

Memoria Estática

Punteros, Vectores y Matrices

David González Márquez

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Notas sobre la instrucción: lea

- La instrucción lea quiere decir *load effective address*.

lea **dst**, [**src**]

Notas sobre la instrucción: lea

- La instrucción lea quiere decir *load effective address*.

lea **dst**, [**src**]

- Toma exactamente dos parámetros:
 - [**src**] es siempre memoria
 - **dst** siempre un registro.
- Calcula la dirección que sería accedida por [**src**] y almacena el valor en **dst**

Notas sobre la instrucción: lea

- La instrucción lea quiere decir *load effective address*.

lea **dst**, [**src**]

- Toma exactamente dos parámetros:
 - [**src**] es siempre memoria
 - **dst** siempre un registro.
- Calcula la dirección que sería accedida por [**src**] y almacena el valor en **dst**
- Ejemplos:
lea rax, [rdx*4+rcx]

Notas sobre la instrucción: lea

- La instrucción lea quiere decir *load effective address*.

lea **dst**, [**src**]

- Toma exactamente dos parámetros:
 - [**src**] es siempre memoria
 - **dst** siempre un registro.
- Calcula la dirección que sería accedida por [**src**] y almacena el valor en **dst**
- Ejemplos:

lea rax, [rdx*4+rcx] \rightarrow $\text{rax} \leftarrow \text{rdx} * 4 + \text{rcx}$

lea rax, [rsi+rdi]

Notas sobre la instrucción: lea

- La instrucción lea quiere decir *load effective address*.

lea **dst**, [**src**]

- Toma exactamente dos parámetros:
 - [**src**] es siempre memoria
 - **dst** siempre un registro.
- Calcula la dirección que sería accedida por [**src**] y almacena el valor en **dst**
- Ejemplos:

lea rax, [rdx*4+rcx] → $\text{rax} \leftarrow \text{rdx} * 4 + \text{rcx}$

lea rax, [rsi+rdi] → $\text{rax} \leftarrow \text{rsi} + \text{rdi}$

lea rax, [rax*2+rax]

Notas sobre la instrucción: lea

- La instrucción lea quiere decir *load effective address*.

lea **dst**, [**src**]

- Toma exactamente dos parámetros:
 - [**src**] es siempre memoria
 - **dst** siempre un registro.
- Calcula la dirección que sería accedida por [**src**] y almacena el valor en **dst**
- Ejemplos:

lea rax, [rdx*4+rcx] → $\text{rax} \leftarrow \text{rdx} * 4 + \text{rcx}$

lea rax, [rsi+rdi] → $\text{rax} \leftarrow \text{rsi} + \text{rdi}$

lea rax, [rax*2+rax] → $\text{rax} \leftarrow \text{rax} * 3$

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
(ejemplo: una variable cuyo valor es una dirección de memoria)

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
(ejemplo: una variable cuyo valor es una dirección de memoria)
- Tiene un tipo y un nombre.

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
(ejemplo: una variable cuyo valor es una dirección de memoria)
- Tiene un tipo y un nombre.
- Almacena una dirección de memoria.

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
(ejemplo: una variable cuyo valor es una dirección de memoria)
- Tiene un tipo y un nombre.
- Almacena una dirección de memoria.
- Sirve para referenciar una posición de memoria.

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
(ejemplo: una variable cuyo valor es una dirección de memoria)
- Tiene un tipo y un nombre.
- Almacena una dirección de memoria.
- Sirve para referenciar una posición de memoria.
- Operadores:

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
(ejemplo: una variable cuyo valor es una dirección de memoria)
- Tiene un tipo y un nombre.
- Almacena una dirección de memoria.
- Sirve para referenciar una posición de memoria.
- Operadores:
 - **&** → Da como resultado la dirección de memoria de una variable.

