

Experiencia N°03

Métodos de Programación

1-2021



CONTENIDO

Variable local y global

Array

Array dinámico

Ejercicio



Variable local y global

Variable local y global

Array

Array dinámicos

Ejercicio

Variable local

- Es una variable definida dentro de una función.
- Pueden ser utilizadas sólo dentro de la función en que fue declarada.

```
void funcion() {  
  
    int a = 35;  
  
    printf("%d\n",a);  
    return;  
}
```



Variable local y global

Variable local y global

Array

Array dinámicos

Ejercicio

Variable global

- Es una variable definida fuera de una función.
- Se encuentran disponibles para todo el código a partir de la línea en que fue declarada.

```
int a = 34;  
  
void funcion() {  
    printf("%d\n", a);  
}
```



Variable local y global

Array

Array dinámicos

Ejercicio

Variable local y global

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

```
int num = 20;
```

Variable Global

```
void ejemplo(){  
    int otro = 43;  
    num += otro;  
    return;  
}
```

Variable Local

```
int main(){
```

```
    ejemplo();
```

```
    printf("%d\n",num);
```

Imprime 63

```
    printf("%d\n",otro);
```

ERROR

```
    return 0;
```

```
}
```



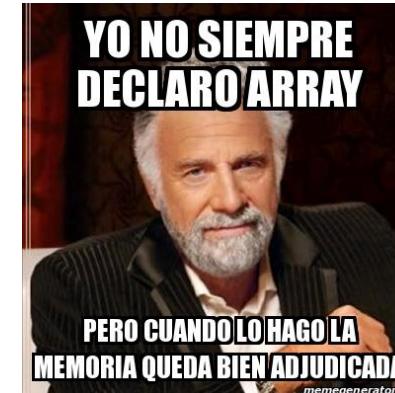
Variable local y global

Array

ARRAY

Array dinámicos

Ejercicio





Variable local y global

Array

Array dinámicos

Ejercicio

Array

- Los array o arreglos son colecciones en las que se puede referenciar varios valores bajo un mismo nombre y donde cada valor se almacena en forma **consecutiva** en la memoria.
- Similar al tipo de dato list que fue visto en Python, pero con claras diferencias al momento de utilizarlos en C.
- Debido a que C es fuertemente tipado, debemos especificar el tipo de dato que contendrá el array.
- Sí, cuando decimos EL tipo de dato, es porque el array solo puede contener valores de un solo tipo. Declarando así un array solo para enteros o uno solo para caracteres por ejemplo.



Variable local y global

Array

Array dinámicos

Ejercicio

Array

Para declarar un array en C se debe especificar:

- Tipo de dato que almacenará el array.
- Nombre del array.
- Tamaño del array entre corchetes. `int arreglo[5];`

¿Se dieron cuenta que dijimos tamaño?

- Es porque los array tienen un tamaño estático y no se le pueden agregar más elementos que el tamaño para el cual fue definido.





Variable local y global

Array

Array dinámicos

Ejercicio

Array

- El acceso es por indexación al igual que las list de Python.
- Para acceder a un valor en específico del array es necesario indicar el nombre del array, seguido de la posición del dato entre corchetes.
- Las posiciones se comienzan a contar desde cero y terminan en $n - 1$.

```
int arreglo[5];  
  
arreglo[0] = 2;  
arreglo[1] = 4;  
arreglo[2] = 6;  
arreglo[3] = 8;  
arreglo[4] = 10;
```

SE VIENE LO
TRISTE

- ```
printf("%d %d %d %d \n",arreglo[0],arreglo[1],arreglo[2],arreglo[3],arreglo[4]);
```
- **No** existen métodos como **append** para agregar más elementos.
  - **No** podemos **obtener el largo** de la cadena (los sabemos gracias a que nosotros lo definimos).
  - **No** existe **acceso a posiciones negativas** para recorrer al revés.





