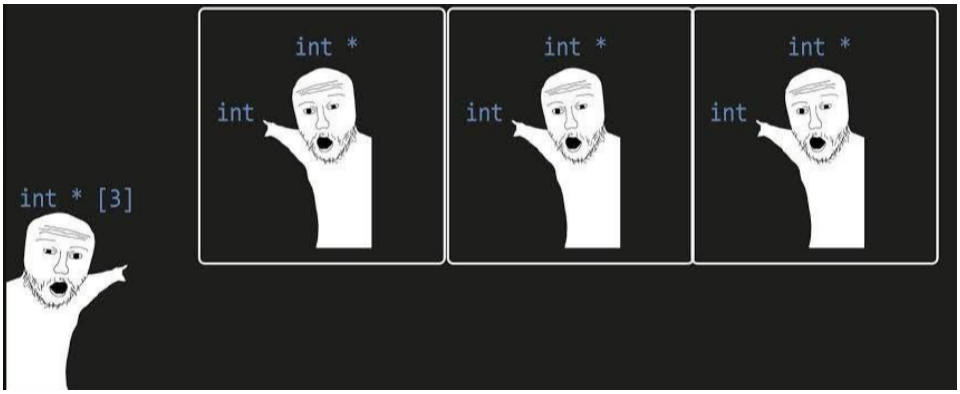
Dado un array de dos dimensiones, definido de la siguiente manera:

Snippet 41:

```c
int a[ROWS][COLS];
```

Podemos pensar en `a` como un puntero a un puntero. En este caso, `a` es un puntero a un array de `COLS` enteros. Por lo tanto, `a` es de tipo `int (*)[COLS]`. Esto significa que `a` es un puntero a un array de enteros de tamaño `COLS`. Notar que ponemos el operador `*` entre paréntesis para indicar que `a` es un puntero a un array de enteros. Si no lo hiciéramos, `a` sería un array de punteros a enteros, que no es técnicamente lo mismo.

Un array de punteros a int se vería algo así:



Consideraciones:

- Un array multidimensional es un array de arrays.
- La memoria se almacena de forma contigua.
- La notación `a[i][j]` es equivalente a `*(a[i] + j)`.

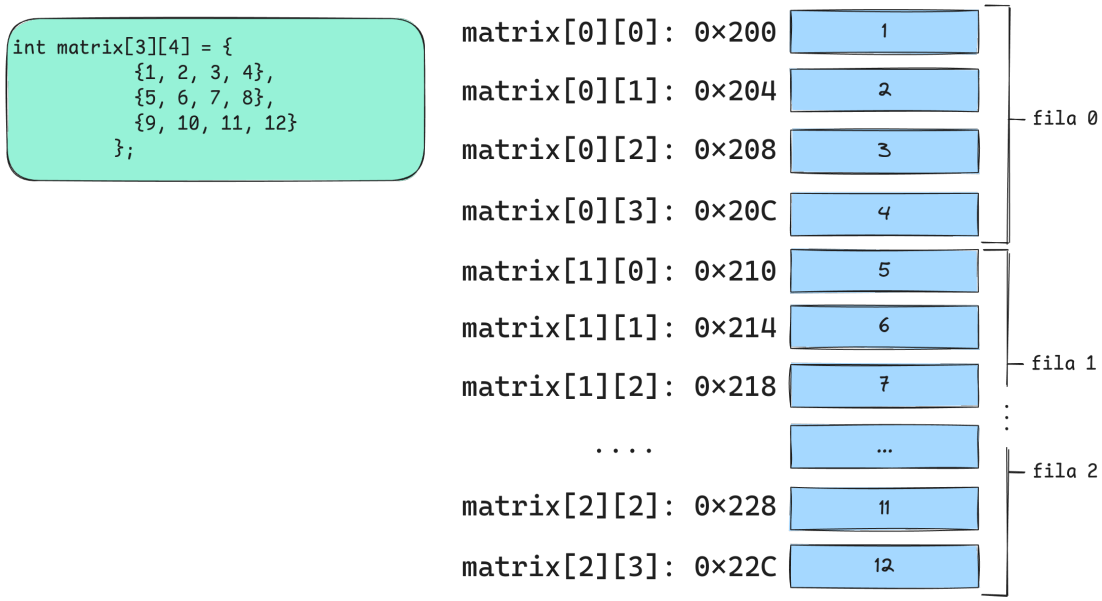
- La notación `a[i][j]` es equivalente a `*(a + i*COLS + j)`.
- El tipo de a es `int (*)[COLS]`.
- En memoria, los arrays multidimensionales se almacenan en orden de fila (*Row-major order*)[4]. Esto significa que los elementos de la primera fila se almacenan en memoria contiguamente, seguidos por los elementos de la segunda fila, y así sucesivamente.

Veamos un ejemplo:

Snippet 42:

```c
int matrix[3][4] = {
  {1, 2, 3, 4},
  {5, 6, 7, 8},
  {9, 10, 11, 12}
};

// Todos estos prints imprimen 7
printf("matrix[1][2]: %d\n", matrix[1][2]);
printf("matrix[1][2]: %d\n", *(*(matrix + 1) + 2));
printf("matrix[1][2]: %d\n", *((int*) matrix + 4*1 + 2));

m[0][3] = 100; // asigna 100 a la fila 0, columna 3
printf("matrix[0][3]: %d\n", matrix[0][3]); // imprime 100
```

La siguiente figura muestra el layout en memoria de dicho array:



[4] https://en.wikipedia.org/wiki/Row-major_order

Ejercicio 19:

Analicen cuidadosamente el siguiente ejemplo y respondan las preguntas:

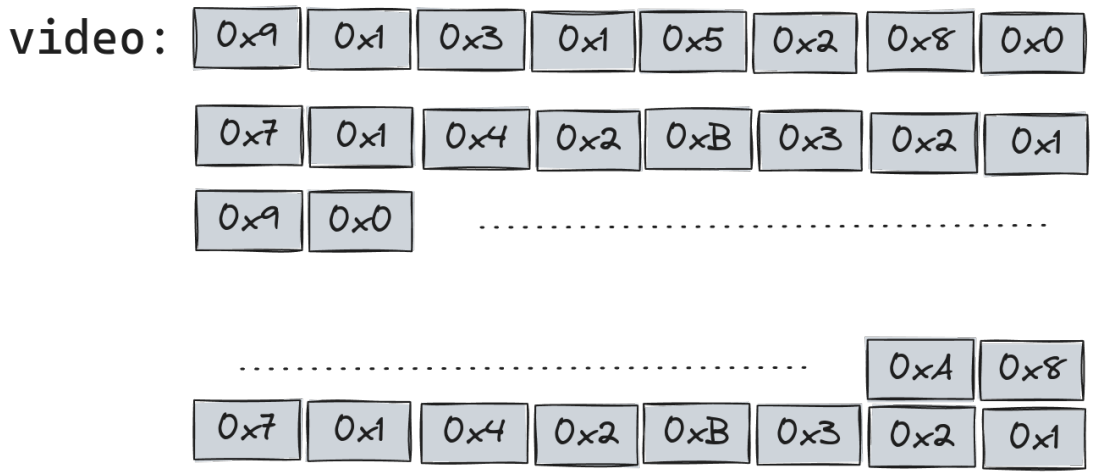
Snippet 43:

```c
#include <stdio.h>
int main() {
    int matrix[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
        };
    // p apunta al int en la fila 0, columna 0
    int *p = &matrix[0][0];

    // ¿qué es reshape?
    int (*reshape)[2] = (int (*)[2]) p;

    printf("%d\n", p[3]); // Qué imprime esta línea?
    printf("%d\n", reshape[1][1]); // Qué imprime esta línea?
    return 0;
}
```