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
(ejemplo: una variable cuyo valor es una dirección de memoria)
- Tiene un tipo y un nombre.
- Almacena una dirección de memoria.
- Sirve para referenciar una posición de memoria.
- Operadores:
 - **&** → Da como resultado la dirección de memoria de una variable.
 - ***** → Da como resultado el valor apuntado por un puntero.
(Además de ser el indicador del tipo puntero)

Repaso de punteros

Ejemplos:

- `int *pepe`

Repaso de punteros

Ejemplos:

- `int *pepe`

Declara un puntero de tipo entero con nombre *pepe*.

Repaso de punteros

Ejemplos:

- `int *pepe`

Declara un puntero de tipo entero con nombre *pepe*.

- `int x = 5`
`pepe = &x`

Repaso de punteros

Ejemplos:

- `int *pepe`

Declara un puntero de tipo entero con nombre *pepe*.

- `int x = 5`
`pepe = &x`

Guarda en el puntero *pepe* la dirección de *x*. Se dice que *pepe* apunta a *x*.

Repaso de punteros

Ejemplos:

- `int *pepe`

Declara un puntero de tipo entero con nombre *pepe*.

- `int x = 5`
`pepe = &x`

Guarda en el puntero *pepe* la dirección de *x*. Se dice que *pepe* apunta a *x*.

- `*pepe = 8`

Repaso de punteros

Ejemplos:

- `int *pepe`

Declara un puntero de tipo entero con nombre *pepe*.

- `int x = 5`
`pepe = &x`

Guarda en el puntero *pepe* la dirección de *x*. Se dice que *pepe* apunta a *x*.

- `*pepe = 8`

Guarda 8 en la posición apuntada por el puntero *pepe*.

Repaso de punteros

Ejemplos:

- `int *pepe`

Declara un puntero de tipo entero con nombre *pepe*.

- `int x = 5`
`pepe = &x`

Guarda en el puntero *pepe* la dirección de *x*. Se dice que *pepe* apunta a *x*.

- `*pepe = 8`

Guarda 8 en la posición apuntada por el puntero *pepe*.

- `int y`
`y = *pepe`

Guarda en *y* el valor apuntado por *pepe*.

Repaso de punteros

Ejemplos:

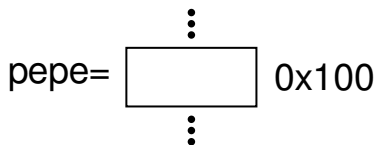
① `int *pepe`

② `int x = 5`
`pepe = &x`

③ `*pepe = 8`

④ `int y`
`y = *pepe`

1.



Repaso de punteros

Ejemplos:

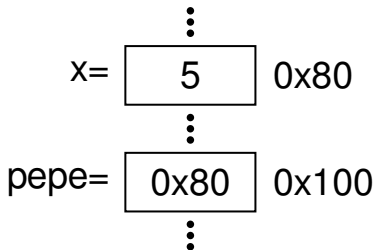
① `int *pepe`

② `int x = 5`
`pepe = &x`

③ `*pepe = 8`

④ `int y`
`y = *pepe`

2.



Repaso de punteros

Ejemplos:

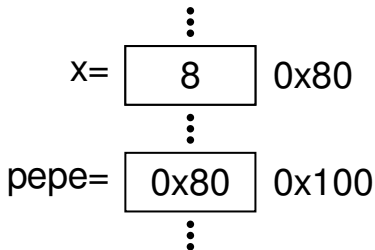
① `int *pepe`

② `int x = 5`
`pepe = &x`

③ `*pepe = 8`

④ `int y`
`y = *pepe`

3.



Repaso de punteros

Ejemplos:

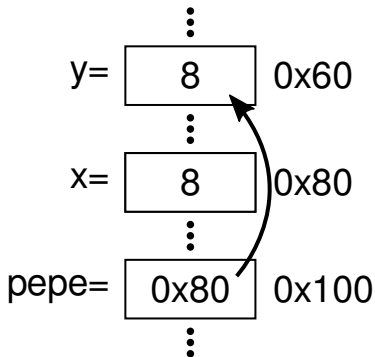
① `int *pepe`

② `int x = 5`
`pepe = &x`

③ `*pepe = 8`

④ `int y`
`y = *pepe`

4.