# Array

Variable local y global

↓  
Array

↓  
Array dinámicos

↓  
Ejercicio

- Tal vez piensen que los array se almacenan en memoria de la misma forma que cualquier variable declarada
- Pero ahora que sabemos que son los array, hay algo que debemos contarles.
- Los array “no existen” en verdad.



| Dirección | Contenido |
|-----------|-----------|
| 0061FF20  | a = 5     |
| 0061FF21  |           |
| 0061FF22  |           |
| 0061FF23  |           |

- En realidad ninguna colección existe como tal en memoria.
- Sino que una colección (el nombre del arreglo) solo indica en donde empieza la serie de valores en memoria.



# Array

Variable local y global

↓  
Array

↓  
Array dinámicos

Ejercicio

Veamos el siguiente ejemplo de tiempos donde el lenguaje era hermoso.

Recuerdan una clase en que el profesor de FCyP les dijo:

Si quieren  
copiar una lista  
no hagan esto.

Sino que tiene  
que hacerlo así.

$L = [1, 2, 3]$   
 $L2 = L$



$L = [1, 2, 3]$   
 $L2 = L[:]$





Variable local y global

Array

Array dinámicos

Ejercicio

# Array

Explicaremos ahora la razón

>>> L = [1, 2, 3]

Como dijimos, los array, listas y colecciones en general solo indican en donde comienzan los valores en la memoria.

O sea, que si entendemos bien L como variable no es otra cosa que una dirección de memoria, que apunta a donde comienzan los datos de la lista

| Dirección | Contenido    |
|-----------|--------------|
| 0061FF20  | L = 0061FF25 |
| 0061FF21  |              |
| 0061FF22  |              |
| 0061FF23  |              |
| 0061FF24  |              |
| 0061FF25  | 1            |
| 0061FF26  | 2            |
| 0061FF27  | 3            |
| 0061FF28  |              |
| 0061FF29  |              |
| 0061FF30  |              |
| 0061FF31  |              |



## Variable local y global

Array

Array dinámicos

Ejercicio

```
>>> L=[1,2,3]
>>> L2 = L
>>> L
[1, 2, 3]
>>> L2
[1, 2, 3]
```

Por lo que copiar con solo la asignación no es copiar en realidad, solo hemos creado dos formas de acceder a los mismos datos.

# Array

| Dirección | Contenido     |
|-----------|---------------|
| 0061FF20  | L = 0061FF25  |
| 0061FF21  |               |
| 0061FF22  | L2 = 0061FF25 |
| 0061FF23  |               |
| 0061FF24  |               |
| 0061FF25  | 1             |
| 0061FF26  | 2             |
| 0061FF27  | 3             |
| 0061FF28  |               |
| 0061FF29  |               |
| 0061FF30  |               |
| 0061FF31  |               |



## Variable local y global

Array

Array dinámicos

Ejercicio

```
>>> L=[1,2,3]
>>> L2 = L
>>> L
[1, 2, 3]
>>> L2
[1, 2, 3]
>>> L.append(4)
```

Y esta es la razón que al realizar una operación de agregación se obtenía un resultado extraño al revisar las listas.

Pero que ahora debería ser bastante previsible.

# Array

| Dirección | Contenido     |
|-----------|---------------|
| 0061FF20  | L = 0061FF25  |
| 0061FF21  |               |
| 0061FF22  | L2 = 0061FF25 |
| 0061FF23  |               |
| 0061FF24  |               |
| 0061FF25  | 1             |
| 0061FF26  | 2             |
| 0061FF27  | 3             |
| 0061FF28  | 4             |
| 0061FF29  |               |
| 0061FF30  |               |
| 0061FF31  |               |



## Variable local y global

Array

Array dinámicos

Ejercicio

```
>>> L=[1,2,3]
>>> L2 = L
>>> L
[1, 2, 3]
>>> L2
[1, 2, 3]
>>> L.append(4)
>>> L
[1, 2, 3, 4]
>>> L2
[1, 2, 3, 4]
```

# Array

| Dirección | Contenido     |
|-----------|---------------|
| 0061FF20  | L = 0061FF25  |
| 0061FF21  |               |
| 0061FF22  | L2 = 0061FF25 |
| 0061FF23  |               |
| 0061FF24  |               |
| 0061FF25  | 1             |
| 0061FF26  | 2             |
| 0061FF27  | 3             |
| 0061FF28  | 4             |
| 0061FF29  |               |
| 0061FF30  |               |
| 0061FF31  |               |



Variable local y global

Array

Array dinámicos

Ejercicio

>>> L=[1, 2, 3]

Veamos un ejemplo más complejo, pero si entendemos lo anterior no debiese ser tan confuso.

