

# Práctica de Organización del Computador II

Programación orientada a datos - Segunda parte

---

Primer Cuatrimestre 2022

Organización del Computador II  
DC - UBA

# Punteros

---

Habiendo presentado un modelo de memoria contigua y direccionable vamos intentar explicar de qué hablamos cuando hablamos de punteros:

- Estructuralmente son datos numéricos (del mismo tamaño que una dirección de memoria)
- La declaración de tipo (`val_type *ptr;`) no sólo indica que se trata de un puntero (modificador `*`) sino del tipo de dato al que apunta (tipo referido `val_type`).
- El valor almacenado se corresponde con una dirección de memoria donde comienza un dato de tipo `val_type`.

Habiendo presentado un modelo de memoria contigua y direccionable vamos intentar explicar de qué hablamos cuando hablamos de punteros:

- Si aplicamos el operador de referencia (`&val`) semánticamente estamos consiguiendo la dirección de memoria donde comienza el dato `val`.
- Si aplicamos el operador de desreferencia (`*ptr`) semánticamente estamos accediendo al dato apuntado por `ptr`.

Habiendo presentado un modelo de memoria contigua y direccionable vamos intentar explicar de qué hablamos cuando hablamos de punteros:

- Tengan cuidado porque el operador de desreferencia (`*ptr`) y el indicador de tipo puntero `val_type *ptr`; son el mismo caracter pero indican cosas bien distintas.
- Si declaran varios punteros en la misma línea pongan el modificador en los nombres de variable:

```
val_type *ptr, *ptr2; .
```

En este punto es conveniente pensar en qué significado tiene, en relación a la memoria un identificador de variable.

Por ejemplo, cuando realizamos una asignación:

```
uint16_t i = 3;  
struct s obj = {'a',4};  
float arr[3] = {123.43f, 2f, 3.5f};
```

A la hora de comprender la asignación tenemos que pensar que el identificador en cuestión `i`, `obj.c`, `arr[5]` es un sustituto conveniente que al compilar y ejecutar el código se traduce en la posición de memoria donde comienza el dato.

¿Cómo se resuelve cada referencia a partir de una referencia por identificador?

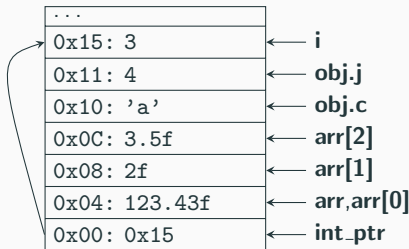
...	
0x11: 3	← <b>i</b>
0x0D: 4	← <b>obj.j</b>
0x0C: 'a'	← <b>obj.c</b>
0x08: 3.5f	← <b>arr[2]</b>
0x04: 2f	← <b>arr[1]</b>
0x00: 123.43f	← <b>arr,arr[0]</b>

Agreguemos un puntero a entero al caso anterior.

```
uint16_t i = 3;  
struct s obj = {'a', 4};  
float arr[3] = {123.43f, 2f, 3.5f};  
uint16_t *int_ptr = &i;
```

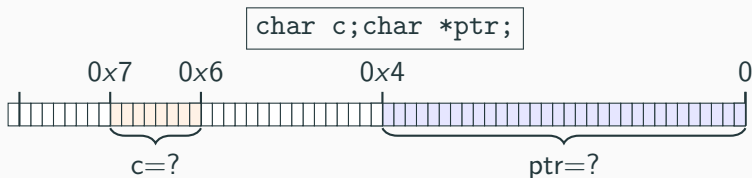


Dibujemos las referencias.

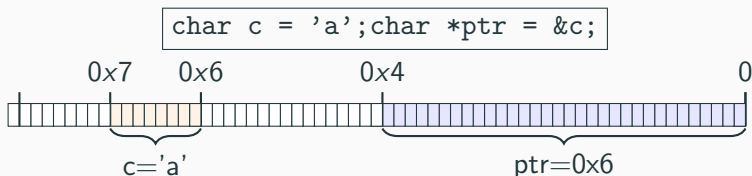


Los identificadores se resuelven en una dirección de memoria cuyo valor es del tipo declarado para el dato, y un puntero se resuelve en una dirección de memoria cuyo tipo es a su vez una dirección de memoria.

Lo importante es que al compilar y ejecutar el código, las variables se traducen en posiciones de memoria donde comienza un dato, al igual que pasa con las etiquetas en ASM.



- ¿Qué valores tienen `ptr` y `c`?
- No sabemos, probablemente lo que hubiera antes en `0x0` y `0x6` respectivamente.
- **Nota:** La constante `NULL` o `0` se utiliza para identificar punteros no inicializados (que aún no apuntan a ningún sitio).