Vectores

Un vector o arreglo es una secuencia ordenada de elementos consecutivos en memoria de un tamaño fijo.

A_0	A_1	A_2	\cdots	A_{n-3}	A_{n-2}	A_{n-1}
-------	-------	-------	----------	-----------	-----------	-----------

Vectores

Declaremos un vector v en C :

Vectores

Declaremos un vector v en C:

```
int v[5];
```

¿Cómo está guardado en memoria?

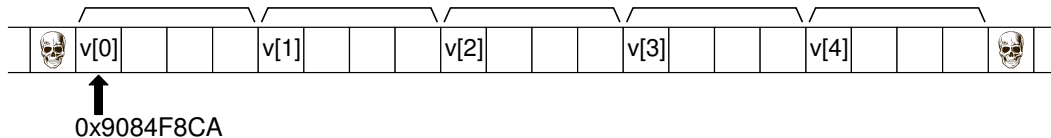
Vectores

Declaremos un vector v en C:

```
int v[5];
```

¿Cómo está guardado en memoria?

Como 5 enteros (*doublewords* / 4 bytes) **consecutivos**:



Vectores

Si rememoramos el ejemplo de la primera clase:

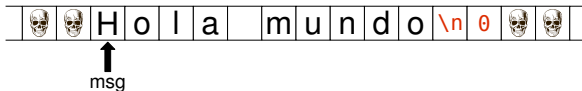
```
section .rodata:  
msg: DB 'Hola mundo', 10, 0  
largo: EQU $-msg-1
```

Vectores

Si rememoramos el ejemplo de la primera clase:

```
section .rodata:  
msg: DB 'Hola mundo', 10, 0  
largo: EQU $-msg-1
```

msg es una etiqueta que, vista como un puntero, es un vector de caracteres almacenados de la siguiente manera:

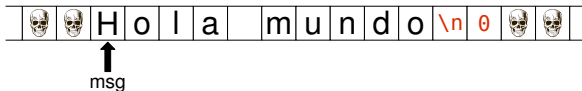


Vectores

Si rememoramos el ejemplo de la primera clase:

```
section .rodata:  
msg: DB 'Hola mundo', 10, 0  
largo: EQU $-msg-1
```

msg es una etiqueta que, vista como un puntero, es un vector de caracteres almacenados de la siguiente manera:



msg es un `char*`, es como si en C hiciéramos:

```
char msg[11] = "Hola mundo\n";
```


Vectores

Dado:

```
int v[5];
```

Vectores

Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

Vectores

Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```

Vectores

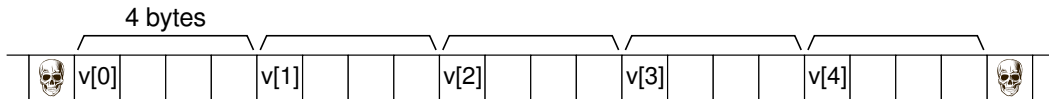
Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```



Vectores

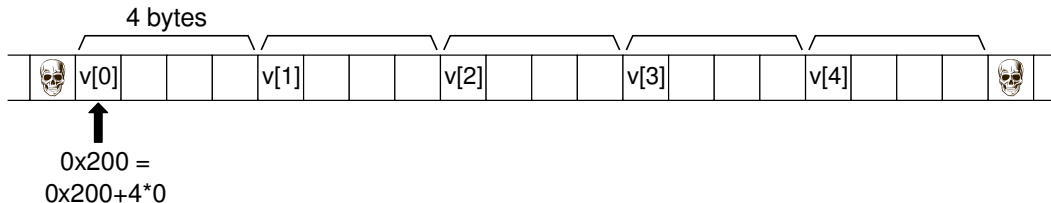
Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```