Supongamos la siguiente lista a la que le realizaremos varias operaciones y veamos que se muestra al acceder a ella luego.

# Array

| Dirección | Contenido    |
|-----------|--------------|
| 0061FF20  | L = 0061FF21 |
| 0061FF21  | 1            |
| 0061FF22  | 2            |
| 0061FF23  | 3            |
| 0061FF24  |              |
| 0061FF25  |              |
| 0061FF26  |              |
| 0061FF27  |              |
| 0061FF28  |              |
| 0061FF29  |              |
| 0061FF30  |              |
| 0061FF31  |              |



# Array

Variable local y global

```
>>> L=[1,2,3]
>>> L.append(L)
```

Array

Agreguemos *L* a *L*

Al mostrar *L* nuevamente  
¿qué veremos?

Array dinámicos

Ejercicio





Variable local y global

Array

Array dinámicos

Ejercicio

# Array

```
>>> L=[1,2,3]
>>> L.append(L)
>>> L
[1, 2, 3, [...]]
```

Debido a que `L` no es la lista en sí, sino una dirección de memoria, hemos puesto un cuarto elemento el cual nos indica cómo volver al inicio de la lista.

Comprobemos esto.  
Vamos al [...]

| Dirección | Contenido    |
|-----------|--------------|
| 0061FF20  | L = 0061FF21 |
| 0061FF21  | 1            |
| 0061FF22  | 2            |
| 0061FF23  | 3            |
| 0061FF24  | 0061FF21     |
| 0061FF25  |              |
| 0061FF26  |              |
| 0061FF27  |              |
| 0061FF28  |              |
| 0061FF29  |              |
| 0061FF30  |              |
| 0061FF31  |              |



Variable local y global

Array

Array dinámicos

Ejercicio

# Array

```
>>> L=[1,2,3]
>>> L.append(L)
>>> L
[1, 2, 3, [1, 2, 3, [...]]]
>>> L[3]
[1, 2, 3, [...]]
```

Correcto, como era de esperarse, nos vuelve a aparecer la misma lista.

¿Será así siempre?  
Como pensaría cualquier programador que se respete,  
solo queda una cosa por hacer...

| Dirección | Contenido    |
|-----------|--------------|
| 0061FF20  | L = 0061FF21 |
| 0061FF21  | 1            |
| 0061FF22  | 2            |
| 0061FF23  | 3            |
| 0061FF24  | 0061FF21     |
| 0061FF25  |              |
| 0061FF26  |              |
| 0061FF27  |              |
| 0061FF28  |              |
| 0061FF29  |              |
| 0061FF30  |              |
| 0061FF31  |              |



Variable local y global

Array

Array dinámicos

Ejercicio

# Array

```
>>> L=[1,2,3]
>>> L.append(L)
>>> L
[1, 2, 3, [...]]
>>> L[3]
[1, 2, 3, [...]]
>>> L[3][3][3][3][3]
[1, 2, 3, [...]]
```

OK, el mensaje fue fuerte  
y claro.



