

# Lenguaje de máquina: datos

Organización del computador - FIUBA

---

2.<sup>do</sup> cuatrimestre de 2023

Última modificación: Sun Oct 8 20:51:04 2023 -0300

## Créditos

Para armar las presentaciones del curso utilizamos:



R. E. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective*, Third edition, Global edition. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2015.



D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2017.



J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2017.

El contenido de los slides está basado en las presentaciones de Patricio Moreno y de Organización del Computador I - FCEN.

## Tabla de contenidos

---

### 1. Arreglos

- Unidimensionales

- Multidimensionales

- Multinivel

- Tipos de matrices

### 2. Estructuras

- Asignación de memoria

- Accesos

- Alineamiento

### 3. Punto Flotante

## Tabla de contenidos

---

### 1. Arreglos

Unidimensionales

Multidimensionales

Multinivel

Tipos de matrices

### 2. Estructuras

Asignación de memoria

Accesos

Alineamiento

### 3. Punto Flotante

# Tabla de contenidos

---

## 1. Arreglos

Unidimensionales

Multidimensionales

Multinivel

Tipos de matrices

## 2. Estructuras

Asignación de memoria

Accesos

Alineamiento

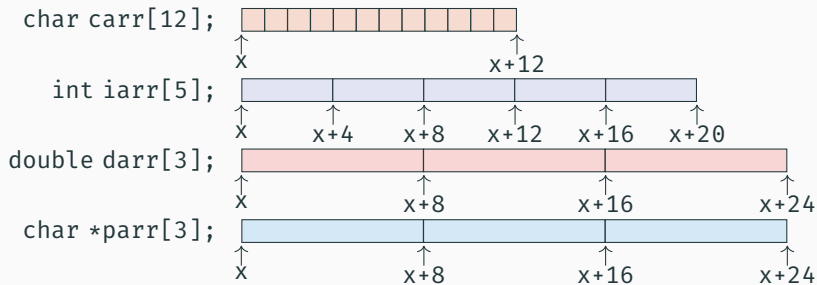
## 3. Punto Flotante

## Asignación de memoria

### Declaración

> `T A[L];`

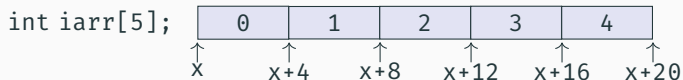
- Arreglo de largo `L` del tipo de dato `T`
- Asignado en una región de memoria *continua* de tamaño `L*sizeof(T)`



## Acceso al arreglo

### Declaración

- `T A[L];` // Arreglo de largo L del tipo de dato T
- El símbolo **A** se puede usar como un punto al primer elemento del arreglo  $\Rightarrow$  su tipo es `T*`

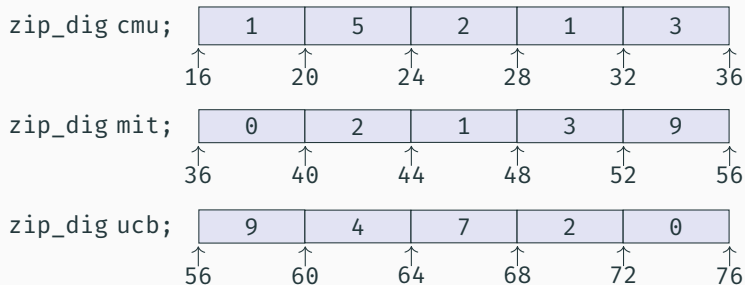


Referencia	Tipo	Valor	
<code>iarr[4]</code>	<code>int</code>	4	
<code>iarr</code>	<code>int *</code>	x	
<code>iarr + 1</code>	<code>int *</code>	x + 4	
<code>&amp;iarr[2]</code>	<code>int *</code>	x + 8	
<code>iarr[5]</code>	<code>int</code>	??	
<code>*(iarr + 1)</code>	<code>int</code>	1	<code>//iarr[1]</code>
<code>iarr + i</code>	<code>int *</code>	x + 4*i	<code>//&amp;iarr[i]</code>