Vectores

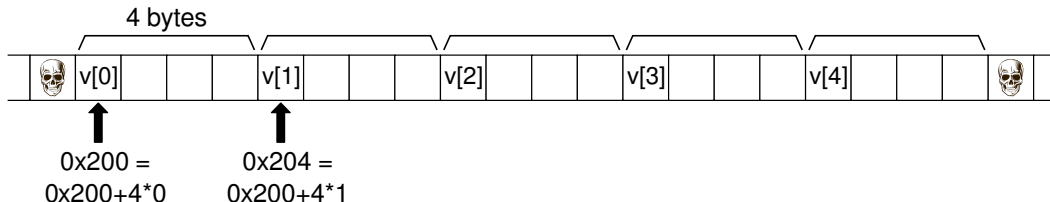
Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```



Vectores

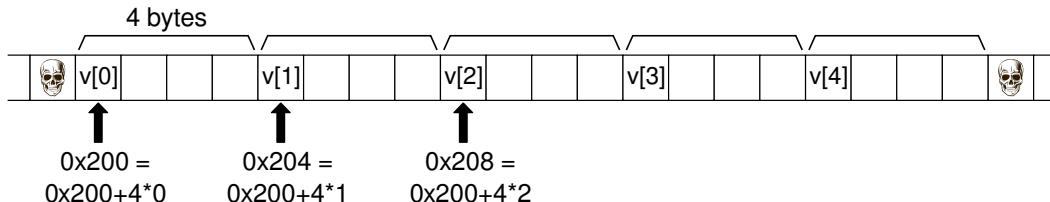
Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```



Vectores

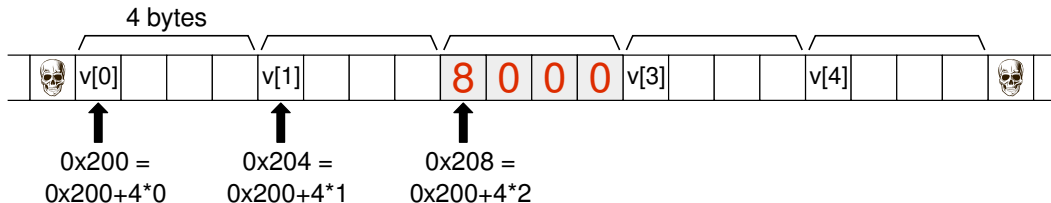
Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```



Vectores

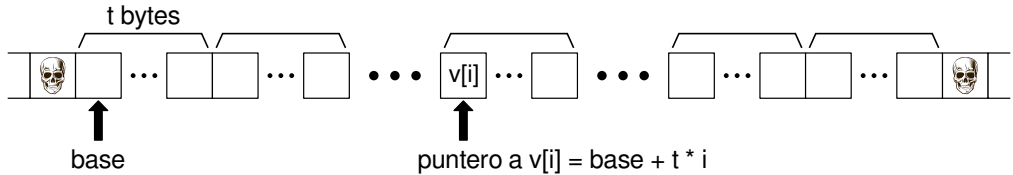
En general:

puntero al inicio + tamaño del dato * índice del elemento

Vectores

En general:

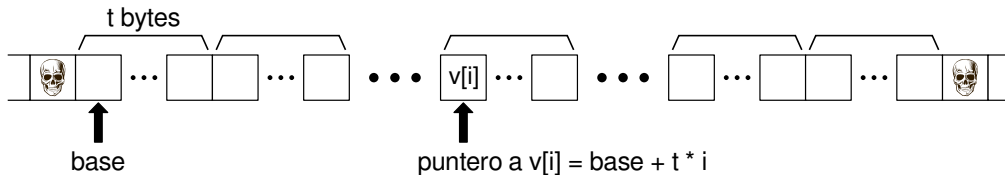
puntero al inicio + tamaño del dato * índice del elemento



Vectores

En general:

puntero al inicio + tamaño del dato * índice del elemento



En Intel:

Si el tamaño de los elementos es un valor válido como escala,

$[\text{<base>} + \text{<índice>} * \text{<scale>}] \xrightarrow{\text{ejemplo}} [\text{rax+rbx*4}]$

Vectores y Punteros

Si tenemos:

```
int v[5];
```

`v` es un puntero al primer elemento del vector.