¿Pero podemos acceder  
al resto de elementos

| Dirección | Contenido    |
|-----------|--------------|
| 0061FF20  | L = 0061FF21 |
| 0061FF21  | 1            |
| 0061FF22  | 2            |
| 0061FF23  | 3            |
| 0061FF24  | 0061FF21     |
| 0061FF25  |              |
| 0061FF26  |              |
| 0061FF27  |              |
| 0061FF28  |              |
| 0061FF29  |              |
| 0061FF30  |              |
| 0061FF31  |              |



## Variable local y global

Array

Array dinámicos

Ejercicio

# Array

```
>>> L=[1,2,3]
>>> L.append(L)
>>> L
[1, 2, 3, [...]]
>>> L[3]
[1, 2, 3, [...]]
>>> L[3][3][3][3][3][3]
[1, 2, 3, [...]]
>>> L[3][3][3][1]
2
```

Tal como vemos, se puede acceder sin problemas.

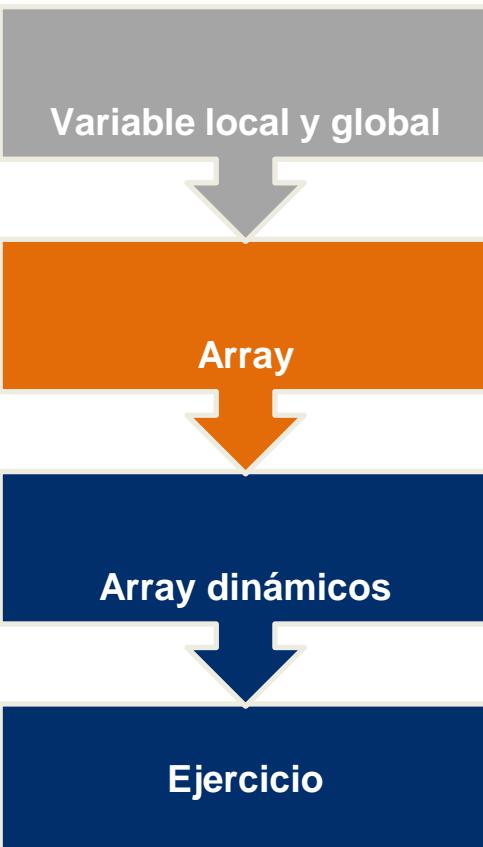
¿Qué pasa si ahora realizamos una concatenación?

```
>>> L = L + [5]
```

| Dirección | Contenido    |
|-----------|--------------|
| 0061FF20  | L = 0061FF21 |
| 0061FF21  | 1            |
| 0061FF22  | 2            |
| 0061FF23  | 3            |
| 0061FF24  | 0061FF21     |
| 0061FF25  |              |
| 0061FF26  |              |
| 0061FF27  |              |
| 0061FF28  |              |
| 0061FF29  |              |
| 0061FF30  |              |
| 0061FF31  |              |



# Array



La concatenación es una operación que ocupa una nueva dirección de memoria para guardar el resultado.

| Dirección | Contenido    |
|-----------|--------------|
| 0061FF20  | L = 0061FF26 |
| 0061FF21  | 1            |
| 0061FF22  | 2            |
| 0061FF23  | 3            |
| 0061FF24  | 0061FF21     |
| 0061FF25  |              |
| 0061FF26  | 1            |
| 0061FF27  | 2            |
| 0061FF28  | 3            |
| 0061FF29  | 0061FF21     |
| 0061FF30  | 5            |
| 0061FF31  |              |



# Array

Variable local y global

Array

Array dinámicos

Ejercicio

```
>>> L=[1,2,3]
>>> L.append(L)
>>> L
[1, 2, 3, [...]]
>>> L[3]
[1, 2, 3, [...]]
>>> L[3][3][3][3][3]
[1, 2, 3, [...]]
>>> L[3][3][3][1]
2
>>> L = L + [5]
>>> L
[1, 2, 3, [1, 2, 3, [...]], 5]
```

Hagamos una última maldad, agreguemos **L** a **L** nuevamente.

```
>>> L.append(L)
```



# Array

Variable local y global

Array

Array dinámicos

Ejercicio

```
>>> L=[1,2,3]
>>> L.append(L)
>>> L
[1, 2, 3, [...]]
>>> L[3]
[1, 2, 3, [...]]
>>> L[3][3][3][3][3]
[1, 2, 3, [...]]
>>> L[3][3][3][1]
2
>>> L = L + [5]
>>> L
[1, 2, 3, [1, 2, 3, [...]], 5]
>>> L.append(L)
```

¿Qué veremos al mostrar *L*?