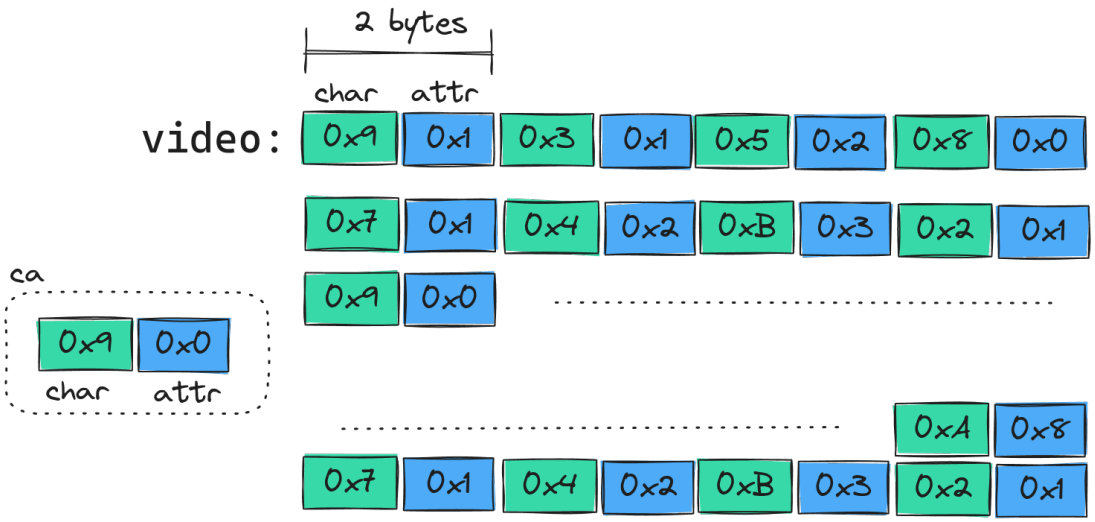
Array de array de structs

Analicemos ahora un caso un poco más complejo. Esto está sacado directamente del taller de system programming que veremos más adelante en la materia.

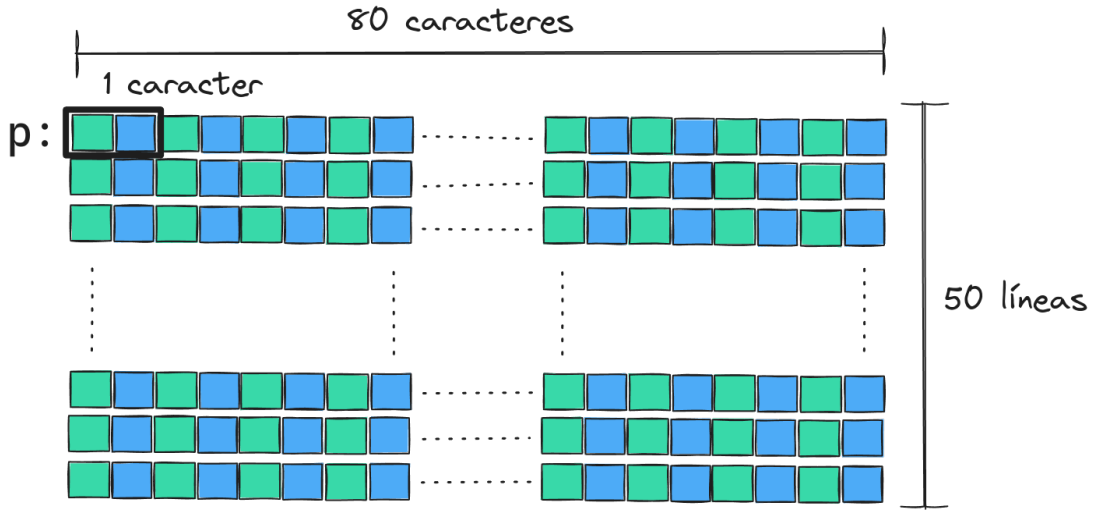
Tenemos una posición de memoria determinada donde se aloja el buffer de vídeo. Todo lo que escribamos en ese buffer se verá reflejado en la pantalla. Recordemos que la memoria en su forma más cruda no es más que un array de bytes, como se ve a continuación:



El buffer de vídeo es un array de bytes y en este formato en particular, cada 2 bytes, se representa un carácter en pantalla. El primer byte representa el carácter en ascii y el segundo byte representa los atributos de color del carácter (*background* y *foreground*). Entonces, nos gustaría interpretar a nuestro buffer como un array de estos elementos de 2 bytes, como se ve en la siguiente figura:



En particular, como una matriz de 80x50, donde cada elemento de la matriz es un *struct* que contiene el carácter y los atributos de color.



El código que veremos a continuación inicializa el buffer de video:

Snippet 44:

```c
#define VIDEO_COLS 80
#define VIDEO_FILS 50
// Cada posicion de memoria tiene 2 bytes
typedef struct ca_s {
  uint8_t c; // caracter
  uint8_t a; // atributos
} ca;
void screen_draw_layout(void) {

  // VIDEO es un puntero a la dirección de memoria del buffer de video
  ca(*p)[VIDEO_COLS] = (ca(*)[VIDEO_COLS])VIDEO;
  uint32_t f,c;
  for (f = 0; f <  VIDEO_FILS; f++) {
    for (c = 0; c < VIDEO_COLS; c++) {
      p[f][c].c = ' ';
      p[f][c].a = 0x10;
    }
  }
}
```

Ejercicio 20:

Analicen el código anterior y respondan las siguientes preguntas:

- ¿Qué hace `ca(*p)[VIDEO_COLS] = (ca(*)[VIDEO_COLS])VIDEO;`?
- Siendo `p` el puntero declarado ¿Por qué funciona esto: `p[f][c].c = ' ';`? Explicar detalladamente el proceso de desreferenciación.
- ¿Qué pasaría si en vez de `ca(*p)[VIDEO_COLS]` hubiéramos declarado `ca** p`?

Punteros a función

Deep in the human unconscious is a pervasive need for a logical universe that makes sense. But the real universe is always one step beyond logic.

Los punteros a función son punteros que apuntan a funciones en lugar de apuntar a datos. Esto permite pasar funciones como argumentos a otras funciones, lo que es útil para implementar callbacks y otras técnicas de programación avanzada. Los punteros a función se declaran de la siguiente manera:

```
tipo_retorno (*nombre_puntero)(tipo_parametro1, tipo_parametro2, ...);
```

Veamos algunos ejemplos de declaraciones de punteros a función:

Snippet 45:

```c
//puntero a una función que toma dos enteros y devuelve un entero
int (*suma)(int, int);

// puntero a una función que toma un puntero a char y devuelve void
void (*callback)(char*);
```

Veamos un ejemplo sencillo de uso:

Snippet 46: puntero a función

```c
#include <stdio.h>

void print_int(int x) {
    printf("%d\n", x);
}

void pretty_print_int(int x) {
    printf("Entero[%lu bits]: %d\n", sizeof(x)*8, x);
}

int main() {
    void (*print)(int) = print_int;
    print(42);  // () desreferencia el puntero a función
    print = pretty_print_int;
    print(3);
}
```

Notar como el operador `()` desreferencia el puntero a función.

