

Programación a Nivel-Máquina IV: Datos

Estructura de Computadores
Semana 6

Bibliografía:

[BRY16] Cap.3 Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016
Signatura ESIT/C.1 BRY com

Transparencias del libro CS:APP, Cap.3

Introduction to Computer Systems: a Programmer's Perspective

Autores: Randal E. Bryant y David R. O'Hallaron

<http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html>

Guía de trabajo autónomo (4h/s)

■ **Lectura:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Array Allocation and Access
 - § 3.8 pp.291-301
- Heterogeneous Data Structures
 - § 3.9 pp.301-312

■ **Ejercicios:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Probl. 3.36 – 3.40 § 3.8, pp.292,294,295,298₂
- Probl. 3.41 – 3.45 § 3.9, pp.304,305,308,311₂

Bibliografía:

[BRY16] Cap.3

Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016

Signatura ESIIT/C.1 BRY com

Programación Máquina IV: Datos

■ Arrays[†]

- Uni-dimensionales
- Multi-dimensionales (anidados)
- Multi-nivel

■ Estructuras

- Ubicación
- Acceso
- Alineamiento

■ Uniones

† Hay autores españoles que distinguen entre “vectores” (1D) y “matrices” (2D) 3

Tipos de Datos Básicos

■ Enteros

- Almacenados y manipulados en registros (enteros) propósito general
- Con/sin signo depende de las instrucciones usadas[†]

Intel	ASM [‡]	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	d	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

■ Punto Flotante

- Almacenados y manipulados en registros punto flotante

Intel	ASM	Bytes	C
Single	s	4	float
Double	d	8	double
Extended	t	10/12/16	long double

*‡ sufijos en sintaxis AT&T Linux
[†] y del tipo datos indicado en C,
de los flags ó "condition codes"
consultados en código ASM. 4*

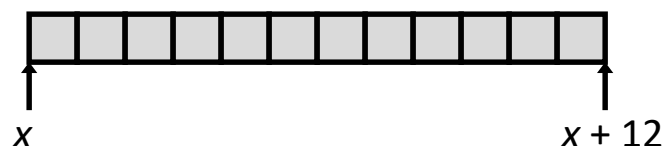
Ubicación[†] de Arrays

■ Principio Básico

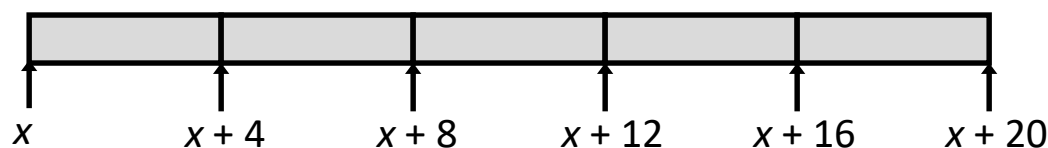
T $A[L];$

- Array de tipo T y longitud L
- Reservada[†] región contigua en memoria de $L * \text{sizeof}(T)$ bytes

`char string[12];`



`int val[5];`



`double a[3];`



`char *p[3];`



[†]“(de)allocate” = reservar/liberar

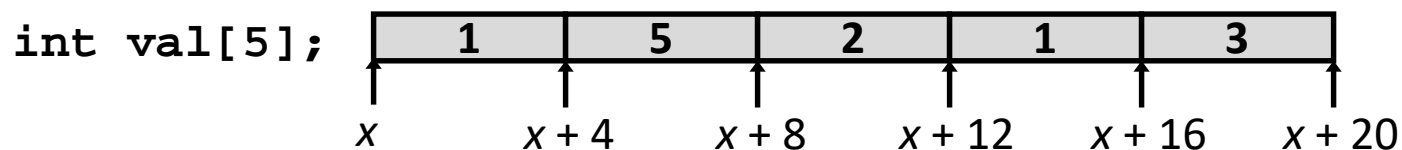
+ “allocation” = ubicación 5

Acceso a Arrays

■ Principio Básico

T $A[L]$;

