# Algoritmos y Estructuras de Datos II

TALLER - 18 de abril 2024

# Laboratorio 4: Punteros y TADs

- Revisión 2024: Marco Rocchietti

## Objetivos

1. Comenzar a trabajar con punteros en C
2. Simular variables de salida con punteros
3. Comprender uso de punteros para mayor desempeño
4. Administración de memoria dinámica (`malloc()`, `calloc()`, `free()`)
5. Llevar a lenguaje C los conceptos de TAD estudiados en el Teórico-Práctico
6. Comprender conceptos de encapsulamiento vs acoplamiento
7. Comprender concepto de implementación opaca

## Preliminares: Punteros 101

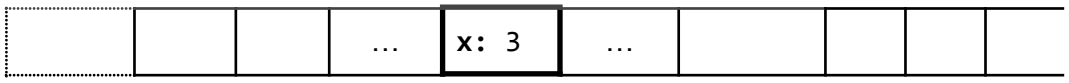
El objetivo del primer ejercicio es adquirir un entrenamiento básico y comprender el funcionamiento de punteros en C.

Un puntero es un tipo de variable especial que guarda una dirección de memoria. En C se denotan los punteros usando el símbolo **`*`** . Es decir que una variable **`p`** declarada como **`int *p;`** es del tipo *puntero a* `int`.

Para el manejo de punteros contamos con dos operadores unarios básicos, el operador de referenciación y el de desreferenciación.

### Operación de referenciación (&)

Este operador obtiene la dirección de memoria de una variable. También se lo conoce como operador de dirección (*address operator*). Si se tiene una variable entera **`x`** declarada como **`int x=3;`** entonces la expresión **`&x`** retornará la dirección de memoria donde está alojado el contenido de la variable **`x`**.



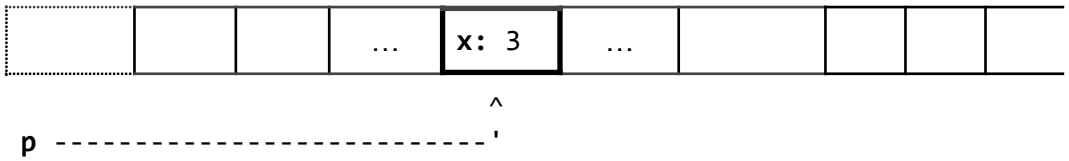

(memoria)

Particularmente la expresión **`&x`** en este caso es del tipo **`int*`**, es decir del tipo puntero a **`int`**. Por lo tanto, se puede hacer lo siguiente:

```
int x=3;
int *p;
p = &x;
```

notar que la asignación es correcta porque **`p`** y **`&x`** tienen el mismo tipo. En este ejemplo entonces el

puntero **p** guarda la dirección de memoria de **x** y podemos decir que ***p apunta a x.***




## Operación de desreferenciación (*)

Obtiene el **valor** de lo *apuntado* por el puntero. También se lo conoce como el operador de indirección (*indirection operator*). Se lo puede pensar como una operación de inspección ya que accede al valor alojado en una dirección de memoria. Si se tiene una variable de tipo **int*** llamada **p**, entonces la expresión ***p** retornará el valor entero que se aloja en la dirección de memoria que contiene **p**. En el ejemplo de más arriba ***p** devuelve **3**.

Además si se utiliza ***p** del lado izquierdo de una asignación:

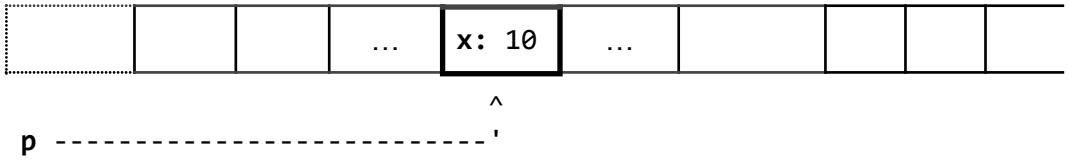
```
*p = <expresión>;
```

la asignación escribirá el resultado de la expresión en la dirección de memoria apuntada por **p**, por lo que se cambia el valor contenido en esa dirección de memoria (sin modificar a **p**, que seguirá apuntando al mismo lugar). Por ejemplo, si se hace la asignación

```
int x=3;
int *p;

p = &x;
*p = 10;
```

entonces sucede que




Notar que se cambió el valor de la variable **x** de manera indirecta usando el puntero **p**.

Cabe aclarar que cuando se declara la variable de tipo puntero **int *p;** el símbolo ***** no actúa como operador sino que simplemente indica que la variable **p** se declara como puntero.

Para pensar: ¿Qué valor tendrá la variable **y** luego de ejecutar el siguiente código?

```
int x = 3;
int y = 10;
y = *(&x);
```

En el *Laboratorio 1* se utilizaba la función **fscanf()** ¿Qué parámetros tomaba dicha función y cuál era el tipo de cada uno?