Volviendo al ejemplo de la lista enlazada, podemos lograr una mayor flexibilidad al usar punteros a función para manejar los diferentes tipos de archivos. En lugar de tener un `switch` en la función `listAddFirst`, podemos definir un puntero a función que apunte a la función de copia correspondiente según el tipo de archivo. Esto nos permite agregar nuevos tipos de archivos sin modificar el código existente.

Snippet 47: type.h cont.

```
typedef void* (*funcCopy_t)(void*);
typedef void (*funcRm_t)(void*);
funcCopy_t getCopyFunction(type_t t);
funcRm_t getRmFunction(type_t t);
```

Cuando vamos a devolver punteros a función, es conveniente definir un `typedef` para el puntero a función. Esto hace que el código sea más legible y fácil de entender. En este caso, definimos `funcCopy_t` como un puntero a una función que toma un puntero a `void` y devuelve un puntero a `void`. También definimos `funcRm_t` como un puntero a una función que toma un puntero a `void` y no devuelve nada.

Snippet 48: list.c cont.

```
funcCopy_t getCopyFunction(type_t t) {
  switch (t) {
      case TypeFAT32: return (funcCopy_t) copy_fat32; break;
      case TypeEXT4:  return (funcCopy_t) copy_ext4; break;
      case TypeNTFS:  return (funcCopy_t) copy_ntfs; break;
      default: return NULL; break;
  }
}

funcRm_t getRmFunction(type_t t) {
  switch (t) {
      case TypeFAT32: return (funcRm_t) rm_fat32; break;
      case TypeEXT4:  return (funcRm_t) rm_ext4; break;
      case TypeNTFS:  return (funcRm_t) rm_ntfs; break;
      default: return NULL; break;
  }
```

Entonces podemos modificar las funciones `listAddFirst` y `listDelete` para usar los punteros a función:

Snippet 49: list.c cont.

```
void listAddFirst(list_t* l, void* data) {
  node_t* n = malloc(sizeof(node_t));
  n->data = getCopyFunction(l->type)(data);
  n->next = l->first;
  l->first = n;
  l->size++;
}

void listDelete(list_t* l){
  node_t* n = l->first;
  while(n){
    node_t* tmp = n;
    n = n->next;
    getRmFunction(l->type)(tmp->data);
    free(tmp);
  }
  free(l);
}
```

Ejercicio 21:

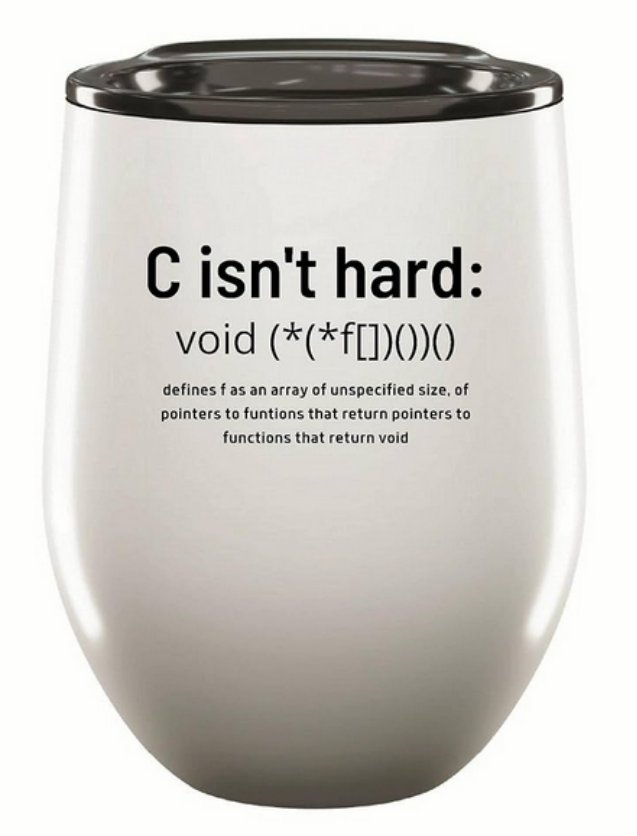
Agregar las modificaciones necesarias al ejemplo de la lista enlazada para usar punteros a función.

Declaraciones confusas

Todo es risas y diversión hasta que nos encontramos con declaraciones siniestras como las siguientes:

Snippet 50:

```c
// puntero a función que toma dos punteros a double y devuelve un puntero
// a un array de 10 double
double (*(*f3)(double*, double*))[10];

// puntero a función que toma un puntero a char y devuelve un puntero a
// función que toma un puntero a int y devuelve void
void (*(*f4)(char*))(int*);
```



Pueden visitar el sitio https://cdecl.org/ para ver declaraciones y su interpretación en inglés. En general las declaraciones pueden volverse más manejables con los `typedef` correspondientes.

Snippet 51:

```
// definimos primero f3_t, como un puntero a función tradicional
typedef double DoubleArray10[10];
typedef DoubleArray10* DoubleArray10Ptr;
typedef DoubleArray10Ptr (*f3_t)(double*, double*);

// y luego declaramos f3 de tipo f3_t
f3_t f3;


// definimos primero InnerFunc, como un puntero a función tradicional
//  que toma un puntero a int y devuelve void
typedef void InnerFunc(int*);

// luego, definimos primero f4_t, como un puntero a función tradicional
//  que toma  un puntero a char y devuelve un puntero a InnerFunc
typedef InnerFunc* (*f4_t)(char*);

// finalmente, declaramos f4 de tipo f4_t
f4_t f4;
```

Debugging en C

Bugs are not a failure, but a discovery.
Each one brings us closer to a solution.

A lo largo de la cursada va a ser indispensable la habilidad para interpretar errores y utilizar las herramientas de *debugging*. En la mayoría de los casos, sobretodo en *Assembly*, mirar fijo el código no es suficiente (ni eficiente) para encontrar los puntos de falla. Repasemos brevemente las herramientas de las que disponemos para esta tarea:

GDB: *debugger* que usaremos para depurar nuestros programas en C o Assembler.

Valgrind: herramienta que verifica el manejo correcto de los recursos del sistema por parte del código ejecutado.

man: utilidad de la terminal de los sistemas UNIX que nos da fácil acceso a la documentación de las funciones de la `libc`, entre otras cosas.

GDB: GNU Debugger

Si bien existen alternativas, como el *printing debugging*, que pueden ser efectivas en casos puntuales, al trabajar con Assembly las cosas se complican rápidamente. Familiarizarse con su uso en C hará mucho más amena la experiencia de uso en assembler y les permitirá programar con confianza en ambos lenguajes.

Con GDB podemos ejecutar un programa paso a paso, establecer puntos de interrupción (*breakpoints*), inspeccionar el valor de variables y registros, e incluso modificar el flujo de ejecución en tiempo real. Esto nos permite asegurarnos de que el estado del programa es el esperado durante toda su ejecución y detectar posibles comportamientos inesperados.