- Array de tipo T y longitud L
- El identificador A (Tipo T^*) puede usarse como puntero al elemento 0



■ Referencia[†] Tipo Valor

<code>val[4]</code>	<code>int</code>
<code>val</code>	<code>int *</code>
<code>val+1</code>	<code>int *</code>
<code>&val[2]</code>	<code>int *</code>
<code>val[5]</code>	<code>int</code>
<code>*(val+1)</code>	<code>int</code>
<code>val + i</code>	<code>int *</code>

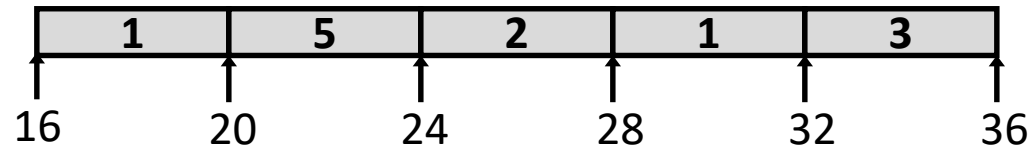
[†] otros autores usan “(de)reference” en sentido mucho más estricto, para indicar el tipo puntero, o la operación de seguir el puntero. 6

Ejemplo de Arrays

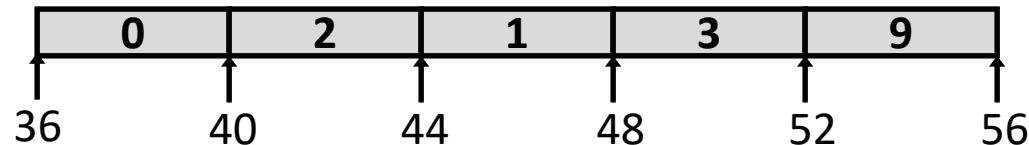
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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

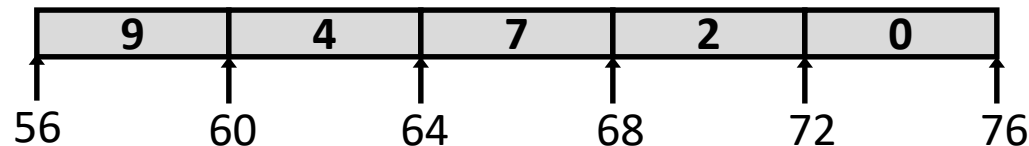
zip_dig cmu;



zip_dig mit;



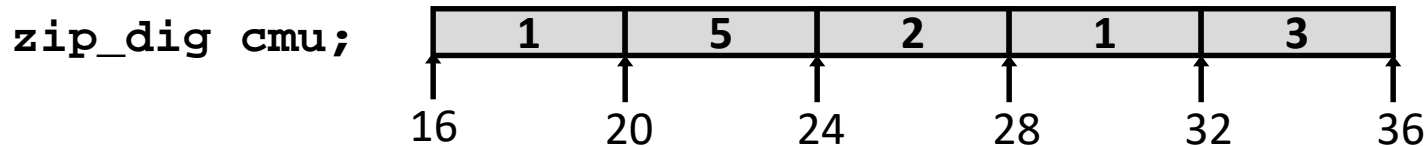
zip_dig ucb;



- Declaración “`zip_dig cmu`” equivalente a “`int cmu[5]`”
- Los arrays del ejemplo fueron ubicados en bloques sucesivos de 20 bytes
 - En general no está garantizado que suceda

+ ZIP = “Zone Improvement Plan”
especie de código postal en USA 7

Ejemplo de Acceso a Arrays



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

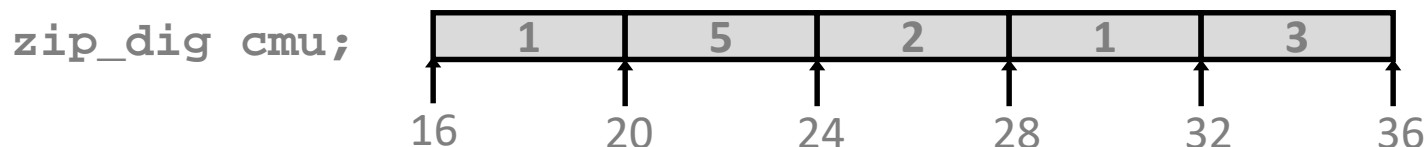
```
get_digit:                # z en %rdi,
†                          # digit %rsi
    movl (%rdi,%rsi,4), %eax # z[digit]
    ret
```