Vectores y Punteros

Si tenemos:

```
int v[5];
```

`v` es un puntero al primer elemento del vector.

Luego, vale en C:

```
int *p_v = v; ← tomo el puntero al vector
```

ó

```
int *p_v = &v[0]; ← tomo la dirección del primer elemento
```

Matrices

- Se representan en memoria como un **vector de vectores**.

Matrices

- Se representan en memoria como un **vector de vectores**.
- Si la matriz tiene dimensión $M \times N$ entonces sabemos que está formada por M vectores de N elementos cada uno.

Matrices

- Se representan en memoria como un **vector de vectores**.
- Si la matriz tiene dimensión $M \times N$ entonces sabemos que está formada por M vectores de N elementos cada uno.
- En C, las matrices se almacenan por filas.

Matrices

Ejemplo

Si tenemos M , una matriz de enteros de 3×3 :

$m[0][0]$	$m[0][1]$	$m[0][2]$
$m[1][0]$	$m[1][1]$	$m[1][2]$
$m[2][0]$	$m[2][1]$	$m[2][2]$

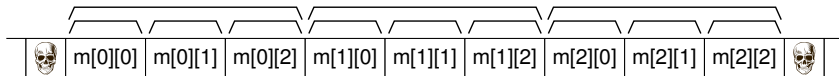
Matrices

Ejemplo

Si tenemos M , una matriz de enteros de 3×3 :

$m[0][0]$	$m[0][1]$	$m[0][2]$
$m[1][0]$	$m[1][1]$	$m[1][2]$
$m[2][0]$	$m[2][1]$	$m[2][2]$

En memoria se representa:



Matrices

Sí el primer elemento de M se almacena en la dirección $0x300$.

Y queremos asignar el valor 7 en $M[2, 1]$ en C:

Matrices

Sí el primer elemento de M se almacena en la dirección $0x300$.

Y queremos asignar el valor 7 en $M[2, 1]$ en C:

```
int m[3][3];  
m[2][1] = 7;
```

Matrices

Sí el primer elemento de M se almacena en la dirección $0x300$.

Y queremos asignar el valor 7 en $M[2, 1]$ en C:

```
int m[3][3];  
m[2][1] = 7;
```

¿Y en ensamblador?

Matrices

Sí el primer elemento de M se almacena en la dirección $0x300$.

Y queremos asignar el valor 7 en $M[2, 1]$ en C:

```
int m[3][3];  
m[2][1] = 7;
```

4 bytes

m[0][0]	m[0][1]	m[0][2]
m[1][0]	m[1][1]	m[1][2]
m[2][0]	m[2][1]	m[2][2]

¿Y en ensamblador?

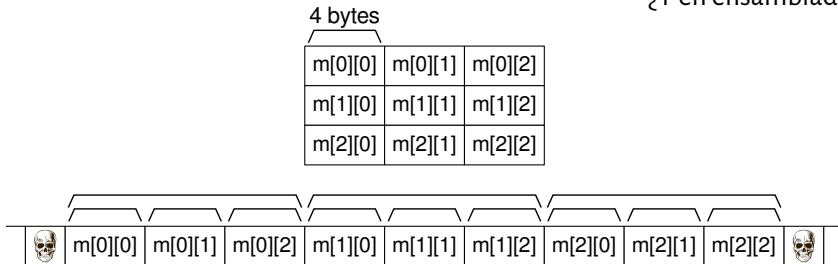
Matrices

Sí el primer elemento de M se almacena en la dirección $0x300$.

Y queremos asignar el valor 7 en $M[2, 1]$ en C:

```
int m[3][3];  
m[2][1] = 7;
```

¿Y en ensamblador?