# Array

Variable local y global

Array

Array dinámicos

Ejercicio

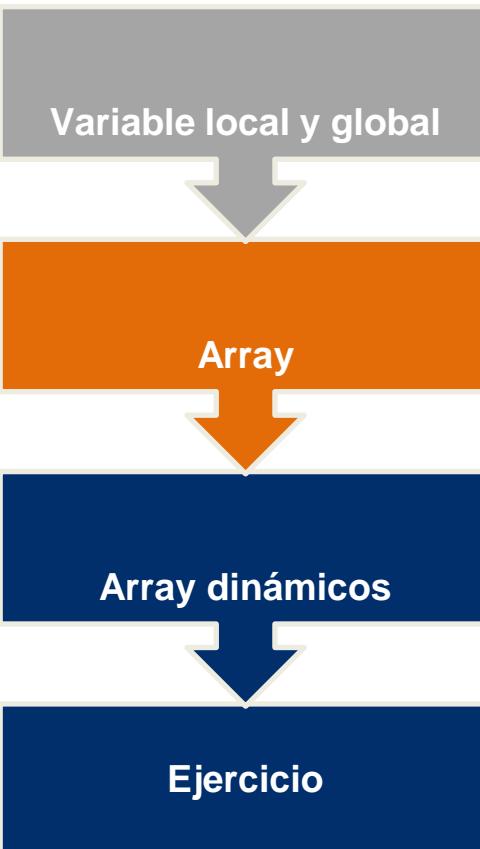
```
>>> L=[1,2,3]
>>> L.append(L)
>>> L
[1, 2, 3, [....]]
>>> L[3]
[1, 2, 3, [....]]
>>> L[3][3][3][3][3]
[1, 2, 3, [....]]
>>> L[3][3][3][1]
2
>>> L = L + [5]
>>> L
[1, 2, 3, [1, 2, 3, [....]], 5]
>>> L.append(L)
>>> L
[1, 2, 3, [1, 2, 3, [....]], 5, [....]]
```

Finalmente, vamos a los [...] para ver que hay.





# Array



```
>>> L=[1,2,3]
>>> L.append(L)
>>> L
[1, 2, 3, [...]]
>>> L[3]
[1, 2, 3, [...]]
>>> L[3][3][3][3][3]
[1, 2, 3, [...]]
>>> L[3][3][3][1]
2
>>> L = L + [5]
>>> L
[1, 2, 3, [1, 2, 3, [...]], 5]
>>> L.append(L)
>>> L
[1, 2, 3, [1, 2, 3, [...]], 5, [...]]
>>> L[5]
[1, 2, 3, [1, 2, 3, [...]], 5, [...]]
```

## Era de esperarse ¿no?



# Array

Variable local y global

Array

Array dinámicos

Ejercicio

```
>>> L=[1,2,3]
>>> L.append(L)
>>> L
[1, 2, 3, [...]]
>>> L[3]
[1, 2, 3, [...]]
>>> L[3][3][3][3][3]
[1, 2, 3, [...]]
>>> L[3][3][3][1]
2
>>> L = L + [5]
>>> L
[1, 2, 3, [1, 2, 3, [...]], 5]
>>> L.append(L)
>>> L
[1, 2, 3, [1, 2, 3, [...]], 5, [...]]
>>> L[5]
[1, 2, 3, [1, 2, 3, [...]], 5, [...]]
>>> L[3][3]
[1, 2, 3, [...]]
```

¿Pero por qué ahora mostró algo distinto?