- ¿Qué valores tienen `ptr` y `c`?
- Ahora en 0x6(`c`) vamos a tener el dato numérico que representa al caracter `'a'` y en `ptr` la posición de memoria (0x6) donde comienza el dato identificado por la variable `c`.

Ahora volvamos a las operaciones propias de punteros.

```
char *left, *right, tmp;
```

**La desreferencia de una lectura** `tmp = *left` recupera un dato la dirección almacenada en el puntero `left` y lo almacena en destino `tmp`.

**La desreferencia de una escritura** `*right = tmp` pisa el dato ubicado en la dirección almacenada en el puntero `right` con el valor de origen `tmp`.

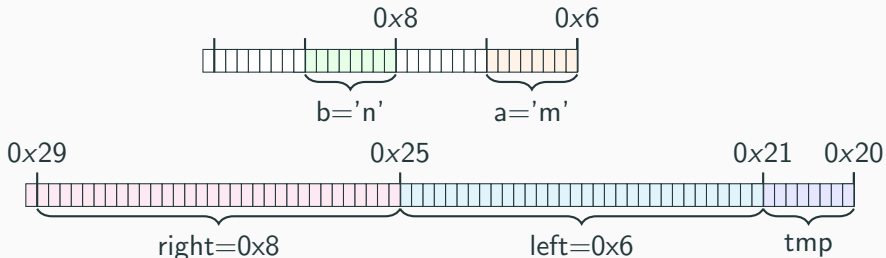
Implementando swap:

```
char a = 'm'; char b = 'n';  
charSwap(&a, &b);  
void charSwap(char* left, char* right){  
    char tmp = *left;  
    *left = *right;  
    *right = tmp;  
}
```

¿Qué sucede con la ejecución de este programa?

¿Por qué la función no es charSwap(char, char)?

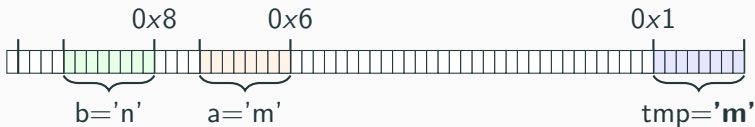
# Referencia-Desreferencia, un ejemplo



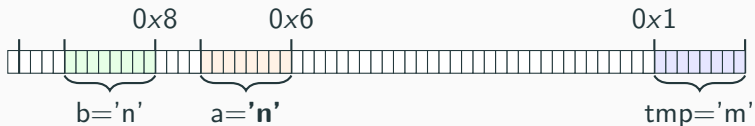
```
charSwap(&a,&b);
```

```
left = &a; right = &b;
```

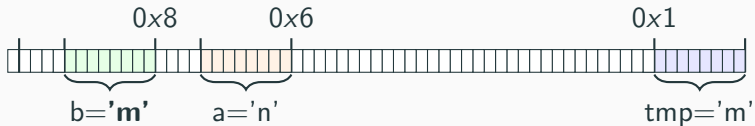
# Referencia-Desreferencia, un ejemplo



```
char tmp = *left;
```



```
*left = *right;
```



```
*right = tmp;
```



Mutando swap:

```
char a = 'm'; char b = 'n';  
charSwap(&a, &b);  
void charSwap(char* left, char* right){  
    char tmp = *left;  
    *left = *right;  
    right = tmp;  
}
```

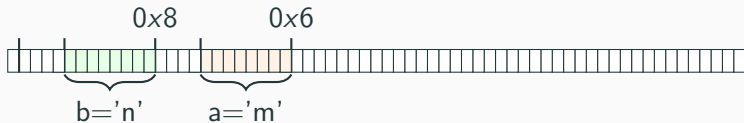
¿Qué hubiese pasado si al final hubiésemos hecho lo siguiente?

```
right = tmp;
```

¿Qué valor habría en a al regresar de la función?

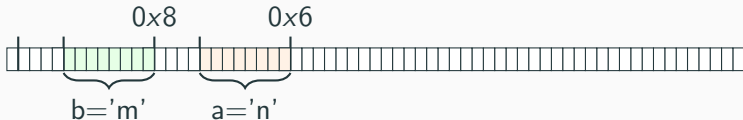
**Respuesta:** a seguiría valiendo 'n' porque se habría pisado el valor right, que es una variable local a la función.

```
Lo explicaremos en breve.
```



```
char a = 'm';  
char b = 'n';  
charSwap(&a, &b);  
printf("a: %c, b: %c", a, b);
```

¿Qué valores tienen las variables a y b al regresar de la función?