En el campus de la materia, sección **Recursos**, encontrarán dos apuntes con comandos comunes y ejemplos de uso.

Veamos un pequeño ejercicio de ejemplo.

Snippet 52:

```c
#include <stdio.h>

void contarHasta(int hasta){
  for(int i = 0; i < hasta; i++){
    printf("%d\n", i);
  }
}

int main(void){
  contarHasta(10);
  return 0;
}
```

Copiar el snippet a un archivo C y compilarlo con los símbolos de debug activos:
`gcc -g <nombre_del_archivo>.c o main`.

Esto nos generará el ejecutable `main`. Ahora, podemos debuggear el programa con `GDB` a través de éste usando el comando `gdb main` en la terminal (parados en el directorio que lo contenga).

Una vez dentro de `GDB`, podemos empezar a seguir la ejecución del programa. Para eso, necesitamos al menos un breakpoint, de lo contrario el programa ejecutaría normalmente hasta finalizar (con o sin éxito). Para colocar un *breakpoint* usamos `b <nombre_del_archivo>.c:<selector>` donde el selector puede ser el nombre de una función del programa o una línea específica del archivo.

Prueben lo siguiente:

- Colocar un breakpoint en la función main.
- Usar hlrun para avanzar la ejecución hasta el primer *breakpoint*.
- Colocar un *breakpoint* en el llamado a la función `contarHasta`.
- Usar `continue` o `c` para avanzar hasta el siguiente *breakpoint*.
- Podemos usar `step into` o `si` para entrar dentro de la ejecución de la función llamada.
- Una vez entramos en la función, usar `next` o `n` para avanzar a la siguiente línea del archivo actual, independientemente de si se trata de una operación simple o un llamado a función.
- Dentro del ciclo, usar `p <variable>` para ver el valor de las variables de la función en cada paso.

Valgrind: errores comunes y como interpretarlos

Valgrind[5] es una herramienta de depuración que se utiliza para detectar errores de memoria en programas. Para usarla, es recomendable compilar el programa con la opción `-g` para incluir información de debugging y luego basta con ejecutar en consola `valgrind ./<programa>`. El programa se ejecutará y mientras valgrind irá registrando y mostrando información sobre los *leaks* de memoria, accesos a memoria no válida y otros problemas.

Algunos de los mensajes más comunes son:

Invalid Read: lectura de memoria inválida. Notar que nos indica en qué función, archivo y línea se produjo, el tamaño de la lectura, la dirección leída, y si se encuentra cerca de direcciones válidas o no. Por ejemplo, en este caso la dirección leída se encuentra inmediatamente después (*0 bytes after*) de un bloque válido de memoria de 4 *bytes*.

```
==67560== Invalid read of size 4
==67560==    at 0x4012B8: main (debug2.c:68)
==67560==  Address 0x4a9b094 is 0 bytes after a block of size 4 allocd
==67560==    at 0x4848899: malloc (in /usr/libexec/valgrind/...)
==67560==    by 0x4012AF: main (debug2.c:67)
```

Invalid Write: intento de escritura sobre una dirección de memoria inválida. Mismos indicadores que el error de lectura.

```
==4725== Invalid write of size 4
==4725==    at 0x109187: main (main.c:9)
==4725==  Address 0x4a98068 is 0 bytes after a block of size 40 allocd
==4725==    at 0x484880F: malloc (vg_replace_malloc.c:431)
==4725==    by 0x109165: main (main.c:6)
```

Conditional jump or move depends on uninitialised value(s): un salto condicional (un if en C) o una asignación depende de una variable no inicializada. Esto significa que el valor de la variable es basura (no vale asumir que son ceros) ya que no se lo escribió explícitamente, lo que puede resultar en un comportamiento indefinido de nuestro programa (a veces la condición resulta verdadera y otras falsa para el mismo programa).

```
==17726== Conditional jump or move depends on uninitialised value(s)
==17726==    at 0x401258: main (debug2.c:71)
```

Segmentation Fault: El sistema operativo detiene la ejecución del programa si este intenta acceder a memoria que no corresponde al proceso en ejecución, por ejemplo la dirección de

[5] Msinfoenhttps://valgrind.org/docs/manual/quick-start.html

memoria 0x00 (NULL) está reservada por el sistema operativo y saltará un segfault al intentar accederla. La principal diferencia entre este error y los primeros dos mencionados en este apartado es que Valgrind puede atajar esos casos y avisar sobre los accesos indebidos dentro de la memoria del programa. En cambio los `segfault` son faltas que el sistema operativo considera graves y no permite continuar con la ejecución del programa. Notar que Valgrind nos indica cuál fue la dirección a la que se intentó acceder indebidamente (en este caso 0x0).

```
==5902== Process terminating with default action of signal 11 (SIGSEGV)
==5902==  Access not within mapped region at address 0x0
==5902==    at 0x1091A3: main (main.c:14)
```

Memory Leaks: puede que nuestro programa complete su ejecución sin errores y aparente funcionar correctamente, pero esté infringiendo normas de uso correcto de memoria. Los *memory leaks* corresponden con situaciones donde alguno de los *bytes* que nuestro programa pide al sistema para usar durante su ejecución no son correctamente devueltos al sistema al terminar. Esto podría ser causa de comportamiento inesperado a nivel sistema. El sistema operativo recupera la memoria ocupada al terminar la ejecución del programa, pero mientras está en ejecución la memoria pedida es considerada en uso y no disponible para otros programas, con lo que un mal manejo de memoria puede provocar que se agote el recurso para el sistema entero.

```
==17726== LEAK SUMMARY:
==17726==    definitely lost: 16 bytes in 1 blocks
==17726==    indirectly lost: 0 bytes in 0 blocks
==17726==      possibly lost: 0 bytes in 0 blocks
==17726==    still reachable: 0 bytes in 0 blocks
==17726==         suppressed: 0 bytes in 0 blocks
```

En este caso, hay 16 *bytes* que seguían reservados al momento de terminar el programa que no fueron liberados. Además, Valgrind nos da información sobre el uso del *heap* por parte de nuestro programa:

```
==17726== HEAP SUMMARY:
==17726==     in use at exit: 16 bytes in 1 blocks
==17726==   total heap usage: 2 allocs, 1 frees, 1,040 bytes allocated
```

Idealmente, deberíamos tener 0 *bytes* en uso al finalizar la ejecución de nuestro código y por cada `alloc` debe corresponderle su respectivo `free`.