# Array

Variable local y global

↓  
Array

↓  
Array dinámicos

Ejercicio

Básicamente, porque al momento de la concatenación teníamos la vieja dirección de memoria de L, por lo que en la posición [3][3] esta esa vieja lista, pero al hacer el append final agregamos al nueva dirección de memoria que se generó en la concatenación.

```
>>> L
[1, 2, 3, [1, 2, 3, [...]], 5]
>>> L.append(L)
>>> L
[1, 2, 3, [1, 2, 3, [...]], 5, [...]]
>>> L[5]
[1, 2, 3, [1, 2, 3, [...]], 5, [...]]
>>> L[3][3]
[1, 2, 3, [...]]
```

| Dirección | Contenido    |
|-----------|--------------|
| 0061FF20  | L = 0061FF26 |
| 0061FF21  | 1            |
| 0061FF22  | 2            |
| 0061FF23  | 3            |
| 0061FF24  | 0061FF21     |
| 0061FF25  |              |
| 0061FF26  | 1            |
| 0061FF27  | 2            |
| 0061FF28  | 3            |
| 0061FF29  | 0061FF21     |
| 0061FF30  | 5            |
| 0061FF31  | 0061FF26     |



# Array

Variable local y global

↓  
Array

↓  
Array dinámicos

Ejercicio

- ¿Y que hay que rescatar de todo esto?

- Algo simple pero muy potente para visualizar nuestros datos a futuro.
- Tal vez antes, en un inicio, veíamos a las variables como datos escalares. Algo como esto



- Una lista como algo así



- Una lista de listas como algo así





# Array

Variable local y global

↓  
Array

↓  
Array dinámicos

Ejercicio

- Pero ya sabemos que en memoria esa forma de matriz no existe.
- Sino que esa matriz tendrá más bien esta forma.
- Sí, el término antes usado “listas de listas” es correcto.
- Ya sabemos que la lista en verdad es una dirección de memoria.
- Por lo que nuestra matriz es en verdad una dirección de memoria que nos indica donde empieza la colección de direcciones de memoria que indican donde empiezan las colecciones datos.

| Dirección | Contenido    |
|-----------|--------------|
| 0061FF20  | M = 0061FF22 |
| 0061FF21  |              |
| 0061FF22  | 0061FF25     |
| 0061FF23  | 0061FF28     |
| 0061FF24  |              |
| 0061FF25  | 1            |
| 0061FF26  | 2            |
| 0061FF27  |              |
| 0061FF28  | 3            |
| 0061FF29  | 4            |
| 0061FF30  |              |
| 0061FF31  |              |



Variable local y global

Array

Array dinámicos

Ejercicio

# Array

- Sí, sabemos que esa representación es media confusa. Así que veámoslo con una versión menos violenta.

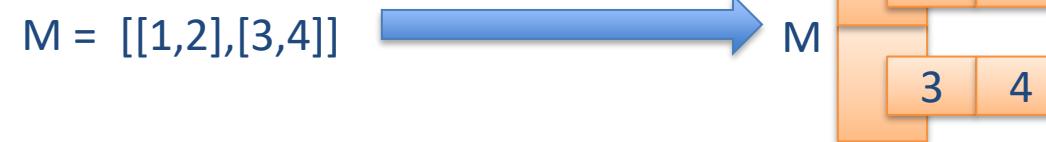
- Si visualizamos una variable así



- Y una lista como algo así



- Entonces una listas de listas debiera representarse así





Variable local y global

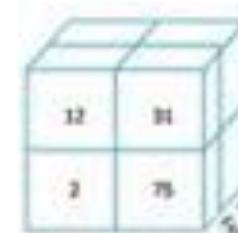
Array

Array dinámicos

Ejercicio

# Array

- Este pequeño cambio permite que pasemos de una representación limitada por la imaginación y dimensiones físicas. A una que permite visualizar n dimensiones.
- Si bien no es complicado visualizar una lista de listas de listas como un cubo de información.