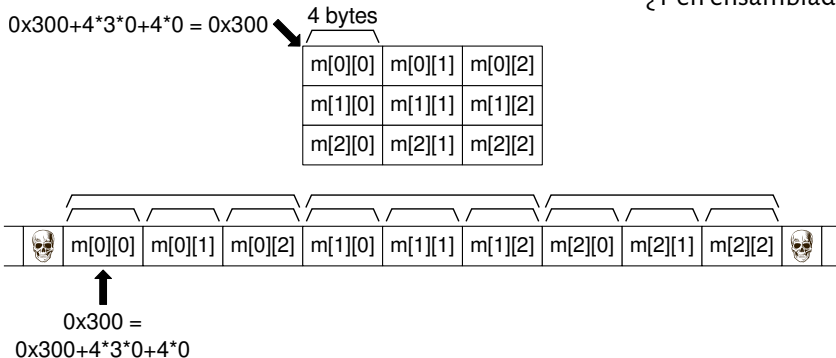
Matrices

Sí el primer elemento de M se almacena en la dirección $0x300$.

Y queremos asignar el valor 7 en $M[2, 1]$ en C:

```
int m[3][3];  
m[2][1] = 7;
```

¿Y en ensamblador?



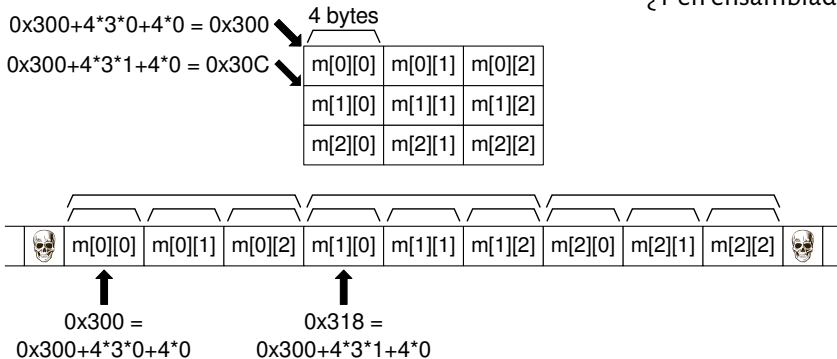
Matrices

Sí el primer elemento de M se almacena en la dirección $0x300$.

Y queremos asignar el valor 7 en $M[2, 1]$ en C:

```
int m[3][3];  
m[2][1] = 7;
```

¿Y en ensamblador?



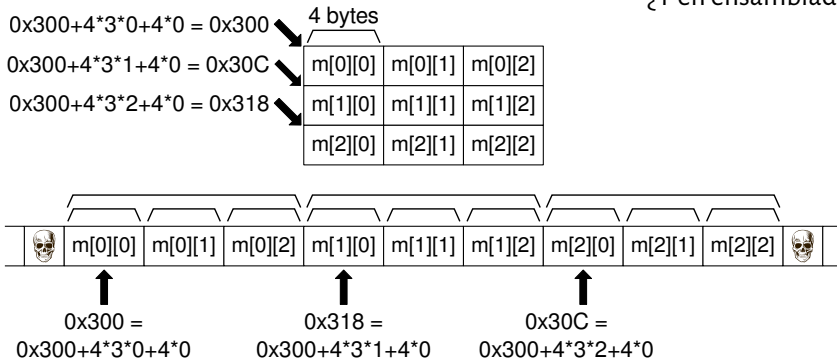
Matrices

Sí el primer elemento de M se almacena en la dirección $0x300$.

Y queremos asignar el valor 7 en $M[2, 1]$ en C:

```
int m[3][3];  
m[2][1] = 7;
```

¿Y en ensamblador?