## Ejemplo

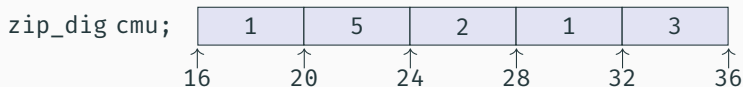
```
#define ZLEN 5
typedef int zip_dig[EZLN];

zip_dig cmu = {1, 5, 2, 1, 3};
zip_dig mit = {0, 2, 1, 3, 9};
zip_dig ucb = {9, 4, 7, 2, 0};
```





## Ejemplo de acceso



### Lenguaje C

```
int get_digit(zip_dig z, int digit)
{
    return z[digit];
}
```

### Lenguaje assembly (x86-64)

```
get_digit:
    # rdi = z
    # esi = digit
    ; movs    rsi, esi
    mov     eax, [rdi + rsi*4]
    ret
```

- El registro **rdi** contiene la dirección donde comienza el arreglo
- El registro **rsi** (**esi**) guarda el índice
- El dígito pedido se encuentra en **rdi + 4\*rsi**
- Se usa la referencia **[rdi+rsi\*4]**

## Ejemplo de acceso en ciclos

```
void zinc(zip_dig z) {
    size_t i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```
    lea    rax, [rdi + 20]
.L2:
    add    dword [rdi], 1
    add    rdi, 4
    cmp    rdi, rax
    jne    .L2
```

```
    xor    eax, eax
.L2:
    add    dword [rdi + rax*4], 1
    add    rax, 1
    cmp    rax, 5
    jne    .L2
```

```
zinc:
    mov    eax, 0
    jmp    .L2
.L3:
    lea    rcx, [rdi + rax*4]
    mov    esi, [rcx]
    lea    edx, [esi + 1]
    mov    [rcx], edx
    add    rax, 1
.L2:
    cmp    rax, 4
    jbe    .L3
    ret
```

# Tabla de contenidos

---

## 1. Arreglos

Unidimensionales

**Multidimensionales**

Multinivel

Tipos de matrices

## 2. Estructuras

Asignación de memoria

Accesos

Alineamiento

## 3. Punto Flotante

## Arreglos multidimensionales (anidados)

### Declaración

```
> T A[R][C];
```

- Arreglo 2D del tipo de dato T
- R filas, C columnas

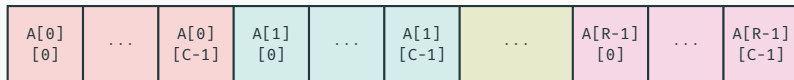
### Tamaño

$R * C * \text{sizeof}(T)$  bytes

### Disposición en memoria

- Memoria *continua*
- *Row-Major Ordering*

$$\begin{bmatrix} A[0][0] & \dots & A[0][C-1] \\ A[1][0] & \dots & A[1][C-1] \\ \vdots & \ddots & \vdots \\ A[R-1][0] & \dots & A[R-1][C-1] \end{bmatrix}$$



$4 * R * C$  bytes

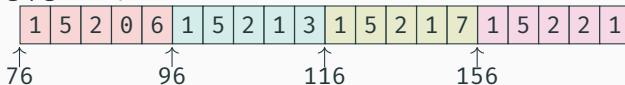
```
int A[R][C];
```

## Ejemplo de arreglos anidados

```
#define PCOUNT 4
typedef int zip_dig[5];

zip_dig pgh[PCOUNT] = {
    {1, 5, 2, 0, 6},
    {1, 5, 2, 1, 3}, // a la vista es una matriz 2D
    {1, 5, 2, 1, 7},
    {1, 5, 2, 2, 1}};
```

zip\_dig pgh[4];