- El registro `%rdi` contiene dirección inicio del array
- El registro `%rsi` contiene el índice al array
- El dígito deseado está en `%rdi + %rsi*4`
- Usar referencia a memoria[†] (`%rdi,%rsi,4`)

[†] `size_t` es usualmente `unsigned long int` en `x86_64`

[‡] "memory reference" en sentido manual

Ejemplo de Acceso a Arrays



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

```
get_digit:                # z en %rdi,
† movslq %esi, %rsi      # digit=%rsi
    movl (%rdi,%rsi,4), %eax # z[digit]
    ret
```

- El registro `%rdi` contiene dirección inicio del array
- El registro `%rsi` contiene el índice al array
- El dígito deseado está en $\%rdi + \%rsi * 4$
- Usar referencia a memoria[‡] (`%rdi,%rsi,4`)

[†] “Move with Sign-extend Long to Quad”, mnemotécnico `MOVSX` según Intel

[‡] “memory reference” en sentido manual

Ejemplo de Bucle sobre Array

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

```
# %rdi = z  
movl    $0, %eax  
jmp     .L3  
.L4:  
addl    $1, (%rdi,%rax,4)  
addq    $1, %rax  
.L3:  
cmpq    $4, %rax  
jbe     .L4  
rep; ret
```

Comprendiendo Arrays y Punteros #1

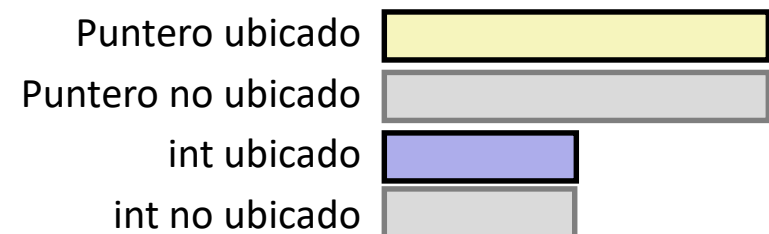
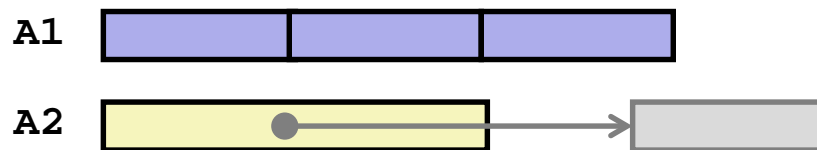
Decl	A1 , A2			*A1 , *A2		
	Comp	Ptr	Size	Comp	Ptr	Size
<code>int A1[3]</code>						
<code>int *A2</code>						

- **Comp: Compila (S/N)**
- **Ptr: Posible error referencia puntero (S/N)**
- **Size: Valor devuelto por `sizeof`**