Tipos de Memory Leaks:

- **Definitely Lost:** Sucede cuando se reserva memoria dinámica pero pierde todas las referencias a ese bloque antes de liberarla con `free`. Como ya no existe ningún puntero válido que referencie ese bloque de memoria, no hay forma de liberarlo más adelante. Ejemplo:

  Snippet 53:

  ```
  int* array = (int*) malloc(sizeof(int)*10);
  array = NULL; //pierdo para siempre el puntero al bloque pedido
  ```

- **Indirectly Lost:** Se refiere a cuando se pierde un puntero a un bloque que a su vez tiene punteros a otros bloques. Ejemplo:

Snippet 54:

```c
int** puntero_a_punteros = (int**) malloc(sizeof(int*) * 10);

for(int i=0; i<10;i++){
    puntero_a_punteros[i] = (int*) malloc(sizeof(int));
    *puntero_a_punteros[i] = i;
}

free(puntero_a_punteros);
// Liberamos el puntero al array y, por lo tanto, perdemos acceso
// a los punteros dentro del mismo
```

- **Possibly Lost:** Sucede cuando Valgrind encuentra un puntero al interior de un arreglo. El puntero pudo apuntar al principio del arreglo pero en algún momento se lo avanzó o puede ser que no tenga relación alguna. Valgrind marca estos bloques como dudosos porque no puede determinar si todavía existe un puntero válido para liberar este bloque. Ejemplo:

Snippet 55:

```c
// Pedimos memoria
char *p = malloc(10);

// Incrementamos el puntero
p = p + 1;

// Abortamos la ejecución del programa, simulando un crasheo
abort();

// No se puede asegurar que sea el puntero al inicio
p = p - 1;

// Liberamos el bloque
free(p);
```

- **Still Reachable:** Significa que al terminar el programa quedó memoria reservada que no se liberó, pero todavía existen punteros válidos que la apunta. Ejemplo:

Snippet 56:

```c
int main() {
    int *p = malloc(sizeof(int) * 5);
    return 0;
    // Hay un puntero global a la memoria reservada sin liberar
}
```

- **Suppressed:** Significa que ciertos errores de memoria fueron omitidos intencionalmente gracias a un archivo de supresiones o a la configuración de Valgrind, dado que provienen de alguna librería o fuente de código externo y no tiene que ver con la ejecución del programa ejecutado directamente.

MAN en Linux

El comando `man` es el manual integrado en la terminal de Linux. Permite acceder a documentación sobre funcionalidades de la terminal, syscalls del kernel, funciones de la librería estandar, archivos de configuración, entre otros. Pueden usar el comando `man man` para obtener

más información sobre el mismo. Cada categoría (o página) de documentación está numerada (1: comandos de terminal, 2: syscalls, etc.). En nuestro caso, nos interesa la página 3 ya que contiene las funciones de la `libc` (biblioteca estándar de C). Si una entrada existe en varias páginas, hay que indicar el número explícitamente.

```
man printf    # comando de shell (sección 1)
man 3 printf  # función de la libc (sección 3)
```

Ejemplo de uso:
Veamos un ejemplo ejecutando `man strcpy`.
La entrada correspondiente está dividida en distintas secciones:

- **NAME:** Lista los nombres que corresponden a esta entrada. A veces agrupa funciones relacionadas o variaciones de una misma.

  ```
  NAME
     strcpy, strncpy - copy a string
  ```

- **SYNOPSIS:** En esta sección podemos distinguir:
  - Las dependencias necesarias (string.h).
  - La firma de la función (parámetros y tipo de dato que devuelve) que debemos respetar para su uso en nuestros programas.

  ```
  SYNOPSIS
     #include <string.h>

     char *strcpy(char *dest, const char *src);

     char *strncpy(char *dest, const char *src, size_t n);
  ```

- **DESCRIPTION:** Explicación en lenguaje natural del funcionamiento y motivación de la función.

  ```
  DESCRIPTION
    The  strcpy() function copies the string pointed to by src, including
    the terminating null byte ('\0'), to the buffer pointed to by dest.
    The strings may not overlap, and the destination string dest must be
    large enough to receive the copy. Beware of buffer overruns!
    (See BUGS.)
    ...
  ```

- **RETURN VALUE:** Qué devuelve la función en una ejecución correcta (o en caso de error, según corresponda).

  ```
  RETURN VALUE
     The strcpy() and strncpy() functions return a pointer to the
     destination string dest.
  ```

- **ATTRIBUTES:** Muestra atributos de seguridad y concurrencia de la función, por ejemplo la siguiente tabla donde *see attributes(7)* indica que con `man 7 attributes` podemos obtener más información sobre los términos que la componen:

```
For an explanation of the terms used in this section, see attributes(7)
...
```

- **CONFORMING TO:** Estándares (POSIX, ISO C, etc.) que cumple la implementación actual.

```
CONFORMING TO
   POSIX.1-2001, POSIX.1-2008, C89, C99, SVr4, 4.3BSD.
```

- **NOTES:** Información adicional que no encaja en otras secciones.

```
NOTES
   Some  programmers  consider  strncpy() to be inefficient
   and error prone.  If the programmer knows (i.e., includes
   code to test!) that the size of dest is greater than the
   length of src, then strcpy() can be used.
    ...
```

- **BUGS:** ¿Por qué existe esta sección en funciones de la `libc`? En muchos casos no son "bugs" de implementación, sino limitaciones o peligros del diseño original. Cambiar su comportamiento rompería miles de programas que dependen de él. Se documentan aquí para advertir al programador y, en algunos casos, recomendar funciones alternativas más seguras. En el caso de `strcpy()`, suele recomendarse `strncpy()` o `strlcpy()` (en BSD).

```
BUGS
   If the destination string of a strcpy() is not large enough,
   then anything might happen.
   Overflowing fixed-length string buffers is a favorite cracker
   technique for taking complete control of the machine.
   ...
```

- **SEE ALSO:** Referencias a funciones relacionadas o documentación complementaria.

```
SEE ALSO
   bcopy(3), memccpy(3), memcpy(3), memmove(3), stpcpy(3),
   stpncpy(3), strdup(3), string(3), wcscpy(3), wcsncpy(3)
```

- **COLOPHON:** Contiene:
  - El paquete al que pertenece la página (man-pages en Linux).
  - El proyecto de documentación que la mantiene.
  - Un enlace al sitio web y a la forma de reportar bugs en la página de manual.

```
COLOPHON
   This page is part of release 5.10 of the Linux man-pages project.
   A description of the project, information about reporting bugs,
   and the latest version of this page, can be found at
   https://www.kernel.org/doc/man-pages/.
```

Ejercicio de Debugging

Parte 1: El heladero Pedro quiere desarrollar una plataforma de compra y venta de helados usando el lenguaje C. Veremos algunos problemas que se encontró durante su trabajo en la aplicación (aún en desarrollo, va lento porque Pedro nunca aprendió a usar GDB). Observen la siguiente pieza de código, se trata de una función que Pedro implementó para dar de alta usuarios.

Snippet 57: Parte 1

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

typedef struct{
    char nombre[5];
    uint8_t habilitado; // bool
    int saldo;
    int gustoDeHeladoFavorito;
} usuario_t;

void habilitarUsuario(usuario_t *usuario){
    usuario->habilitado = 1;
    for(int i = 0; i <= 5; i++){
        usuario->nombre[i] = '\0';
    }
    usuario->saldo = 0;
}

int main(void){
    usuario_t *nuevoUsuario = malloc(sizeof(usuario_t));
    habilitarUsuario(nuevoUsuario);

    if(nuevoUsuario->habilitado){
        printf("Usuario habilitado con éxito!\n");
    } else{
        printf("Error al habilitar usuario.\n");
    }