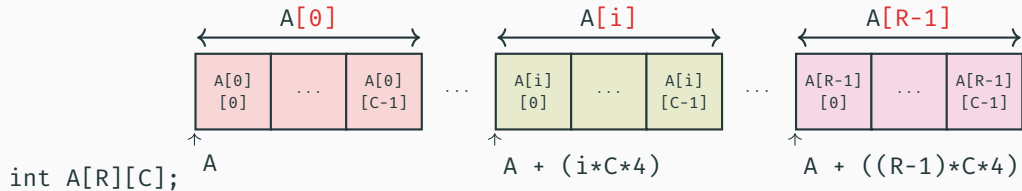


- zip\_dig pgh[4] es equivalente a int pgh[4][5]
  - pgh: es un arreglo de 4 *elementos*
  - cada *elemento* es un arreglo de 5 ints

## Acceso al arreglo

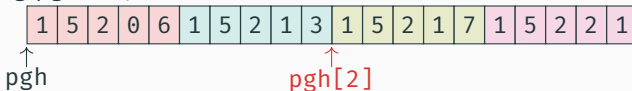
### Vectores fila

- $A[i]$  es un arreglo de  $C$  elementos del tipo de dato  $T$
- Comienza en la dirección  $A + i * (C * \text{sizeof}(T))$



## Ejemplo de acceso a arreglos anidados

zip\_dig pgh[4];



```
extern int pgh[4][5];
```

```
int * get_pgh_zip(unsigned long index) {
    return pgh[index]; // global
}
```

-Og

get\_pgh\_zip:

```
    lea    rdx, [rdi + 4*rdi]
    lea    rax, [rdx*4]
    add    rax, qword [pgh]
    ret
```

-O1

get\_pgh\_zip:

```
    lea rax, [rdi + 4*rdi]
    lea rax, [pgh + 4*rax]
    ret
```

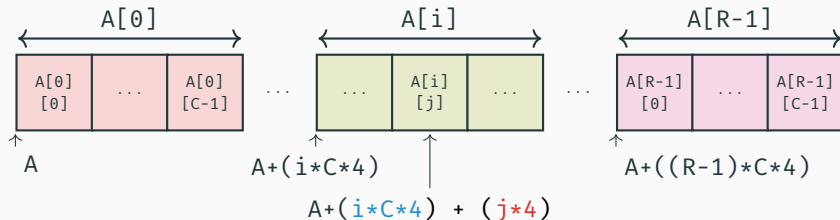
En ambos casos se calcula  $\text{pgh} + 4 * (\text{index} + 4 * \text{index})$

## Acceso a elementos del arreglo

### Elementos del arreglo

- $A[i][j]$  es un elemento tipo de dato T, tamaño K bytes
- Comienza en la dirección

$$A + i*(C*K) + j*K = A + (i*C + j) * K$$

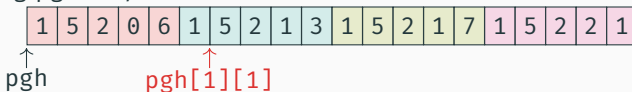


```
int A[R][C];
```



## Ejemplo de acceso a elemento en arreglos anidados

```
zip_dig pgh[4];
```



```
extern int pgh[4][5];
```

```
int * get_pgh_digit(unsigned long index, unsigned long digit) {
    return pgh[index][digit]; // global
}
```

-Og

```
get_pgh_digit:
    lea rax, [rdi + 4*rdi]
    add rax, rsi
    mov eax, DWORD [pgh + 4*rax]
    ret
```

-O1

```
get_pgh_digit:
    lea rax, [rdi + 4*rdi]
    add rax, rsi
    mov eax, DWORD [pgh + 4*rax]
    ret
```

El gcc genera:  $\text{pgh} + 4 * (\text{index} + 4 * \text{index} + \text{digit})$