Comprendiendo Arrays y Punteros #1

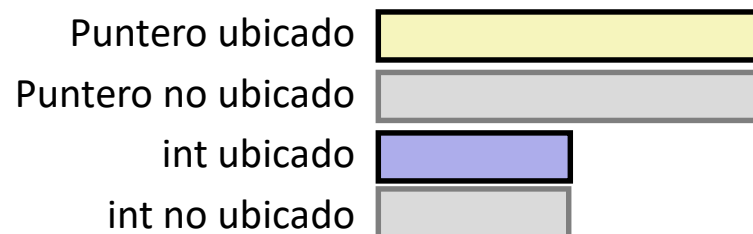
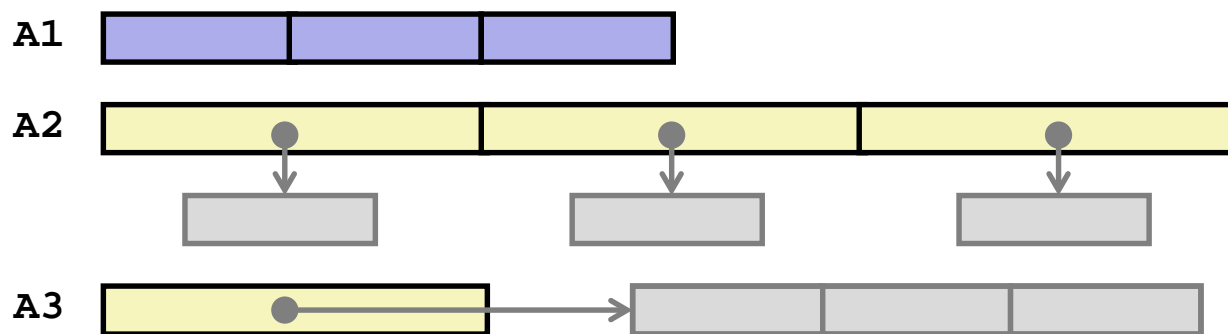
Decl	A1 , A2			*A1 , *A2		
	Comp	Ptr	Size	Comp	Ptr	Size
<code>int A1[3]</code>	.	.	12	.	.	4
<code>int *A2</code>	.	.	8	.	S	4

- **Comp: Compila (S·/N)**
- **Ptr: Posible error referencia puntero (S/N·)**
- **Size: Valor devuelto por sizeof**



Comprendiendo Arrays y Puntereros #2

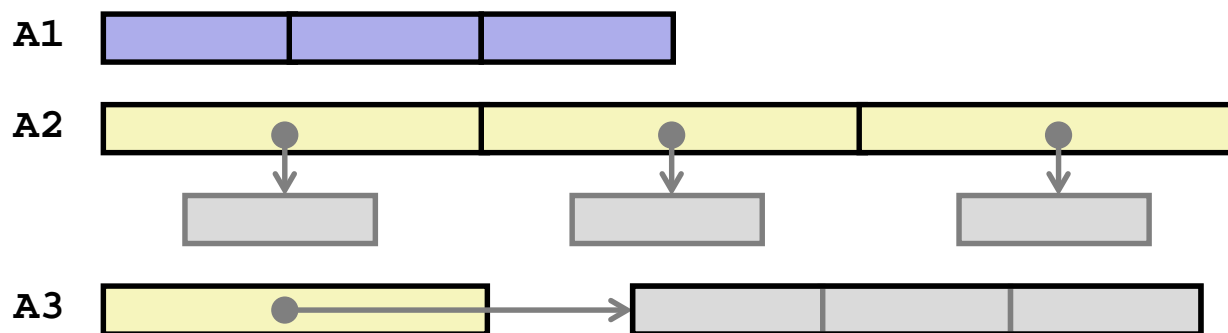
Decl	An			*An			**An		
	Cmp	Ptr	Size	Cmp	Ptr	Size	Cmp	Ptr	Size
<code>int A1[3]</code>									
<code>int *A2[3]</code>									
<code>int (*A3)[3]</code>					.†	12			



† gcc traduce `void*v=A3;`
 igual que `int*p=*A3;`
 como `movq A3(%rip), %rax,`
`movq %rax, p/v(%rip).`

Comprendiendo Arrays y Puntereros #2

Decl	An			*An			**An		
	Cmp	Ptr	Size	Cmp	Ptr	Size	Cmp	Ptr	Size
<code>int A1[3]</code>	.	.	12	.	.	4	N	-	-
<code>int *A2[3]</code>	.	.	24	.	.	8	.	S	4
<code>int (*A3)[3]</code>	.	.	8	.	.†	12	.	S	4



† gcc traduce `void*v=A3;`
 igual que `int*p=*A3;`
 como `movq A3(%rip), %rax,`
`movq %rax, p/v(%rip).`

Arrays Multidimensionales (Anidados)