    free(nuevoUsuario);
    return 0;
}
```

Compilen con información de debug activa y ejecuten el programa. Pedro espera poder habilitar a los usuarios correctamente, ¿cuál es la salida obtenida?

Utilicen GDB para determinar el origen del problema. Una vez detectado, corrijanlo y asegurense de que los usuarios pueden ser dados de alta correctamente.

Pista: analicen qué sucede dentro de la función `habilitarUsuario`.

Parte 2: Con el sistema de alta de usuarios listo, lo siguiente que desarrolló es la funcionalidad de comprar. Es necesario que el saldo del comprador sea suficiente, por lo que Pedro empezó programando la función `aumentarSaldo`.

Tomen esta pieza de código y reemplacen `habilitarUsuario` por su versión corregida.

Snippet 58: Parte 2

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

typedef enum{
    MENTA,
    DDL,
    OREO,
    LIMON
} gustoDeHelado_t;

char* gustosDeHelado[4]= {"MENTA", "DDL", "OREO", "LIMON"};

typedef struct{
    uint32_t precio;
    gustoDeHelado_t gusto;
} helado_t;

helado_t heladoDDL = {.precio = 5, .gusto = DDL};

typedef struct{
    char nombre[5];
    uint8_t habilitado; // bool
    int saldo;
    int gustoDeHeladoFavorito;
} usuario_t;

void habilitarUsuario(usuario_t *usuario){
    usuario->habilitado = 1;
    for(int i = 0; i <= 5; i++){
        usuario->nombre[i] = '\0';
    }
    usuario->saldo = 0;
}

void aumentarSaldo(int saldoDelUsuario, int cantidad){
    saldoDelUsuario += cantidad;
}

void comprarHelado(usuario_t *usuario, helado_t* *helado){
    if(usuario->saldo >= 5){
        usuario->saldo -= 5;
        *helado = &heladoDDL;
        printf("Helado comprado con exito\n");
    }else{
        printf("El usuario no tiene saldo suficiente\n");
    }
}
```

Snippet 59: Parte 2 cont.

```c
int main(void){
    usuario_t *nuevoUsuario = malloc(sizeof(usuario_t));
    habilitarUsuario(nuevoUsuario);

    if(nuevoUsuario->habilitado){
        printf("Usuario habilitado con éxito!\n");
        aumentarSaldo(nuevoUsuario->saldo, 10);

        helado_t* helado = NULL;
        comprarHelado(nuevoUsuario, &helado);
        nuevoUsuario->gustoDeHeladoFavorito = helado->gusto;

        printf("Usuario creado con éxito. \n")
        printf("Su saldo es de %d pesos y su gusto favorito es %s.\n",
               nuevoUsuario->saldo,
               gustosDeHelado[nuevoUsuario->gustoDeHeladoFavorito]
        );
    } else{
        printf("Error al habilitar usuario %d\n", nuevoUsuario->habilitado);
    }

    free(nuevoUsuario);
    return 0;
}
```

Luego vuelvan a compilar el programa con la información de debug activa y ejecutenlo. ¿Cuál es la salida del programa?

Ejecuten el programa compilado usando Valgrind. ¿Qué errores de memoria detecta?

Usen GDB para encontrar el origen del problema, corrijanlo y asegurense de que puede agregarse saldo a los usuarios correctamente y realizar una compra.

Ejecuten el programa corregido con Valgrind y verificar si aún presenta errores de memoria.

Parte 3: Consciente de que no sabe bien lo que está haciendo, Pedro nos envió la versión más actualizada de su implementación y nos pide que la revisemos para asegurarnos de que todo funciona como debe. Probablemente debido al cansancio del trabajo en la heladería, el código que nos envió no tiene corregidos los errores antes mencionados y tampoco nos explicó qué funcionalidades nuevas implementó.

Snippet 60: Parte 3

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

typedef enum{
    MENTA,
    DDL,
    OREO,
    LIMON
} gustoDeHelado_t;
```

Snippet 61: Parte 3 cont.

```
typedef enum{
    ABIERTO,
    CERRADO
} estadoDelLocal;

char* gustosDeHelado[4]= {"MENTA", "DDL", "OREO", "LIMON"};

typedef struct{
    uint32_t precio;
    gustoDeHelado_t gusto;
} helado_t;

helado_t heladoDDL = {.precio = 5, .gusto = DDL};

typedef struct{
    char nombre[5];
    uint8_t habilitado; // bool
    int saldo;
    int gustoDeHeladoFavorito;
} usuario_t;

void habilitarUsuario(usuario_t *usuario){
    usuario->habilitado = 1;
    for(int i = 0; i <= 5; i++){
        usuario->nombre[i] = '\0';
    }
    usuario->saldo = 0;
}


void aumentarSaldo(int saldoDelUsuario, int cantidad){
    saldoDelUsuario += cantidad;
}

void comprarHelado(usuario_t *usuario, helado_t* *helado){
    if(usuario->saldo >= 5){
        usuario->saldo -= 5;
        *helado = &heladoDDL;
        printf("Helado comprado con exito\n");
    }else{
        printf("El usuario no tiene saldo suficiente\n");
    }
}
```

Snippet 62: Parte 3 cont.

```
int main(void){
  usuario_t *nuevoUsuario = malloc(sizeof(usuario_t));
  habilitarUsuario(nuevoUsuario);
  int local_abierto;

  if(local_abierto == ABIERTO){
      if(nuevoUsuario->habilitado){
          printf("Usuario habilitado con éxito!\n");
          aumentarSaldo(nuevoUsuario->saldo, 10);

          helado_t* helado = NULL;
          comprarHelado(nuevoUsuario, &helado);
          nuevoUsuario->gustoDeHeladoFavorito = helado->gusto;

          printf("Usuario creado con éxito. \n")
          printf("Su saldo es de %d pesos y su gusto favorito es %s.\n",
                 nuevoUsuario->saldo,
                 gustosDeHelado[nuevoUsuario->gustoDeHeladoFavorito]
          );
      } else{
        printf("Error al habilitar usuario %d\n", nuevoUsuario->habilitado);
      }
  }
  free(nuevoUsuario);
  return 0;
}
```

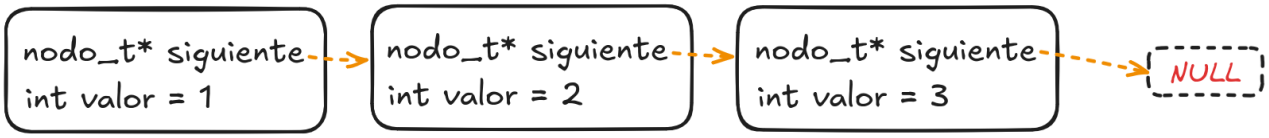
Exploren el código, identifiquen los errores y corríjanlos.

Anexo: Ejemplo integrador

Intelligence takes chance with limited data in an arena where mistakes are not only possible but also necessary.

Como ejemplo integrador vamos a definir algunas operaciones sobre listas enlazadas en C.

Una lista enlazada es una estructura de datos donde cada nodo contiene un valor y un puntero al siguiente nodo de la lista. La lista $[1, 2, 3]$ se puede pensar de la siguiente manera:



El primer nodo de una lista se denomina la **cabeza** de la lista. Una lista se recorre comenzando por la cabeza y avanzando al siguiente nodo hasta que no haya un siguiente nodo al cual avanzar.

Primero tenemos que definir las estructuras que van a representar a la lista enlazada en memoria. Para esto vamos a definir dos structs, `lista_t` para representar el la lista en sí misma y `nodo_t` para los elementos que forman la lista.

Snippet 63: declaración de estructuras de lista enlazada