Al llegar a estructuras de más de 3 dimensiones es complejo de visualizar



# Array

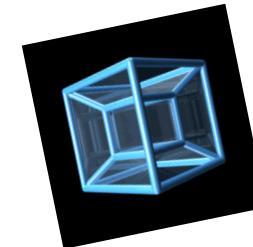
Variable local y global

↓  
Array

↓  
Array dinámicos

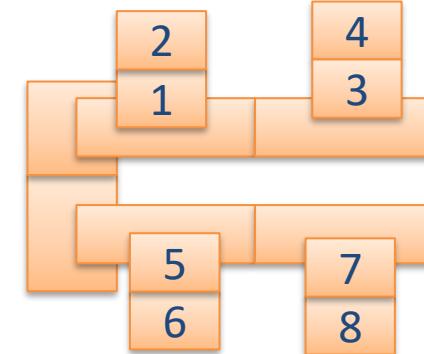
Ejercicio

- Para una lista de listas de listas de listas, tendríamos que visualizar un teseracto de información



- Pero ahora podemos ver una lista de listas de listas como

$C = [[[1,2],[3,4]],[[5,6],[7,8]]]$  → C





Variable local y global

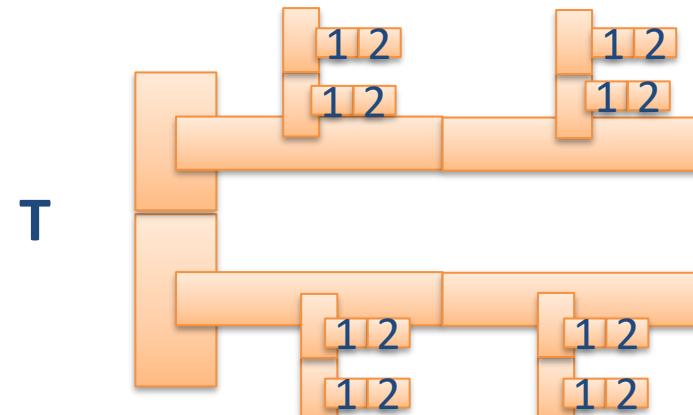
Array

Array dinámicos

Ejercicio

# Array

- Y agregar una nueva dimensión entonces no es un problema.
- Entendiendo lo que es una colección, sabemos que sus dimensiones no son dimensiones físicas, sino que una abstracción simplemente.
- Es por esto que podemos definir colecciones de n dimensiones sin mayor dificultad.





# Array dinámicos

Variable local y global



Array



Array dinámicos



Ejercicio

El hecho que sepamos que los array no son otra cosa que un puntero, nos da la posibilidad de crear este array desde cero.

Recordemos que podemos declarar punteros agregando un \* después del tipo de dato al definir una variable.

```
int * arreglo;
```

Esto significa que hay una forma de poder trabajar arreglos dinámicos, es decir, poder modificar su tamaño.

Para esto deberemos calcular la memoria necesaria para nuestro nuevo array, reservar y adjudicar.

Para realizar estos procesos debemos utilizar la operación **sizeof** y alguna función como **malloc**.



Variable local y global

↓  
Array

↓  
Array dinámicos

Ejercicio

# Array dinámicos

**malloc** es una función de la librería **stdlib.h**, la cual permite reservar y adjudicar memoria.

Para esto necesita la cantidad de memoria a adjudicar como parámetro de entrada.

Como resultado, malloc entrega un puntero void el cual apunta a la memoria reservada o se entrega un puntero nulo en caso de no llevarse a cabo la acción.

Cabe destacar que es necesario castear (casting o conversión de tipo) el resultado de la función malloc de void al tipo de dato deseado. Esto se hace anteponiendo entre paréntesis el tipo de dato al que se desea forzar a convertir algo.

`int a = (int) x;`

Se fuerza al valor de **x** a convertirse en un entero y se guarda en **a**.



# Array dinámicos

Variabile local y global

**sizeof** no es una función, sino que un operador. Por lo que es como signo de **+** en este aspecto, pero se utiliza muy similar a una función.

A este se le ingresa un tipo de dato como parámetro de entrada.

sizeof entrega un entero, él representa el espacio en memoria ocupado por ese tipo de dato en bytes.