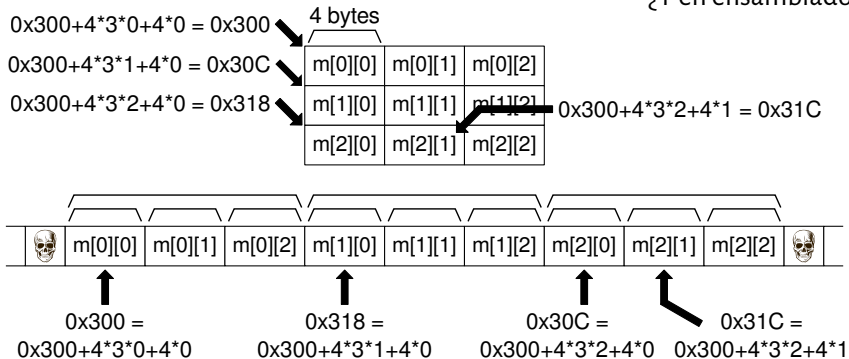
Matrices

Sí el primer elemento de M se almacena en la dirección $0x300$.

Y queremos asignar el valor 7 en $M[2, 1]$ en C:

```
int m[3][3];  
m[2][1] = 7;
```

¿Y en ensamblador?



Matrices

En general, para una matriz $M[i, j]$:

puntero al inicio	+	tamaño dato	*	índice fila	*	tamaño fila
	+	tamaño dato	*	índice columna		

Matrices

En general, para una matriz $M[i, j]$:

puntero al inicio	+	tamaño dato	*	índice fila	*	tamaño fila
	+	tamaño dato	*	índice columna		

En Intel:

Si el tamaño de los elementos es un valor válido como escala,

Matrices

En general, para una matriz $M[i, j]$:

puntero al inicio	+	tamaño dato	*	índice fila	*	tamaño fila
	+	tamaño dato	*	índice columna		

En Intel:

Si el tamaño de los elementos es un valor válido como escala,

Primero obtengo el offset de la fila

$\langle \text{índiceFila} \rangle * \langle \text{tamañoDato} * \text{tamañoFila} \rangle$

ejemplo

→ `mul rax, rdx ; ojo! modifica rdx también`

Matrices

En general, para una matriz $M[i, j]$:

puntero al inicio + tamaño dato * índice fila * tamaño fila
+ tamaño dato * índice columna

En Intel:

Si el tamaño de los elementos es un valor válido como escala,

Primero obtengo el offset de la fila

$\langle \text{índiceFila} \rangle * \langle \text{tamañoDato} * \text{tamañoFila} \rangle$

ejemplo

→ `mul rax, rdx ; ojo! modifica rdx también`

Segundo el offset dentro de la fila

$[\langle \text{índiceFila} * \text{tamañoDato} * \text{tamañoFila} \rangle + \langle \text{índiceColumna} \rangle * \langle \text{tamañoDato} \rangle]$

ejemplo

→ `lea rsi, [rax+rcx*2]; rsi <= rax+rcx*2`

Matrices

En general, para una matriz $M[i, j]$:

puntero al inicio + tamaño dato * índice fila * tamaño fila
+ tamaño dato * índice columna

En Intel:

Si el tamaño de los elementos es un valor válido como escala,

Primero obtengo el offset de la fila

$\langle \text{índiceFila} \rangle * \langle \text{tamañoDato} * \text{tamañoFila} \rangle$

ejemplo → `mul rax, rdx ; ojo! modifica rdx también`

Segundo el offset dentro de la fila

$[\langle \text{índiceFila} * \text{tamañoDato} * \text{tamañoFila} \rangle + \langle \text{índiceColumna} \rangle * \langle \text{tamañoDato} \rangle]$

ejemplo → `lea rsi, [rax+rcx*2]; rsi <= rax+rcx*2`

Tercero, accedo al dato

$[\langle \text{base} \rangle + \langle \text{índiceFila} * \text{tamañoDato} * \text{tamañoFila} + \text{índiceColumna} * \text{tamañoDato} \rangle]$

ejemplo → `mov rdx, [rbx+rsi]`

Matrices y Punteros

Si tenemos:

```
int m[3][4];
```

Matrices y Punteros

Si tenemos:

```
int m[3][4];
```

m es un puntero al primer elemento de la matriz.