```c
typedef struct nodo_s {
    // hay que definir struct nodo_s* acá porque el tipo nodo_t
    // no está declarado.
    struct nodo_s* siguiente;
    int valor;
} nodo_t;

typedef struct lista_s {
    nodo_t* cabeza;
} lista_t;
```

Una `lista_t` está definida por el puntero a la cabeza de la lista, es decir el primer nodo de la lista. Luego en cada `nodo_t` se almacena el valor que este representa y el puntero al siguiente nodo en la lista (si es que dicho nodo existe).

Primero vamos a definir funciones pare crear una lista vacía y un nuevo nodo.

Snippet 64:

```c
lista_t* crear_lista_vacia() {
    lista_t* lista_vacia = malloc(sizeof(lista_t));
    lista_vacia->cabeza = NULL; // El operador x->y es equivalente a (*x).y
    return lista_vacia;
}

nodo_t* crear_nuevo_nodo(int valor) {
    nodo_t* nuevo_nodo = malloc(sizeof(nodo_t));
    nuevo_nodo->siguiente = NULL;
    nuevo_nodo->valor = valor;
    return nuevo_nodo;
}
```

Estas funciones crean una lista vacía y un nodo respectivamente, retornan un puntero a la estructura creada. Ambas estructuras se encuentran en el *heap*.

¿Por qué tenemos que reservar memoria con `malloc` para crear una lista vacía o un nuevo nodo? Pensemos en esta otra implementación (errónea) de `crear_nuevo_nodo`:

Snippet 65:

```
nodo_t* crear_nuevo_nodo_incorrecto(int valor) {
    nodo_t nuevo_nodo; // Esta estructura se almacena en el stack.
    nuevo_nodo.siguiente = NULL;
    nuevo_nodo.valor = valor;
    return &nuevo_nodo; // Retornamos una dirección de memoria del stack.
}
```

En esta implementación el nuevo nodo se define en el stack. Esa porción de memoria es válida durante todo el tiempo de vida del *stack frame*, es decir hasta el momento que retorna la función. Luego cualquier acceso a ese puntero estaría escribiendo o leyendo en una dirección de memoria que probablemente le corresponda al *stack frame* de otra función. **Evitar este tipo de errores en C queda a criterio del programador. Hay instancias en las que se pueden cometer errores de memoria similares a este y que el programa aparente funcionar bien.**

Ahora con estas funciones podemos definir otras operaciones típicas en listas enlazadas, como agregar elementos al final:

Snippet 66:

```
void insertar_al_final(lista_t* lista, int valor) {
    nodo_t* actual = lista->cabeza;
    nodo_t* a_insertar = crear_nuevo_nodo(valor);

    // Caso lista vacia.
    // Intentar desreferenciar el puntero NULL causa un
    // Segmentation Fault (SIGSEGV).
    if (actual == NULL) {
        lista->cabeza = a_insertar;
        return;
    }

    while (actual->siguiente != NULL)
        actual = actual->siguiente;

    actual->siguiente = a_insertar;
}
```

A diferencia de otros lenguajes en C no hay excepciones[6]. Una convención que se suele usar es que una función retorne un código de error indicando si la operación tuvo éxito o no. Por ejemplo podemos escribir una función que borre la cabeza de una lista. Retorna 0 si pudo borrar la cabeza de la lista y retorna -1 si la lista ya estaba vacía:

[6]Esto quiere decir que operaciones que en otros lenguajes, como Python, levantan un error en C pasan desapercibidas a menos que se haga un chequeo explícito. Por ejemplo en Python si $lista = [1, 2, 3]$, la operación $lista[5]$ levanta una excepción. En C nada nos impide indexar un array fuera de rango.

Snippet 67:

```c
int eliminar_cabeza(lista_t* lista) {
    nodo_t* actual = lista->cabeza;
    if (actual == NULL)
        return -1;
    lista->cabeza = actual->siguiente;
    free(actual);
    return 0;
}
```

Notar que esta función una vez que acomoda la lista con la nueva cabeza, llama a `free` con el puntero a la anterior cabeza. Esto es necesario pues ese es el último momento en el que el puntero a la cabeza original es alcanzable dentro de algún *scope*. En C si no se indica explícitamente, nunca se libera memoria reservada con `malloc`. Hay que tener cuidado de no llamar a `free` antes de estar seguros de que nadie va a querer usar la porción de memoria referenciada. En programas más complejos esto se puede solucionar, por ejemplo, manteniendo un contador de referencias activas.

Siguiendo este razonamiento podemos escribir una función que borre la lista completa:

Snippet 68:

```c
void eliminar_lista(lista_t* lista) {
    nodo_t* actual = lista->cabeza;
    while (actual != NULL) {
        nodo_t* siguiente = actual->siguiente;
        free(actual);
        actual = siguiente;
    }
    free(lista);
}
```

Como último ejemplo vamos a definir la función `map` sobre listas de números enteros. La función `map` recibe como parámetro un puntero a una `lista_t` y un puntero a una función de la forma `int (*operacion)(int)`. Esto quiere decir un puntero a una función, llamada `operacion` en este caso, que toma como único argumento un `int` y retorna un `int`. Luego `map` modifica la lista aplicando la función `operacion` a cada uno de sus elementos. Por ejemplo:

$$f(x) = 2x$$

$$map([1, 2, 3, 4, 5], f) = [2, 4, 6, 8, 10]$$

Snippet 69:

```c
void map(lista_t* lista, int (*operacion)(int)) {
    nodo_t* actual = lista->cabeza;
    while (actual != NULL) {
        actual->valor = operacion(actual->valor);
        actual = actual->siguiente;
    }
}
```

Ejemplo de uso:

Snippet 70:

```c
int duplicar(int x) {
    return x*2;
}

int main() {
    lista_t* mi_lista = crear_lista_vacia();
    insertar_al_final(mi_lista, 1);
    insertar_al_final(mi_lista, 2);
    insertar_al_final(mi_lista, 3);
    insertar_al_final(mi_lista, 4);
    // En este momento mi_lista = [1,2,3,4]
    map(mi_lista, &duplicar);
    // En este momento mi_lista = [2,4,6,8]
    eliminar_lista(mi_lista);
    return 0;
}
```

Como ejercicio opcional pueden probar implementar la función `filter` que elimina de `lista` todos los elementos en los que `criterio` evalúa a 0. En la página siguiente hay una posible implementación, de todos modos recomendamos que intenten escribir una propia antes de consultarla.