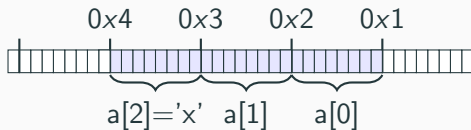


¿Qué pasó con tmp?

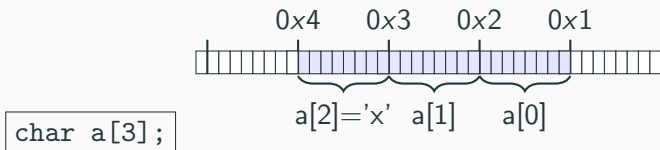
Lo explicaremos en breve.

Volvamos a estudiar los arreglos.

- Recordemos que estructuralmente son una **concatenación en memoria de varias instancias de un mismo tipo**.
- El arreglo **se identifica por la posición de memoria donde comienza**.
- Permite acceder a un elemento en particular a partir del operador de índice (`char a[3]; a[2] = 'x';`).

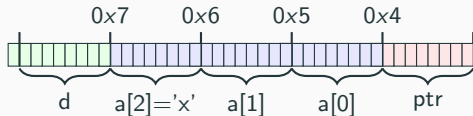


¿Podremos reproducir el funcionamiento del operador de índice haciendo uso de punteros?

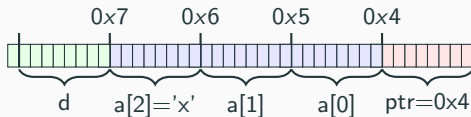


Si queremos acceder al  $i$ -ésimo elemento, deberíamos:

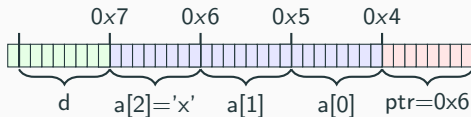
- Definir un puntero del tipo correcto `char *ptr;`.
- Apuntarlo al comienzo del arreglo `ptr = &(a[0]);`.
- Desplazar el puntero hasta la posición de comienzo del  $i$ -ésimo elemento `ptr += i;`.
- Conseguir el valor del dato apuntado por el puntero `d = *ptr;`.



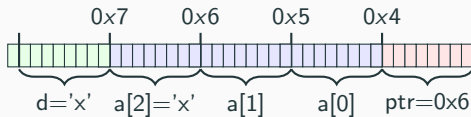
```
char *ptr; char d;
```



```
ptr = &(a[0]);
```



```
ptr += 2;
```



```
d = *ptr;
```

El hecho es que los arreglos se implementan con punteros y la semántica del operador de índice hace exactamente esto que acabamos de explicar:

```
a[i]  $\equiv$  *(a + i);
```

De hecho se puede aplicar el operador de índice sobre un puntero y lo va a interpretar como si fuera un arreglo.

```
char myStr[4] = "hola";
```

```
char *ptr = &(myStr[0]); ptr[3] = 'i';
```

Noten que no hizo falta multiplicar el índice por el tamaño de tipo.

```
a[i]  $\neq$  *(a + i * sizeof(val_type));
```

¿Pero por qué?

**Respuesta:** El compilador le da distintas interpretaciones al operador de suma (+) de acuerdo al tipo de dato a sumar, en el caso del puntero, como contiene la definición del tipo al que apunta define la suma o multiplicación sobre la base del tamaño del tipo.

Los llamados **strings** de ANSI-C se implementan como arreglos de caracteres. Suelen tener un tamaño fijo y la forma de determinar donde dejar de leer cuando queremos, por ejemplo, imprimirlos, es seguir hasta encontrar el caracter nulo o de valor 0.

```
char first_name[15] = "BYTE" ;  
char first_name[15] = { 'B', 'Y', 'T', 'E', '\0' } ;
```

Estas dos inicializaciones tienen el mismo efecto, el compilador de C identifica que estamos intentando inicializar un string en el primer caso y lo convierte en un arreglo de caracteres que terminan con el caracter nulo.

La segunda línea muestra un ejemplo de inicialización estática de arreglos.



Hasta ahora vimos:

- Una introducción a la **perspectiva de datos**.
- Una interpretación de la memoria como un **espacio contiguo y ordenado de bits (bytes)**.
- Un repaso de los tipos de datos básicos y **cómo se representan en memoria**.
- Una **introducción a los punteros** y las operaciones de referencia y desreferencia.

En breve seguimos con:

- Uso del **stack** y del **heap**.
- Manejo de memoria dinámica.
- Estructura completa del espacio de memoria de una aplicación.

**Intervalo y consultas (10 minutos)**

---