## Constante nula de punteros (`NULL`)

Siempre es buena idea dar un valor inicial a las variables apenas se declaran. Para punteros existe en C la constante `NULL` que representa una dirección de memoria nula, en la cual no se puede leer ni escribir. Esta constante es una macro definida en los *headers* de `stdlib.h` como la dirección de memoria `0`.

Recordar que si no se inicializa una variable esta puede contener cualquier valor, en el caso de enteros podría ser un número muy grande y extraño (o quizás un inofensivo 4) y en el caso de punteros puede tener asignada una dirección de memoria que en caso de querer escribir o leer de ella generaría problemas. Por ejemplo en el siguiente programa:

```
int *p;
*p = 3;
```

es fácil imaginar que esto podría generar que el programa termine con un error (violación de segmento - *segmentation fault*) pero podría ser peor. Puede suceder que por azar en `p` se encuentre la dirección de memoria de otra variable del programa y la modifiquemos, generando un **BUG** muy difícil de rastrear.

En esta otra versión:

```
int *p=NULL;
*p = 3;
```

el programa siempre va a fallar, y eso es bueno.

Claramente los ejemplos de arriba son errores que saltan a la vista pero sirven para ilustrar situaciones en la que no es tan obvio que se usa un puntero sin inicializar, pero el efecto es el mismo.

# **Ejercicio 1**: Introducción de punteros

La tarea de este ejercicio consiste en completar el archivo `main.c` de manera tal que la salida del programa por pantalla sea la siguiente:

```
x = 9
m = (100, F)
a[1] = 42
```

Las restricciones son:

- No usar las variables `x`, `m` y `a` en la parte izquierda de alguna asignación.
- Se pueden agregar líneas de código, pero no modificar las que ya existen.
- Se pueden declarar hasta 2 punteros.

Recordar siempre inicializar los punteros en `NULL`:

*Se mostró en el taller cómo hacer debugging de un programa mediante GDB. Esta herramienta también es útil para entender "qué está pasando" con el código cuando se ejecuta. Se recomienda compilar con los símbolos de debugging y poner breakpoints para imprimir los valores de las variables del programa.*

*En gdb también se pueden imprimir valores como:*

- *Dirección de memoria de una variable:* **`print &x`**
- *El valor que hay en la memoria apuntada por un puntero:* **`print *p`**

## **Ejercicio 2**: Simulando procedimientos

En el lenguaje de programación del teórico-práctico se usan funciones y procedimientos que tienen una naturaleza distinta a las funciones de C. Particularmente en C sólo existen las funciones y a veces llamamos “procedimientos” a aquellas funciones que devuelven algo del tipo **`void`** (que es el tipo “vacío” de C, o en otras palabras que no devuelve nada). Se debe entonces traducir el siguiente programa tratando de simular el procedimiento **`absolute()`** usando funciones de C :

```
proc absolute(in x : int, out y : int)
  if x >= 0 then
    y := x
  else
    y := -x
  fi
end proc

fun main() ret r : int
  var a : int
  a := -10
  absolute(a, res)
  {- supongamos que print() muestra el valor de una variable -}
  print(res)
  {- esta última asignación es análoga a `return EXIT_SUCCESS;` -}
  r := 0
end fun
```

¿Qué valor se mostraría al ejecutar la función **`main()`** del programa anterior?

**a)** Abrir el archivo **`abs1.c`** y traducir a lenguaje C usando el siguiente prototipo para **`absolute()`**:

```
void absolute(int x, int y) {
  ()
}
```

luego compilar con el siguiente comando:

```
$ gcc -Wall -Werror -pedantic -std=c99 abs1.c -o abs1
```

(notar que no se utiliza **`-Wextra`** sólo por esta vez) y ejecutar

```
$ ./abs1
```

¿Qué valor se muestra por pantalla? ¿Coincide con el programa en el lenguaje del teórico? Responder en un comentario al final de **`abs1.c`**.

**b)** Ahora abrir el archivo **`abs2.c`** que utiliza el siguiente prototipo de **`absolute()`**:

```
void absolute(int x, int *y) {
  (⋮)
}
```

Pensar qué modificaciones son necesarias hacer dentro de las funciones **`absolute()`** y **`main()`** respecto a la implementación realizada en **`abs1.c`** para trabajar con el nuevo prototipo. Además se debe lograr que el programa en C simule el comportamiento del programa original en lenguaje del teórico-práctico. Implementar esas modificaciones en **`abs2.c`** y luego compilar:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 abs2.c -o abs2
```

y por último ejecutar

```
$ ./abs2
```

¿Se muestra el valor correcto? (en caso contrario revisar hasta lograr que sí lo haga)

| | |
|---|---|
| ❗ | *Notar que conceptualmente en ningún caso los programas son equivalentes puesto que las funciones y procedimientos del lenguaje del teórico tienen una naturaleza completamente distinta a las de C.* |

**c)** Para pensar:

- ¿El parámetro **`int *y`** de `absolute()` es de tipo **`in`**, de tipo **out** o de tipo **`in/out`**?
- ¿Qué tipo de parámetros tiene disponibles C para sus funciones?
  - Parámetros **`in`**
  - Parámetros **`out`**
  - Parámetros **`in/out`**

Responder al final de **`abs2.c`** como un comentario en el código

**d)** En un nuevo archivo **`swap.c`** implementar una traducción del programa que intercambia valores:

```
proc swap(in/out a : int, in/out b : int)
    var aux : int
    aux := a
    a := b
    b := aux
end proc

fun main() ret r : int
  var x, y : int
  x := 6
  y := 4
  print(x, y)
  swap(x, y)
  print(x, y)
  r := 0
end fun
```

# **Ejercicio 3**: Aprovechando punteros para eficiencia

La intención del ejercicio es explorar la conveniencia de utilizar punteros para que los intercambios (*swaps*) sean más eficientes.

Completar el archivo **`sort.c`** copiando código del *Ejercicio 3* del Laboratorio 3 realizando las modificaciones pertinentes para trabajar con arreglos de punteros a estructuras. Van a notar que en la nueva versión de **`player.h`** se redefinió al tipo **`player_t`**,

```
typedef struct _player_t {
      char name[100];
      char country[4];
      unsigned int rank;
      unsigned int age;
      unsigned int points;
      unsigned int tournaments;
} * player_t;
```

siendo entonces ahora **un puntero** a una estructura **`struct _player_t`**.

Notar que la función **`main()`** muestra la cantidad de tiempo empleado en la ordenación.

*¿Funciona más rápido la versión con punteros? ¿Por qué ahora son más eficientes los intercambios?*

# Preliminares: Punteros++

## Los punteros son también variables

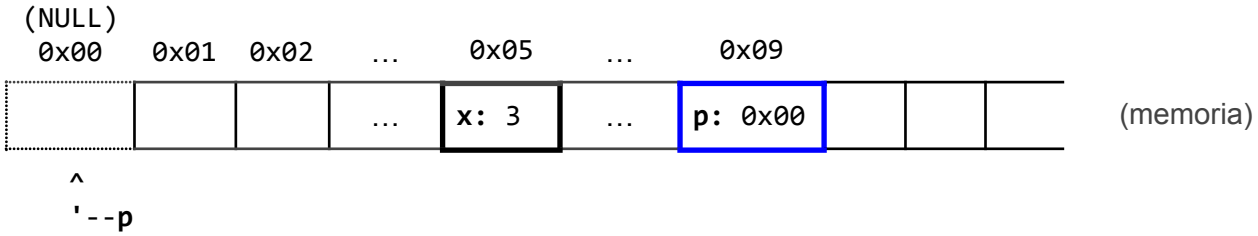
Se vio que un puntero es un tipo de variable especial que guarda una dirección de memoria. La memoria se puede pensar como un gran arreglo, y en ese sentido una dirección de memoria es un índice. A estos índices los escribiremos en base hexadecimal (0, 1, 2, ..., 9, A, B, C, ..., F) que es la base utilizada normalmente para referirse a direcciones de memoria. Volvamos a los ejemplos básicos

```
int x=3;
int *p;
p = &x;
```

Anteriormente se ilustró el resultado de este programa con el siguiente esquema:

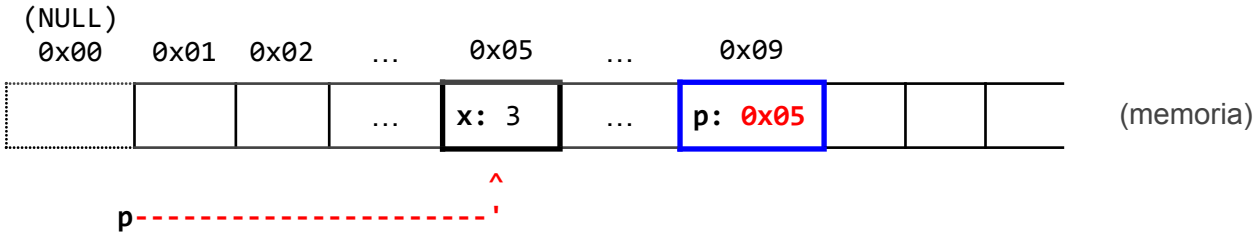

esto, aunque es bastante didáctico, pareciera que deja a **`p`** por fuera de la memoria. El puntero **p** también es una variable y por lo tanto también tiene su lugar en la memoria. Veamos cómo sería la ejecución del programa en sus distintas etapas (en este caso inicializaremos en **`NULL`** al puntero):

```
int x=3;
int *p=NULL;
```




notar que ahora en el esquema se ve que **p** tiene su lugar en la memoria e inicialmente vale `0x00` (el prefijo `0x` indica que es un número hexadecimal) lo cual significa que **p** apunta a **NULL**. Sigamos con el programa:

```
p = &x;
```

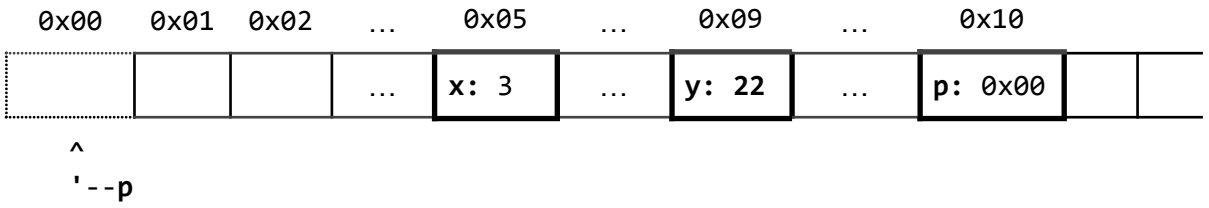



la expresión **&x** hace referencia a la dirección de memoria **0x05**, entonces luego de la asignación, el puntero **p** apunta a la dirección de memoria **0x05**, es decir que **p** simplemente tiene ese valor asignado. Notar que **p**, al ser una variable, su valor también está en algún lugar de la memoria (en la dirección **0x09** en este caso). Veamos ahora un ejemplo más complejo,

```
int x=3, y=22;
int *p=NULL;
p = &x;
y = *p;
*p = 7;
```

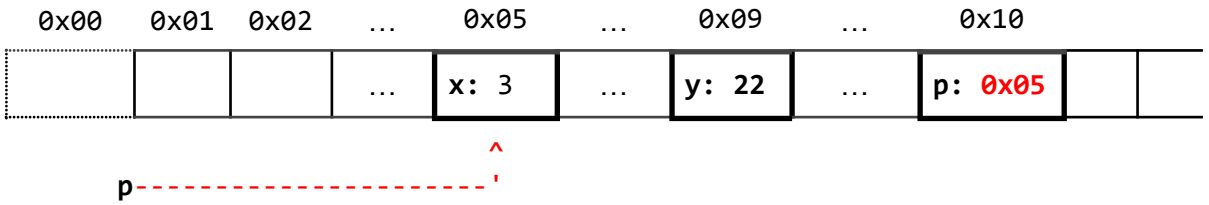
nuevamente lo ejecutamos por etapas:

```
int x=3, y=22;
int *p=NULL;
```

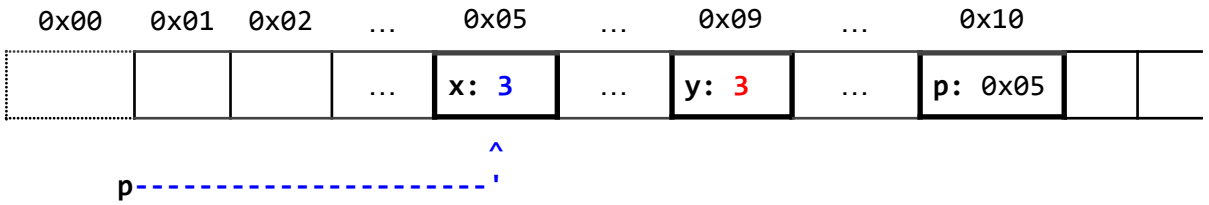



Ahora se muestra como cambia la memoria para las siguientes tres asignaciones:

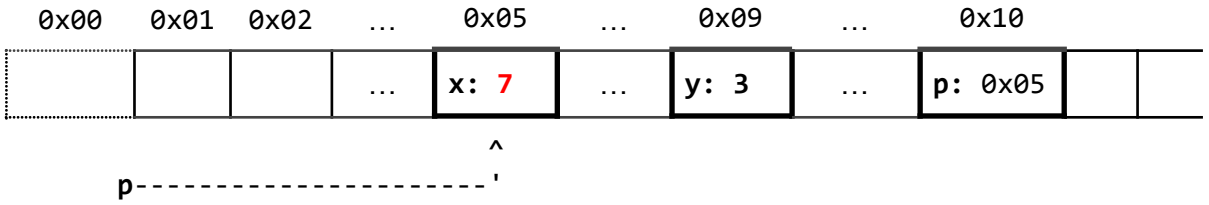
```
p = &x;
```




```
y = *p;
```




```
*p = 7;
```




la primera asignación de **p** hace que apunte a la dirección de **x** que es **0x05**, luego a la variable **y** se le asigna el contenido que hay en la dirección de memoria **0x05** que es (en ese momento) el valor **3**, luego se cambia el contenido de la memoria en la dirección **0x05** y se escribe el valor **7**.

Para pensar: ¿Cuál sería el resultado de la expresión **&p** ?

## Visualizando direcciones de memoria

Los valores de las variables del tipo puntero (las direcciones de memoria) se pueden visualizar. Por lo general esto no se hace, salvo a veces para hacer *debug*. La manera es usando **printf()** con **%p**:

```
int *p=NULL;
int a=55;
p = &a;
printf("La dirección de memoria apuntada por p es: %p", (void *) p);
```

Notar el *casteo* que se le realiza a **p** cuando se lo pasa como parámetro a **printf()**. Cuando antes de una expresión se introduce un tipo entre paréntesis, significa que la expresión se va a convertir al tipo indicado. Por ejemplo **(float) 2** hace que el resultado se interprete como **2.0f**, otro ejemplo puede ser **(int) 1.5** que hace que el resultado sea el entero **1**. En este caso se convierte a **p**, que es un **int***, a un **void*** o sea un puntero a **void** lo cual es simplemente una dirección de memoria pura, puesto que un puntero a **void** no se puede desreferenciar.

La salida del programa va a ser un número en hexadecimal (con prefijo `0x`…), por ejemplo:

```
La dirección de memoria apuntada por p es: 0x7ffd15a1ac60
```

Otra forma con la cual se puede ver el valor decimal de una dirección de memoria es hacer:

```
int *p=NULL;
int a=55;
p = &a;
printf("El índice de memoria alojado en p es: %lu", (uintptr_t) p);
```

aquí se castea el puntero a un entero sin signo que es el tipo **`uintptr_t`** definido en **`<stdint.h>`**. Este tipo es lo suficientemente grande para guardar direcciones de memoria, pero la solución no es muy estándar ya que el comodín **`"%lu"`** podría fallar en algunos sistemas ya que **`printf()`** espera algo del tipo **`long unsigned int`** y podría no ser equivalente a **`uintptr_t`**. La salida sería algo como:

```
El índice de memoria alojado en p es: 140724966370400
```

## Operadores de indexación y flecha

En C además para las variables de tipo puntero se puede usar las operaciones de *indexación* y el *operador flecha* (`->`):

- Indexación (p**`[n]`**): Permite obtener el valor que hay en la memoria moviéndose **`n`** lugares hacia adelante (de manera alineada al tipo de datos del puntero) desde la dirección de memoria guardada en **`p`**. Entonces por ejemplo **`p[0]`** es equivalente a **`*p`**. Cuando se indexa un puntero se debe tener total seguridad de que se va a acceder a memoria asignada a nuestro programa, de lo contrario ocurrirá un *segmentation fault* (viloación de segmento).
- Acceso indirecto (**`->`**): Si **`p`** es un puntero a una estructura **`p->member`** es un atajo a **`(*p).member`** (asumiendo que la estructura tiene un campo llamado **`member`**).

## Arreglos y punteros

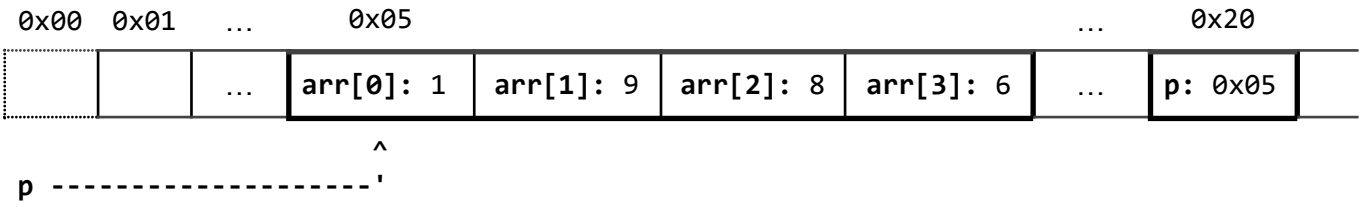
Cuando se declara una variable de tipo arreglo,

```
int arr[4];
```

hay dos formas de obtener la dirección de memoria al primer elemento:

- Usando el operador de referenciación: **`&arr[0]`**
- Usando el nombre del arreglo: **`arr`**

```
int arr[4]={1,9,8,6};
int *p=NULL;
p = &arr[0]; // Usando operador &
p = arr;     // Usando directamente el nombre de variable del arreglo
```




¿Qué diferencia hay entre **p** y **arr**?

Circunstancialmente se puede usar a **p** para acceder a los elementos del arreglo **arr** ya que **p[i]** y **arr[i]** van a devolver exactamente el mismo valor. Sin embargo, más adelante en el código se puede reutilizar a **p** para que apunte a otra variable, por ejemplo haciendo **p = &x;** (suponiendo que tenemos declarada a **int x;**). Por otro lado, aunque con la expresión **arr** obtenemos la dirección de memoria del primer elemento del arreglo, **arr** no es un puntero ya que no es posible hacer

```
int arr[4];
int x;
arr = &x;  // No se puede realizar esta asignación
```

## Memoria dinámica: Stack vs Heap

En el lenguaje del teórico práctico se usa el procedimiento **alloc()** para reservar memoria para un puntero, y **free()** para liberar dicha memoria:

```
var p: pointer to int

alloc(p)
*p := 5
free(p)
```

En C esto se hace usando las funciones **malloc()** y **free()**:

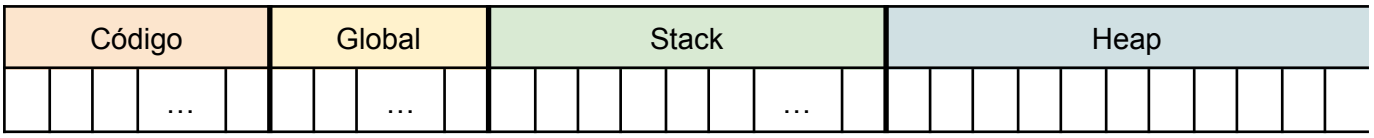
```
int *p=NULL;
p = malloc(sizeof(int));
*p = 5;
free(p);
```

La función **malloc()** toma un parámetro, que es un entero sin signo de tipo **size_t** (muy parecido a **unsigned long int**) que es la cantidad de memoria en *bytes* que se solicita reservar. A diferencia de **alloc()** del teórico, que automáticamente reserva la cantidad necesaria según el tipo de puntero, en C hay que indicar explícitamente la cantidad de *bytes* a reservar. El operador **sizeof()** devuelve la cantidad de *bytes* ocupados por una expresión o tipo, por lo que resulta indispensable para el uso de **malloc()** (aún si uno hubiera memorizado cuantos *bytes* ocupa cada tipo en su computadora, esto puede variar según la versión del sistema operativo o el microprocesador en el que se use el programa).

```
$ man malloc
```

Las direcciones de memoria que devuelve **malloc()** se encuentran en la sección de memoria

denominada *Heap* (no confundir con la estructura de datos que lleva el mismo nombre ya que no tiene ninguna relación).




A modo informativo, el mapa de más arriba es un esquema de cómo se organiza la memoria. La sección de *Código* contiene las instrucciones del programa, la sección *Global* contiene las variables globales, la sección *Stack* es donde están las variables que usamos en las funciones de nuestro programa (memoria estática) y la sección *Heap* es la región de la memoria dinámica la cual se reserva y libera manualmente mediante `malloc()` y `free()`. El *Stack* por su parte se maneja de manera automática, reservando memoria para las variables declaradas en una función que se comienza a ejecutar y liberando esa memoria cuando la función termina su ejecución.

Otra gran diferencia entre el *Stack* y el *Heap*, es que la cantidad de memoria asignada para el Stack es limitada. Si los datos contenidos en el *Stack* superan dicho límite se genera un ***stack overflow***. Si durante la ejecución de un programa se está ejecutando una función `f1` y ésta llama a su vez a otra función `f2`, durante la ejecución de `f2` las variables de `f1` siguen en el *Stack* ya que aún no se terminó de ejecutar. Las variables de las funciones llamadas de manera anidada se van apilando entonces. Hay en consecuencia un límite en la cantidad de llamadas anidadas de funciones, particularmente un número máximo de llamadas recursivas. La cantidad dependerá de cuánta memoria ocupen las variables de las funciones involucradas. Esto hace que si una función declara un arreglo en memoria estática muy grande, podría dejar poco margen para llamadas a otras funciones o directamente generar un *stack overflow* porque el arreglo no entra en el *Stack*.

Por su parte la memoria en el *Heap* tiene disponible toda la memoria *RAM* de la computadora, por lo que mientras haya memoria libre se podrá pedir reservar nueva memoria mediante `malloc()`. Pero *un gran poder conlleva una gran responsabilidad*, por lo que no se debe olvidar liberar la memoria reservada cuando deje de usarse, puesto que los ***memory leaks*** pueden generar a la larga que la computadora se bloquee por completo.

## Ejercicio 4: Punteros avanzados

**a)** En el programa implementado en `array.c` se inicializa en cero un arreglo `arr` (de forma muy rebuscada). Se debe reescribir la sección de código indicada para que mediante el puntero `p` se inicialice en cero el arreglo `arr` sin utilizar los operadores `&` y `*` en ningún momento.

**b)** Programar la función

```
void set_name(name_t new_name, data_t *d);
```

que debe cambiar el campo `name` de la estructura apuntada por `d` con el contenido de `new_name` y utilizarla para modificar la variable `messi` de tal manera que en su campo `name` contenga la cadena `"Lionel Messi".`

**c)** Completar el archivo `sizes.c` para que muestre el tamaño en *bytes* de cada miembro de la estructura `data_t` por separado y el tamaño total que ocupa la estructura en memoria. *¿La suma de los miembros coincide con el total? ¿El tamaño del campo* `name` *depende del nombre que contiene?*

De manera similar a lo hecho para mostrar los tamaños de cada campo de la estructura, agregar un mensaje que muestre la dirección de memoria de cada campo. Se recomienda usar los dos tipos de visualizaciones explicadas anteriormente (direcciones e índices). Analizar la salida y responder: *¿Hay algo raro en las direcciones de memoria?*

**d)** En el directorio **`static`** se encuentra el programa del Laboratorio 1 que carga en un arreglo en memoria estática desde un archivo. Completar en la carpeta **`dynamic`** la función **`array_from_file()`** de **`array_helpers.c`**:

```
int *array_from_file(const char *filepath, size_t *length);
```

que carga los datos del archivo **`filepath`** devolviendo un puntero a memoria dinámica con los elementos del arreglo y dejando en **`*length`** la cantidad de elementos leídos. Completar además en **`main.c`** el código necesario para liberar la memoria utilizada por el arreglo. Probar el programa con todos los archivos de la carpeta **`input`** para asegurar el correcto funcionamiento (notar que la versión en **`static`** no funciona para todos los archivos de la carpeta **`input`**).

# Preliminares: TADS

## Encapsulamiento

Lo primero que debemos observar es la forma en la que logramos mantener separadas la especificación del TAD de su implementación. Cuando definimos un TAD es deseable garantizar **encapsulamiento**, es decir, que solamente se pueda acceder y/o modificar su estado a través de las operaciones provistas. Esto no siempre es trivial ya que los tipos abstractos están implementados en base a los tipos concretos del lenguaje. Entonces es importante que además de separar la especificación e implementación se garantice que quién utilice el TAD no pueda acceder a la representación interna y operar con los tipos concretos de manera descontrolada. Si esto se logra será posible cambiar la implementación del TAD sin tener que modificar ningún otro módulo que lo utilice.

No todos los lenguajes brindan las mismas herramientas para lograr una implementación *opaca* y se debe usar el mecanismo apropiado según sea el caso. Particularmente el lenguaje del teórico-práctico separa la especificación de un TAD de su implementación utilizando las signaturas **`spec`** … **`where`** e **`implement`** … **`where`** respectivamente. En este laboratorio se debe buscar la manera de lograr encapsulamiento usando el lenguaje **C**.

## Métodos de TADs

En el diseño de los tipos abstractos de datos (tal como se vio en el teórico-práctico) aparecen los **constructores**, las **operaciones** y los **destructores,** que se declaran como funciones o procedimientos. Recordar (se vio en el ejercicio 2) que los procedimiento en C no existen como tales sino que se usan funciones con tipo de retorno `void`, es decir, funciones que no devuelven ningún valor al llamarlas. A veces se buscará evitar procedimientos con una variable de salida usando directamente una función para simplificar y evitar así usar punteros extra (en el ejercicio 2 se vio que es necesario usar punteros para simular variables de salida).

A diferencia del práctico, a las *precondiciones* y *postcondiciones* de los métodos **sí vamos a verificarlas** (en la medida de lo posible). Recordar que nuestros programas deben ser **robustos**, por lo tanto cuando corresponda usaremos **`assert()`** para garantizar el cumplimiento de las pre y post

condiciones de los métodos. Esta práctica es propia de la etapa de desarrollo de un programa, y una vez que el mismo está finalizado, verificado y listo para desplegarlo en producción, se pueden eliminar las aserciones mediante un flag de compilación.

## Ejercicio 5: TAD Par

Considerar la siguiente especificación del TAD Par

---

**spec** Pair **where**

**constructors**
  **fun** new(**in** x : int, **in** y : int) **ret** p : Pair
  *{- crea un par con componentes (x, y) -}*

**destroy**
  **proc** destroy(**in/out** p : Pair)
  *{- libera memoria en caso que sea necesario -}*

**operations**
  **fun** first(**in** p : Pair) **ret** x : int
  *{- devuelve el primer componente del par-}*

  **fun** second(**in** p : Pair) **ret** y : int
  *{- devuelve el segundo componente del par-}*

  **fun** swapped(**in** p : Pair) **ret** s : Pair
  *{- devuelve un nuevo par con los componentes de p intercambiados -}*

---

**a)** Abrir la carpeta **`pair_a-tuple`** y revisar la especificación del TAD en **`pair.h`**. Luego crear el archivo **`pair.c`** e implementar las funciones del TAD. Para probar la implementación usar el módulo **`main.c`** como programa de prueba.

**b)** Abrir la carpeta **`pair_b-array`** y revisar la nueva especificación del TAD en **`pair.h`**. Crear el archivo **`pair.c`** con la implementación de las funciones del TAD. Copiar el archivo **`main.c`** del apartado anterior al directorio **`pair_b-array`** y compilar.

I. ¿Por qué falla?
II. Hacer las modificaciones necesarias en **`main.c`** para que compile.
III. *¿El diseño del TAD logra encapsulamiento? ¿Por qué sí? ¿Por qué no?*

**c)** Abrir la carpeta **`pair_c-pointer`** y revisar la especificación del TAD en **`pair.h`**. Luego completar la implementación de las funciones y compilar usando el módulo **`main.c`** como programa de prueba. *¿La implementación logra encapsulamiento? ¿Por qué sí? ¿Por qué no?*

*Para definir constructores, destructores y operaciones de copia será necesario hacer manejo de **memoria dinámica** (pedir y liberar memoria en tiempo de ejecución). En este caso se necesita espacio suficiente para almacenar un valor de tipo* **`struct _pair_t`**.

**d)** Abrir la carpeta **`pair_d-pointer`** y revisar **`pair.h`**. Copiar el archivo **`pair.c`** del apartado anterior y agregar las definiciones necesarias para que compile con la nueva versión de **`pair.h`**. *¿La implementación logra encapsulamiento?* Copiar el archivo **`main.c`** del apartado anterior y compilar. *¿Está bien que falle?* Hacer las modificaciones necesarias en **`main.c`** para que compile sin errores.

**e)** Considerar la nueva especificación polimórfica para el TAD Pair:

```
spec Pair of A where

constructors
    fun new(in x : A, in y : A) ret p : Pair of A
    {- crea un par con componentes (x,  y) -}

destroy
   proc destroy(in/out p : Pair of A)
   {- libera memoria en caso que sea necesario -}

operations
    fun first(in p : Pair of A) ret x : A
    {- devuelve el primer componente del par-}

    fun second(in p : Pair of A) ret y : A
    {- devuelve el segundo componente del par-}

    fun swapped(in p : Pair of A) ret s : Pair of A
    {- devuelve un nuevo par con los componentes de p intercambiados -}
```

¿Qué diferencia hay entre la especificación anterior y la que se encuentra en el `pair.h` de la carpeta `pair_d-pointer`? Copiar `pair.c` del apartado anterior y modificarlo para utilizar la nueva interfaz especificada en `pair.h`. Pueden utilizar el `main.c` del apartado anterior para compilar.

## Ejercicio 6: TAD Contador

Dentro de la carpeta `ej6` se encuentran los siguientes archivos:

| *Archivo* | *Descripción* |
|---|---|
| `counter.h` | Contiene la especificación del TAD Contador. |
| `counter.c` | Contiene la implementación del TAD Contador. |
| `main.c` | Contiene al programa principal que lee uno a uno los caracteres de un archivo chequeando si los paréntesis están balanceados. |

**a)** Implementar el TAD Contador. Para ello deben abrir `counter.c` y programar cada uno de los constructores y operaciones cumpliendo la especificación dada en `counter.h`. Recordar que deben verificar en `counter.c` todas las precondiciones especificadas en `counter.h` usando llamadas a la función `assert()`.

**b)** Usar el TAD Contador para chequear paréntesis balanceados. Para ello deben abrir el archivo `main.c` y entender qué es lo que hace la función `matching_parentheses()` y completar con llamadas al constructor y destructor del contador donde consideren necesario. ¡Es muy importante llamar al destructor del TAD una vez este no sea necesario para poder liberar el espacio de memoria que tiene asignado!

Una vez implementados los incisos **(a)**, **(b)** compilar ejecutando:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c counter.c main.c
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 counter.o main.o -o counter
```

Ahora se puede ejecutar el programa corriendo:

```
$ ./counter input/<file>.in
```

siendo **<file>** alguno de los nombres de archivo dentro de la carpeta **input**. Asegurarse que para aquellos archivos con paréntesis balanceados, al ejecutar el programa se imprima en pantalla

```
Parentheses match.
```

y para aquellos con paréntesis no balanceados imprima

```
Parentheses mismatch.
```

## Ejercicio 7: TAD Lista

Dentro de la carpeta **ej7** se encuentran los siguientes archivos:

| *Archivo* | *Descripción* |
|---|---|
| **main.c** | Contiene al programa principal que lee los números de un archivo para ser cargados en nuestra lista y obtener el promedio. |
| **array_helpers.h** | Contiene descripciones de funciones auxiliares para manipular arreglos. |
| **array_helpers.c** | Contiene implementaciones de dichas funciones. |

**a)** Crear un archivo **list.h**, especificando allí todos los constructores y operaciones vistos sobre el TAD Lista en el teórico. Recomendamos definir el nombre del TAD como `list` ya que en el archivo **main.c** se encuentra mencionado de esa manera.

Existe un par de diferencias entre nuestro TAD Lista en C respecto al visto en el teórico. Para simplificar la implementación, nuestras listas serán solamente de tipo `int`, es decir, no hay *polimorfismo*. Si bien el tipo será fijo (**int**), una buena idea es definir un tipo en **list.h** usando **typedef**. Un ejemplo de esto sería definir

```
typedef int list_elem;
```

y utilizar **list_elem** en vez de **int** en todos los constructores/operaciones (al estilo de lo realizado en el ejercicio ***5e*** ).

Otra diferencia con el teórico es que aquellos procedimientos que modifiquen la lista deben escribirse como funciones que devuelvan la lista resultante. Como ya fue mencionado, esto es para evitar tener que simular parámetros de salida.

No olvidar de:

- Garantizar encapsulamiento en tu TAD.
- Especificar una función de destrucción y copia.
- Especificar las precondiciones.

**b)** Crear un archivo **`list.c`**, e implementar cada uno de los constructores y operaciones declaradas en el archivo **`list.h`** . La implementación debe ser como se presenta en el teórico, es decir, utilizando punteros (listas enlazadas).

**c)** Abrir el archivo **`main.c`** e implementar las funciones **`array_to_list()`** y **`average()`**. Para la implementación de **`average()`** se sugiere revisar la definición del teórico.

Una vez implementados los incisos **a)**, **b)** y **c)**, compilar ejecutando:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c list.c array_helpers.c main.c
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 list.o array_helpers.o main.o -o average
```

Ahora se puede ejecutar el programa corriendo:

```
$ ./average input/<file>.in
```

siendo **`<file>`** alguno de los nombres de archivo dentro de la carpeta **`input`**. Asegurar que el valor de los promedios que se imprimen en pantalla sean correctos y animense a definir sus propios casos de *input*.