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]      →      rax ← rdx * 4 + rcx
lea rax, [rsi+rdi]        →      rax ← rsi + rdi
lea rax, [rax*2+rax]      →      rax ← 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)
- 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`

  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`

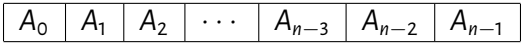
  Guarda 8 en la posición apuntada por el puntero *pepe*.

- ```
  int y
  y = *pepe
  ```

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

Vectores

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



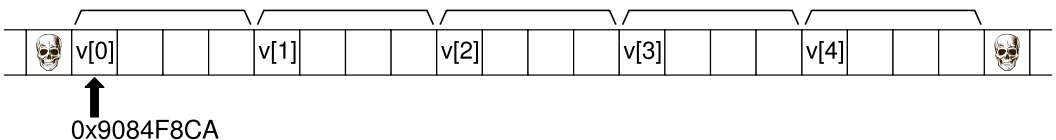
Vectores

Declaremos un vector *v* en C:

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

¿Cómo está guardado en memoria?

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

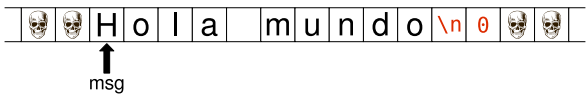


Vectores

Si rememoramos el ejemplo de la primera clase:

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

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



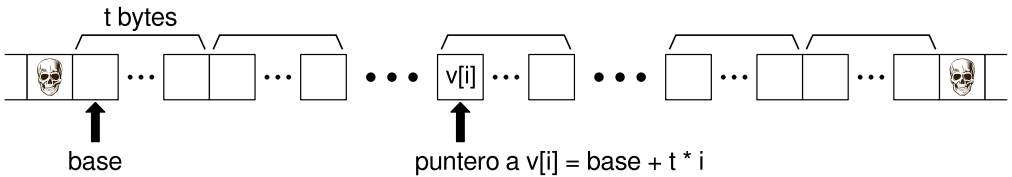
msg es un char*, es como si en C hiciéramos:

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

Vectores

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,

`[ <base> + <indice> * <scale> ]` —ejemplo→ `[rax+rbx*4]`

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**.
- Si la matriz tiene dimensión *MxN* entonces sabemos que está formada por *M* vectores de *N* elementos cada uno.
- En C, las matrices se almacenan por filas.

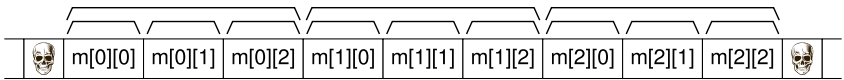
Matrices

Ejemplo

Si tenemos *M*, una matriz de enteros de 3*x*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:

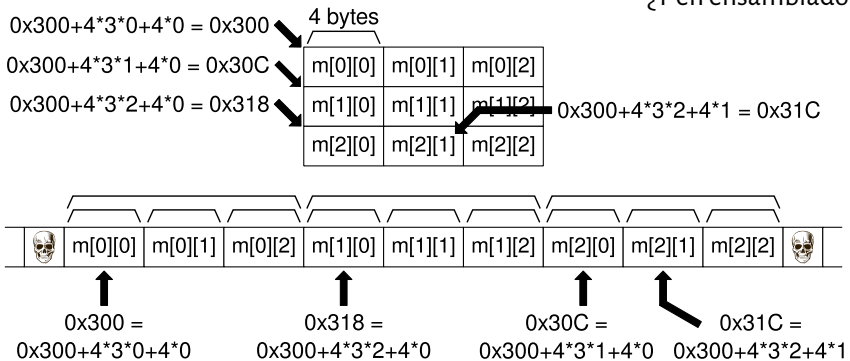


Matrices

Sí el primer elemento de M se almacena en la dirección `0x300`.
Y queremos asignar el valor 7 en $M[2,1]$ en C:

```
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
```

En Intel:

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

Primero obtengo el offset de la fila

```
<índiceFila> * <tamañoDato*tamañoFila>
ejemplo ⟶ mul rax, rdx ; ojo! modifica rdx también
```

Segundo el offset dentro de la fila

```
[<índiceFila*tamañoDato*tamañoFila> + <índiceColumna> * <tamañoDato>]
ejemplo ⟶ lea rsi, [rax+rcx*2]; rsi <= rax+rcx*2
```

Tercero, accedo al dato

```
[<base> + <índiceFila*tamañoDato*tamañoFila+índiceColumna*tamañoDato>]
ejemplo ⟶ mov rdx, [rbx+rsi]
```

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;
```