Array dinámicos

sizeof(int); → 4

Ejercicio



# Array dinámicos

Variable local y global

Reservemos y adjudiquemos memoria.

Creemos un array para 5 enteros:

1. Debemos calcular cuánta memoria requiere un entero.
2. Lo multiplicamos por la cantidad de enteros que queremos.
3. Reservamos y adjudicamos esa cantidad de memoria.
4. Convertimos el puntero a la memoria adjudicada al tipo de dato puntero a entero (tipo de dato que debe ser nuestro array)
5. Asignamos esto a nuestro array previamente declarado.

```
int * a = (int *) malloc (sizeof(int) * 5);
```

¡Hecho! Ahora *a* es un array de 5 enteros listo para ser utilizado.

Ejercicio



Variable local y global

↓  
Array

↓  
Array dinámicos

Ejercicio

# Array dinámicos

Para reservar memoria para un array multidimensional es el mismo proceso, solo que se debe realizar para cada array interno también.

Por ejemplo, la declaración de una “matriz de 2x2”

```
int ** a = (int **) malloc (sizeof(int *) * 2);

a[0] = (int *) malloc (sizeof(int) * 2);
a[1] = (int *) malloc (sizeof(int) * 2);

a[1][0] = 69;
printf("%d\n",a[1][0]);
```

Note el tipo de dato de a, como también el tipo de dato al que se castean las posiciones 0 y 1 de a y finalmente al tipo de dato al que se le calcula el tamaño en la primera línea.



# Array dinámicos

Variable local y global

Si queremos cambiar el tamaño de **a**, podemos declarar un arreglo **b** con el tamaño que queramos (por ejemplo, *un espacio más que a*) y luego copiar el arreglo **a** en **b**.

¡Importante! Una vez se deja de utilizar un arreglo al cual le hemos asignado memoria, debemos liberarla.

Para esto debemos utilizar la función **free()**, la cual recibe como parámetro de entrada el puntero que apunta a la memoria adjudicada, en nuestro caso será el nombre del array.

Array dinámicos

Si se reservó memoria para una array multidimensional, se debe liberar la memoria para cada array al que se adjudicó memoria y siempre desde las dimensiones más internas a las externas. Por lo que si se adjudico memoria para una matriz **M**, nunca se debe empezar haciendo **free(M)**.

Adivine el porqué.

Ejercicio



# Cadenas de caracteres

Variable local y global



Array



Array dinámicos



Ejercicio

Como ya sabemos, en C no existen los Strings, para trabajar texto se utilizan cadenas de caracteres, lo que quiere decir que se trabajan como array de char.

Se utilizan igual que cualquier array de otro tipo de dato. Afortunadamente, al momento de utilizar los caracteres de formato, existe uno particular para las cadenas de caracteres, este es el **%s**, el que permite indicar que se desea trabajar una cadena de caracteres directamente.

Por lo que utilizarlo en un printf nos permitirá imprimir todo el array de char con solo indicar el nombre del array.

En el caso de scanf nos permite capturar texto desde pantalla y asignarlo al array.

Tener cuidado al pasar la variable al scanf. **¡Recuerde que un array ya es una dirección de memoria!**



# Ejercicios

Desarrolle en su casa los siguientes ejercicios a modo de estudio.

Variabile local y global



Cree un programa que:

- Solicite los elementos de un arreglo estático de largo 10.
- Guarde los números impares del arreglo estático en un arreglo dinámico.
- Del arreglo dinámico, indique cuál es el número mayor.

Array dinámicos



Ejercicio





# Ejercicios

Variable local y global



Array



Array dinámicos



Ejercicio

Desarrolle en su casa los siguientes ejercicios a modo de estudio.

1. Desafío propuesto:

1. Ordene de mayor a menor un arreglo de números enteros, cuyo tamaño y cantidad de elementos sea dada por el usuario.
2. En un arreglo(1) de arreglos(2), sume los elementos de cada arreglo(2) por separado, e indique cuál de ellos genera la mayor suma. El tamaño y cantidad de elementos debe ser dado por el usuario.