## Tabla de contenidos

---

### 1. Arreglos

Unidimensionales

Multidimensionales

**Multinivel**

Tipos de matrices

### 2. Estructuras

Asignación de memoria

Accesos

Alineamiento

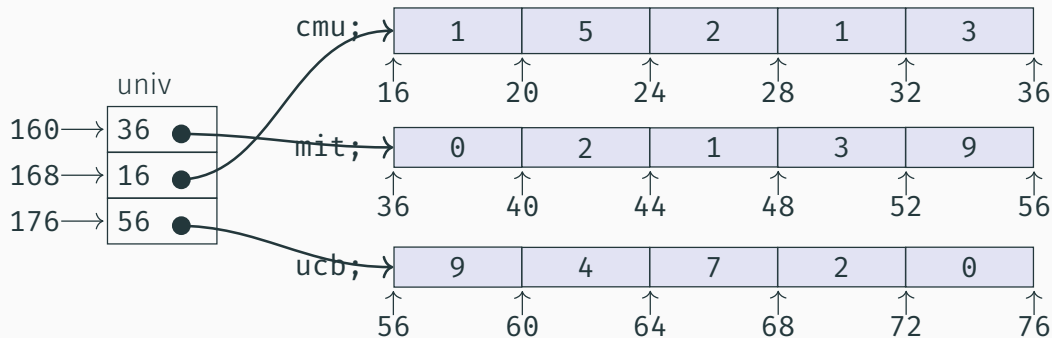
### 3. Punto Flotante

## Arreglos multinivel

```
zip_dig cmu = {1, 5, 2, 1, 3};
zip_dig mit = {0, 2, 1, 3, 9};
zip_dig ucb = {9, 4, 7, 2, 0};
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- **univ** es un arreglo de 3 *elementos*
- cada elemento es un puntero
- cada puntero apunta a un arreglo de **ints**

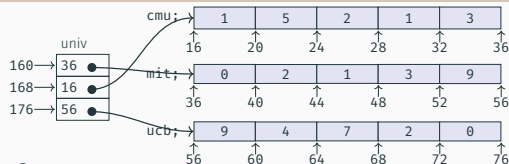


## Ejemplo de acceso a elemento en arreglos multinivel

```
int get_univ_digit
(size_t index, size_t digit)
{
    return univ[index][digit];
}
```

-Og

```
get_univ_digit:
    salq    rsi, 2
    add     rsi, [univ + rdi*8]
    mov     eax, [rsi]
    ret
```



-O1

```
get_univ_digit:
    mov rax, [univ + rdi*8] ; get pointer to
                           row
    mov eax, [rax + rsi*4]  ; get element at
                           column digit
    ret
```

- Calcula:  $\text{Mem}[\text{Mem}[\text{univ} + 8 \cdot \text{index}] * 4 \cdot \text{digit}]$
- Hace 2 accesos a memoria, 2 lecturas
  - Primero obtiene el puntero a la fila
  - La dirección de memoria del elemento **depende** del valor guardado  $\text{univ}[\text{index}]$

# Tabla de contenidos

---

## 1. Arreglos

Unidimensionales

Multidimensionales

Multinivel

Tipos de matrices

## 2. Estructuras

Asignación de memoria

Accesos

Alineamiento

## 3. Punto Flotante

## Matrices NxN

- Dimensiones fijas
  - Se conoce **N** en tiempo de compilación
- Dimensiones variables, indexado explícito
  - El arreglo unidimensional A es fijo
- Dimensiones variables, indexado implícito
  - Soporte desde C99, opcional desde C11

```
#define N 16
typedef int mfiija[N][N];
int mfiija_ij(mfiija A, size_t i, size_t j) {
    return A[i][j];
}
```

```
#define IDX(n, i, j) ((i)*(n)+(j))
int mvar_exp(size_t n, int *A,
             size_t i, size_t j) {
    return A[IDX(n, i, j)];
}
```

```
int mvar_imp(size_t n, int A[n][n],
             size_t i, size_t j) {
    return A[i][j];
}
```

## Ejemplo de acceso a matrices de tamaño variable

- Elementos del arreglo
  - `size_t n;`
  - `int A[n][n]`
  - `A[i][j]` en  $A + i*(C*K) + j*K$
  - $C = n, K = 4$
  - Sí o sí se debe hacer multiplicación de enteros (`imulq`)

```
int mvar_imp(size_t n, int A[n][n], size_t i, size_t j) {
    return A[i][j];
}
```