■ Declaración

$T \ A[R][C];$

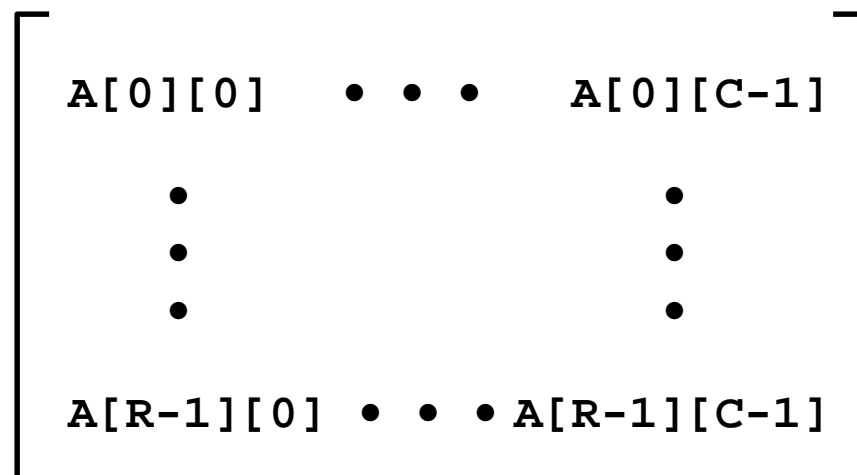
- Array 2D de (elems. de) tipo T
- R filas (rows), C columnas
- Elems. tipo T requieren K bytes

■ Tamaño Array

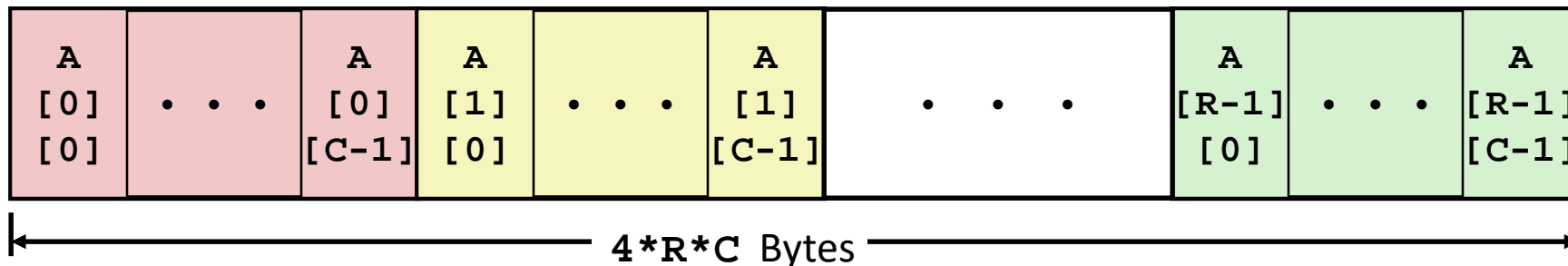
- $R * C * K$ bytes

■ Disposición

- Almacenamiento por filas (row-major-order)[†]