```c
void filter(lista_t* lista, int (*criterio)(int));
```

Un posible criterio podría ser:

Snippet 71:

```c
int es_par(int x) {
    if (x % 2 == 0)
        return 1;
    else
        return 0;
}
```

Esta es una posible solución al problema, no es la única, ni la más compleja, ni la más sencilla:

Snippet 72:

```
void filter(lista_t* lista, int (*criterio)(int)) {
    nodo_t** cabeza_actual = &lista->cabeza;
    while (*cabeza_actual != NULL) {
        if (!criterio((*cabeza_actual)->valor)) {
            nodo_t* a_borrar = *cabeza_actual;
            *cabeza_actual = (*cabeza_actual)->siguiente;
            free(a_borrar);
        } else {
            cabeza_actual = &(*cabeza_actual)->siguiente;
        }
    }
}
```

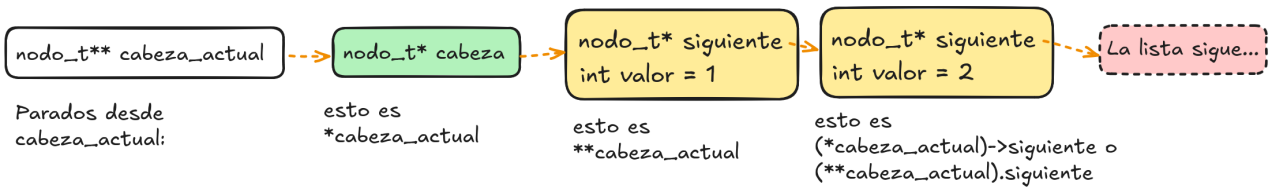
Parece bastante razonable, si la cabeza de la lista no cumple con el criterio, la borro y continúo operando con la sub-lista que me queda. No importa que esté en el primer elemento, el décimo o el último. Pero... ¿Qué está pasando con `cabeza_actual`?

Primero empezamos por el tipo, `cabeza_actual` es un puntero a un puntero a `nodo_t`. Esto quiere decir que es una dirección de memoria que apunta a otra dirección de memoria, y esta última apunta a un *struct* de tipo `nodo_t`. Al principio se inicializa en la dirección de memoria en la que reside el campo `cabeza` de la lista, este campo es de tipo `nodo_t*`, por lo que nos queda algo de tipo `nodo_t**`.

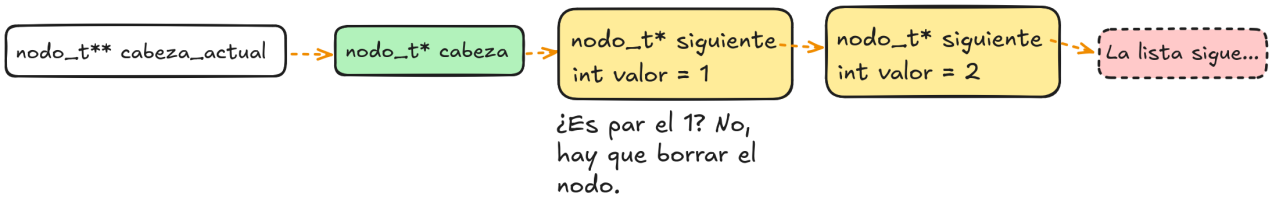
Vamos a ver en detalle las primeras iteraciones para la lista $[1, 2, 3, 4, 5, ..., k]$ para el predicado `es_par` dado como ejemplo en el enunciado del ejercicio.

La convención de coloreo en los dibujos para los tipos representados es:

`Paso 0:` Antes de entrar al ciclo, inicializamos `cabeza_actual` en el principio de la lista.



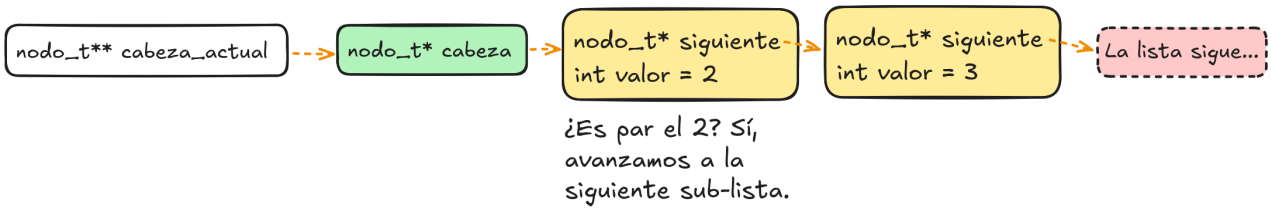
`Paso 1:` Borrar el 1 de la lista y avanzar al siguiente elemento.



Antes de pasar al paso 2 está sucediendo esto:

```
nodo_t* a_borrar = *cabeza_actual;
*cabeza_actual = (*cabeza_actual)->siguiente;
free(a_borrar);
```

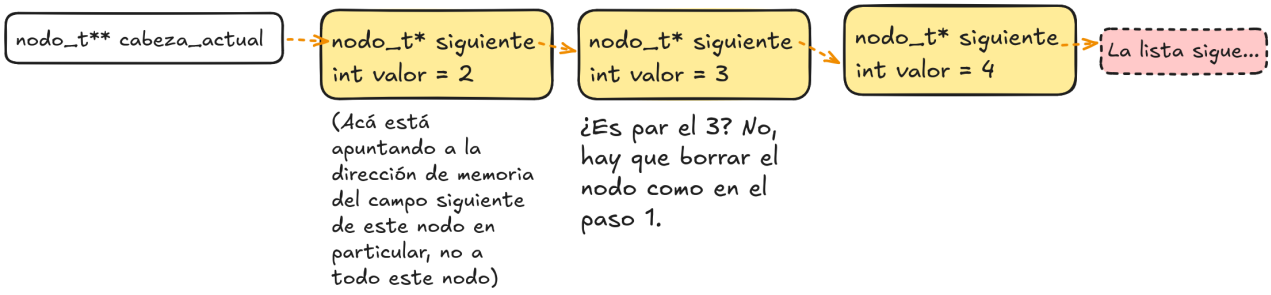
`Paso 2:` Preservar el 2 y avanzar al siguiente elemento.



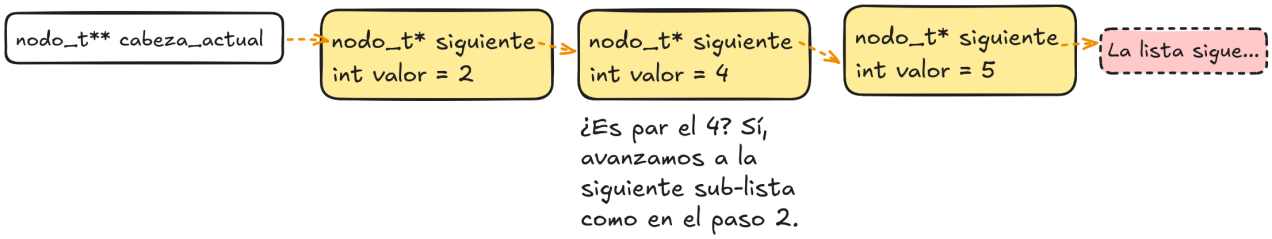
Antes de pasar al paso 3 está sucediendo esto:

```
cabeza_actual = &(*cabeza_actual)->siguiente;
```

`Paso 3:` Borrar el 3 de la lista y avanzar al siguiente elemento.



`Paso 4:` Preservar el 4 y avanzar al siguiente elemento.



Esto continúa hasta llegar al final de la lista, donde `*cabeza_actual` va a ser igual a `NULL` y no se entra al ciclo.

Referencias

Most believe that a satisfactory future requires a return to an idealized past, a past which never in fact existed.

- Kernighan, Ritchie - The C Programming Language, 2nd Edition.
- Seacord, Robert - Effective C: An Introduction to Professional C Programming
- C reference: https://en.cppreference.com/w/c
- GNU C reference: https://gcc.gnu.org/onlinedocs/gcc/
- Valgrind documentation: https://valgrind.org/docs/manual/quick-start.html
- Cdecl: https://cdecl.org/
- C programming: https://www.learn-c.org/