```
# n en rdi, A en rsi, i en rdx, j en rcx
mvar_imp:
    imul rdx, [rdi] ; n * i
    lea rax, [rsi + rdx*4] ; A + 4*n*i
    mov eax, [rax + rcx*4] ; A + 4*n*i + 4*j
    ret
```

# Tabla de contenidos

---

## 1. Arreglos

Unidimensionales

Multidimensionales

Multinivel

Tipos de matrices

## 2. Estructuras

Asignación de memoria

Accesos

Alineamiento

## 3. Punto Flotante



## Tabla de contenidos

---

### 1. Arreglos

Unidimensionales

Multidimensionales

Multinivel

Tipos de matrices

### 2. Estructuras

Asignación de memoria

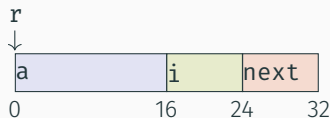
Accesos

Alineamiento

### 3. Punto Flotante

## Representación de estructuras

```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec * next;  
} r;
```



- La estructura se representa como un bloque de memoria
  - Lo suficientemente grande como para contener todos los campos
- Los campos se ordenan **de acuerdo a la declaración**
  - Incluso si otro ordenamiento brinda una representación más compacta
- El compilador determina el tamaño total y los *offsets* de los campos (y los recuerda)
  - En el lenguaje de máquina no se tiene conocimiento de las estructuras en el código fuente

# Tabla de contenidos

---

## 1. Arreglos

Unidimensionales

Multidimensionales

Multinivel

Tipos de matrices

## 2. Estructuras

Asignación de memoria

Accesos

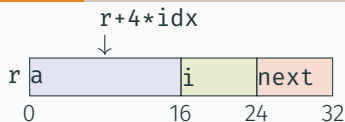
Alineamiento

## 3. Punto Flotante

## Obtención de punteros a los miembros

```
struct rec {
    int a[4];
    size_t i;
    struct rec * next;
} r;
```

- Sabiendo cómo obtener el puntero al dato, obtener el dato es trivial
  - El *offset* de cada miembro de la estructura se determina en tiempo de compilación
  - Sabemos que  $\&a = \&r$ , entonces:
    - $r + 4*idx$



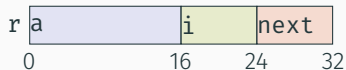
```
int *get_ap(struct rec * r, size_t idx) {
    return &r->a[idx];
}
```

```
get_ap:
    lea rax, [rdi + rsi*4]
    ret
get_ip:
    lea rax, [rdi + 16]
    ret
get_nextp:
    lea rax, [rdi + 24]
    ret
```

## Iterando con la estructura l

```
long largo(struct rec *r) {
    long len = 0L;
    while (r) {
        len++;
        r = r->next;
    }
    return len;
}
```

```
struct rec {
    int a[4];
    size_t i;
    struct rec * next;
} r;
```



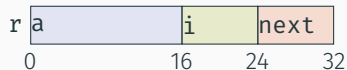
### Código assembly

```
    mov     eax, 0
    jmp     .L2
.L3:
    add     rax, 1
    mov     rdi, [rdi + 24]
.L2:
    test    rdi, rdi
    jne     .L3
```

## Iterando con la estructura II

```
void set_a(struct rec *r,
           int v) {
    while (r) {
        size_t i = r->i;
        r->a[i] = v;
        r = r->next;
    }
}
```

```
struct rec {
    int a[4];
    size_t i;
    struct rec * next;
} r;
```



### Código assembly