Luego, vale en C:

```
int *p_m = (int*)m; ← tomo el puntero a la matriz
```

ó

```
int *p_m = &m[0][0]; ← tomo la dirección del primer dato
```

Notación en C

Declaración y *cast* de un puntero a un puntero a matriz:

```
int (*matrix)[rowSize] = (int (*)(rowSize)) p;
```

Notación en C

Declaración y *cast* de un puntero a un puntero a matriz:

```
int (*matrix)[rowSize] = (int (*)(rowSize)) p;
```

Se declara la variable `matrix` como un **puntero a una matriz**.

Notación en C

Declaración y *cast* de un puntero a un puntero a matriz:

```
int (*matrix)[rowSize] = (int (*)(rowSize)) p;
```

Se declara la variable `matrix` como un **puntero a una matriz**.

`p` es un **puntero a memoria** que se transforma a matriz.

Notación en C

Declaración y *cast* de un puntero a un puntero a matriz:

```
int (*matrix)[rowSize] = (int (*)(rowSize)) p;
```

Se declara la variable `matrix` como un **puntero a una matriz**.

`p` es un **puntero a memoria** que se transforma a matriz.

`rowSize` es la cantidad de datos en una fila (columnas)

Notación en C

Declaración y *cast* de un puntero a un puntero a matriz:

```
int (*matrix)[rowSize] = (int (*)(rowSize)) p;
```

Se declara la variable `matrix` como un **puntero a una matriz**.

`p` es un **puntero a memoria** que se transforma a matriz.

`rowSize` es la cantidad de datos en una fila (columnas)

Como es un puntero, no se declara la cantidad total de filas.

Notación en C

Declaración y *cast* de un puntero a un puntero a matriz:

```
int (*matrix)[rowSize] = (int (*)(rowSize)) p;
```

Se declara la variable `matrix` como un **puntero a una matriz**.

`p` es un **puntero a memoria** que se transforma a matriz.

`rowSize` es la cantidad de datos en una fila (columnas)

Como es un puntero, no se declara la cantidad total de filas.

Esta sintaxis se puede utilizar para declarar matrices de N dimensiones.

```
t (*m)[a]...[z] = (t (*)(a)...[z]) p;
```


Ejercicios

Ejercicio 1

```
short suma(short* vector, short n);
```

Dado un vector de n enteros de 16 bits, devolver la suma de los elementos.

Ejercicio 2

```
void diagonal(short* matriz, short n, short* vector);
```

Dada una matriz de $n \times n$ enteros de 16 bits, devolver los elementos de la diagonal en el vector pasado por parámetro.

Ejercicio 3

```
void gris(pixel* matriz, short n, uint8_t* resultado)
```

Dada una matriz de $n \times n$ pixeles RGB (1 byte por componente), hacer una función que convierta los pixeles a escala de grises usando la formula $(R + 2 \cdot G + B)/4$.

Ejercicio 4

```
int* primerMaximo(int (*matriz)[sizeC], int* f, int* c);
```

Dada una matriz de $f \times c$ enteros de 32 bits, encontrar el primer máximo buscando en el orden de la memoria. Devuelve un puntero a este valor y sus coordenadas en f y c

Bibliografía: Fuentes y material adicional

- Convenciones de llamados a función en x86:
https://en.wikipedia.org/wiki/X86_calling_conventions
- Notas sobre System V ABI:
https://wiki.osdev.org/System_V_ABI
- Documentación de NASM:
<https://nasm.us/doc/>
- Artículo sobre el flag -pie:
<https://eklitzke.org/position-independent-executables>
- Documentación de System V ABI:
https://uclibc.org/docs/psABI-x86_64.pdf
- Manuales de Intel:
<https://software.intel.com/en-us/articles/intel-sdm>

¡Gracias!

Recuerden leer los comentarios al final de este video por aclaraciones o fe de erratas.