```
    jmp .L5
.L6:
    mov     rax, [rdi + 16]
    mov     dword [rdi + rax*4], esi
    mov     rdi, [rdi + 24]
.L5:
    test    rdi, rdi
    jne     .L6
```

# Tabla de contenidos

---

## 1. Arreglos

Unidimensionales

Multidimensionales

Multinivel

Tipos de matrices

## 2. Estructuras

Asignación de memoria

Accesos

Alineamiento

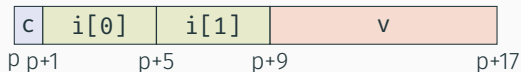
## 3. Punto Flotante

## Alineamiento de estructuras

### Datos no alineados

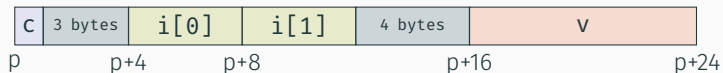
```
struct st_a {
    char c;
    int i[2];
    double v;
} *p;
```

- Todos los datos se guardan “pegados”, independientemente de tamaños.



### Datos alineados

- Si el tipo de dato es de **B** bytes, entonces su dirección debe ser múltiplo de **B**.





# Principios del alineamiento

- Alineación de datos
  - Si el dato requiere **B** bytes
  - La dirección donde comienza debe ser múltiplo de **B**
  - Es requisito en algunas arquitecturas; aconsejado en x86-64
- Motivación
  - La memoria se accede de a pedazos (alineados) de 4 u 8 bytes (dependiente del sistema)
    - Es ineficiente cargar o guardar datos que se extienden por más de una línea de cache (64 bytes)  
Intel recomienda no superar los límites de 16 bytes.
    - La memoria virtual es más complicada cuando los datos abarca 2 páginas (páginas: 4 KB)
- El compilador
  - Inserta huecos en la estructura para asegurar el correcto alineamiento de los datos

## Casos específicos de alineamiento

### x86-64

- 1 byte: **char**
  - no tiene restricciones en la dirección
- 2 byte: **short**
  - el bit más bajo de la dirección debe ser  $0_2$
- 4 byte: **int, float**
  - los 2 bits más bajos de la dirección deben ser  $00_2$
- 8 byte: **double, long char \*, ...**
  - los 3 bits más bajos de la dirección deben ser  $000_2$

## Requerimientos de alineamiento en estructuras

### Internamente

- Se deben satisfacer los requerimientos de cada elemento

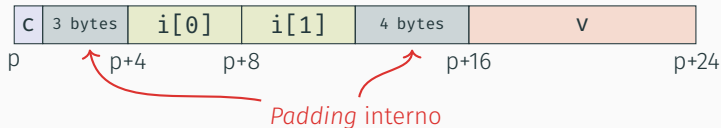
```
struct st_a {
    char c;
    int i[2];
    double v;
} *p;
```

### Ubicación global de la estructura

- Sea  $K$  el requerimiento de alineamiento de la estructura, donde  $K$  es el máximo requerimiento de los elementos internos
- La dirección y el largo de la estructura deben ser múltiplos de  $K$

### Ejemplo

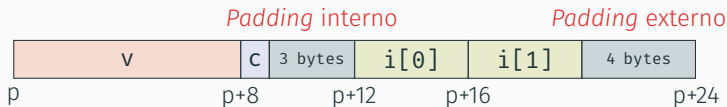
- $K = 8$  debido al `double` ;  $K \leq \text{sizeof}(\text{struct st\_a})$



## Cumpliendo con los requisitos de alineamiento I

- Se debe cumplir con los requisitos de alineamiento interno y con el alineamiento global de la estructura con el requerimiento  $K$  de alineamiento máximo de sus elementos

```
struct st_b {  
    double v;  
    char c;  
    int i[2];  
} *p;
```

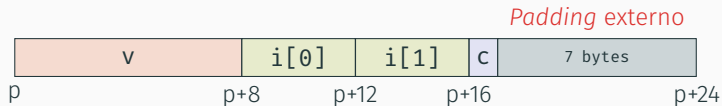


El estándar garantiza que nunca hay *padding* al inicio de una estructura

## Cumpliendo con los requisitos de alineamiento II

- Se debe cumplir el alineamiento global de la estructura con el requerimiento  $K$  de alineamiento máximo de sus elementos

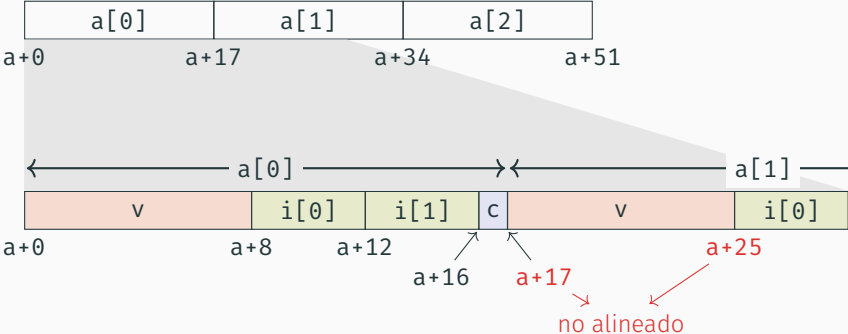
```
struct st_c {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



# Arreglos de estructuras

- Aquí se ve por qué es necesario el *padding* externo.

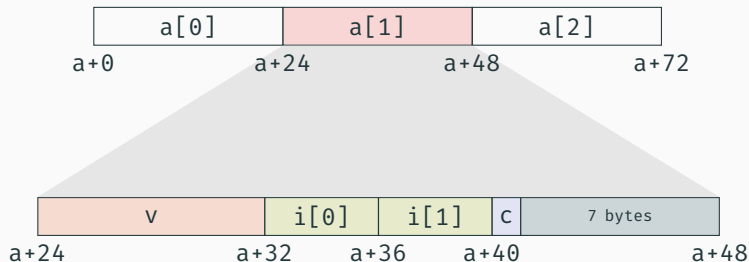
```
struct st_c {  
    double v;  
    int i[2];  
    char c;  
} a[10];
```



## Arreglos de estructuras

- Se deben cumplir los mismos requisitos que para una estructura, para cada elemento del arreglo.
- Con las reglas vistas hasta ahora, el item anterior se cumple siempre.

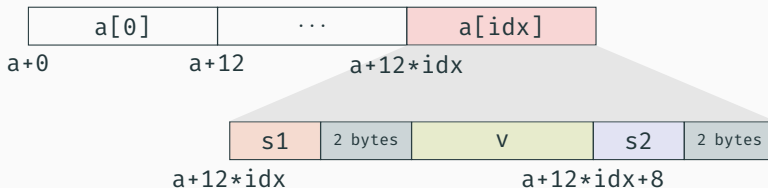
```
struct st_c {  
    double v;  
    int i[2];  
    char c;  
} a[10];
```



## Acceso a los miembros dentro del arreglo

- Calcular la posición del elemento en el arreglo `a`:  
 $12 * \text{idx}$ 
  - 12 es el `sizeof` de `struct st_d`
- `s2` está a un *offset* 8 en la estructura

```
struct st_d {
    short s1;
    float f;
    short s2;
} a[10];
```



```
short get_s2(unsigned long idx) {
    return a[idx].s2; // global
}
```

```
get_s2:
    lea    rax, [rdi + 2*rdi]
    movzx  eax, word [a + 8 + 4*rax]
    ret
```

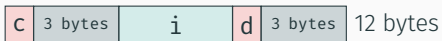


## Asignación óptima

- Simplemente hay que ordenar los datos en la estructura en orden decreciente de tamaño (`sizeof`)

### No óptima

```
struct st_u {  
    char c;  
    int i;  
    char d;  
} *p;
```



### Óptima

```
struct st_o {  
    int i;  
    char c;  
    char d;  
} *p;
```



# Tabla de contenidos

---

## 1. Arreglos

Unidimensionales

Multidimensionales

Multinivel

Tipos de matrices

## 2. Estructuras

Asignación de memoria

Accesos

Alineamiento

## 3. Punto Flotante

# Historia

- Punto Flotante (FP) en x87
  - Arcáico (1977 hasta MMX)
- MMX (Intel P5, 1997)
  - Primeras SIMD, pero **no** soporta FP...
  - ...de hecho, puede corromper el stack de FP
  - Reutiliza parte del x87
- 3DNow! (AMD K6-2, 1998)
  - Extiende MMX
  - Agrega la capacidad de trabajar con FP (32 bits)
- Serie SSE (Streaming SIMD Extensions)
  - Agrega soporte de FP a MMX
  - Agrega los registros `xmm $n$`  ( $n = 0, \dots, 7$ ) de 128 bits
- Serie AVX (Advanced Vector Extensions)
  - Extiende los registros `xmm` a 256 y 512 bits (`ymm` y `zmm`)
  - Es el más avanzado a la fecha

## Programación con SSE4: registros XMM

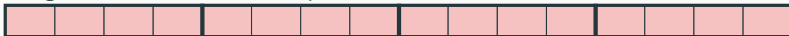
16 registros de enteros de 1 byte



8 registros de enteros de 2 bytes



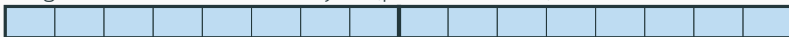
4 registros de enteros de 4 bytes



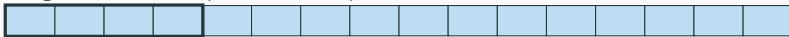
4 registros de **floats** de 4 bytes (precisión simple)



2 registros de **doubles** de 8 bytes (precisión doble)



1 registro de FP en precisión simple



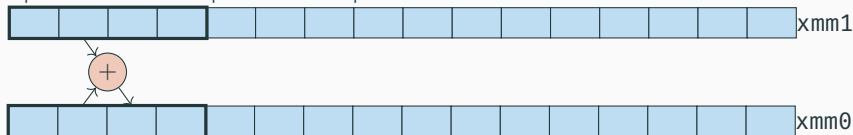
1 registro de FP en precisión doble



## Operaciones escalares y SIMD

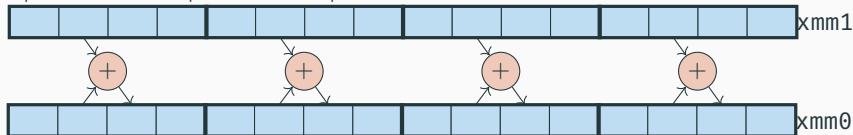
Operación escalar: precisión simple

`addss xmm1,xmm0`



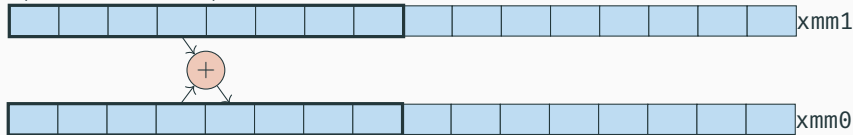
Operación SIMD: precisión simple

`addps xmm0,xmm1`



Operación escalar: precisión doble

`addsd xmm0,xmm1`



## Calling conventions de punto flotante

- Los argumentos se pasan en `xmm0`, `xmm1`, ...
- El resultado se retorna en `xmm0`
- Todos los registros XMM son **caller-saved**

```
float fadd(float x, float y){  
    return x + y;  
}
```

```
fadd:  
    addss    xmm0, xmm1  
    ret
```

```
double dadd(double x, double y){  
    return x + y;  
}
```

```
dadd:  
    addsd    xmm0, xmm1  
    ret
```

## Referenciando memoria con punto flotante

- Los argumentos enteros (y punteros) se pasan en registros comunes (**rdi**, **rsi**, ...)
- Los argumentos de punto flotante se pasan en los registros XMM
- Hay instrucciones separadas para mover datos entre registros XMM, y entre la memoria y los registros XMM

```
double dincr(double *p, double v){  
    double x = *p;  
  
    *p = x + v;  
    return x;  
}
```

```
dincr:  
    movapd    xmm1, xmm0          ; Guarda v en xmm1  
    movsd     xmm0, QWORD [rdi]   ; Desreferencia rdi a xmm0  
    addsd     xmm1, xmm0          ; Suma y guarda en xmm1  
    movsd     QWORD [rdi], xmm1   ; Guarda xmm1 en *p  
    ret                               ; retorna xmm0
```

## Licencia del estilo de beamer

Obtén el código de este estilo y la presentación demo en

`github.com/pamoreno/mtheme`

El estilo *en sí* está licenciado bajo la Creative Commons Attribution-ShareAlike 4.0 International License. El estilo es una modificación del creado por Matthias Vogelgesang, disponible en

`github.com/matze/mtheme`