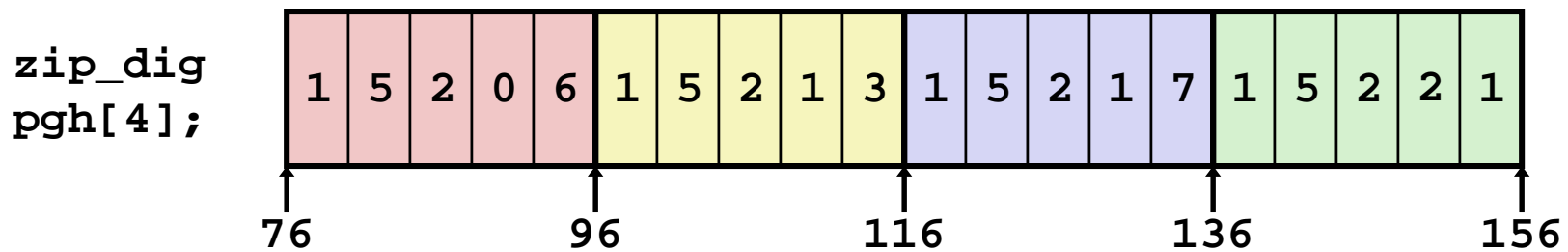


`int A[R][C];`



Ejemplo de Array Anidado

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



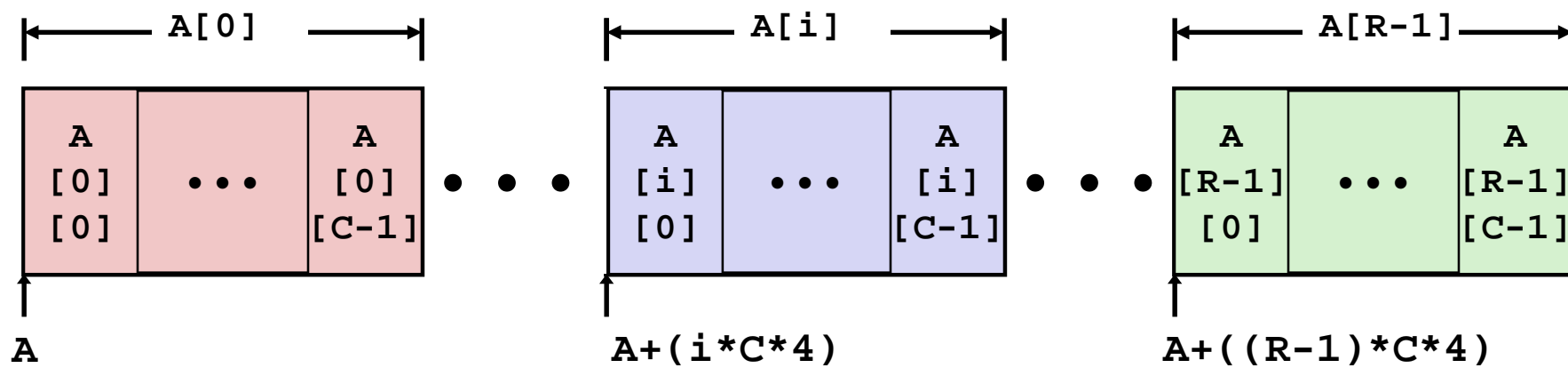
- “zip_dig pgh[4]” equivalente a “int pgh[4][5]”
 - Variable `pgh`: array de 4 elementos, ubicados contiguamente
 - Cada elemento es un array de 5 `int`'s, ubicados contiguamente
- **Garantizado almacenamiento por filas (“row-major order”)**

Acceso a Filas en Arrays Anidados

■ Vectores Fila

- $A[i]$ es un array de C elementos
- Cada elemento de tipo T requiere K bytes
- Dirección de comienzo $A + i * (C * K)$

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



Código Acceso Filas Arrays Anidados

```
int *get_pgh_zip
(size_t index)
{
    return pgh[index];
}
```

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```

```
get_pgh_zip:
                                # index en %rdi
    leaq    (%rdi,%rdi,4), %rax    # 5 * index
    † leaq   pgh(,%rax,4), %rax    # pgh + (20 * index)
    ret
```

■ Vector Fila

- `pgh[index]` es array de 5 `int`'s, comienza en `pgh+20*index`

■ Código x86-64: **bastan 2 lea (compilando con gcc -fno-pie)**

- Calcula `pgh + 4*(index+4*index)`

Código Acceso Filas Arrays Anidados

```
int *get_pgh_zip
(int index)
{
    return pgh[index];
}
```

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```

```
get_pgh_zip:
    movslq  %edi, %rdi           # index = %edi
    leaq   (%rdi,%rdi,4), %rdx   # 5 * index
    † leaq  pgh(%rip), %rax      # pgh
    leaq   (%rax,%rdx,4), %rax   # pgh + (20 * index)
    ret
```

■ Si se usa índice int

- Se añade movslq adicional

■ Si se usa `-fpie` (activado por defecto)

- Se añade lea adicional (`pgh(%rip)`, dir. relativo PC)

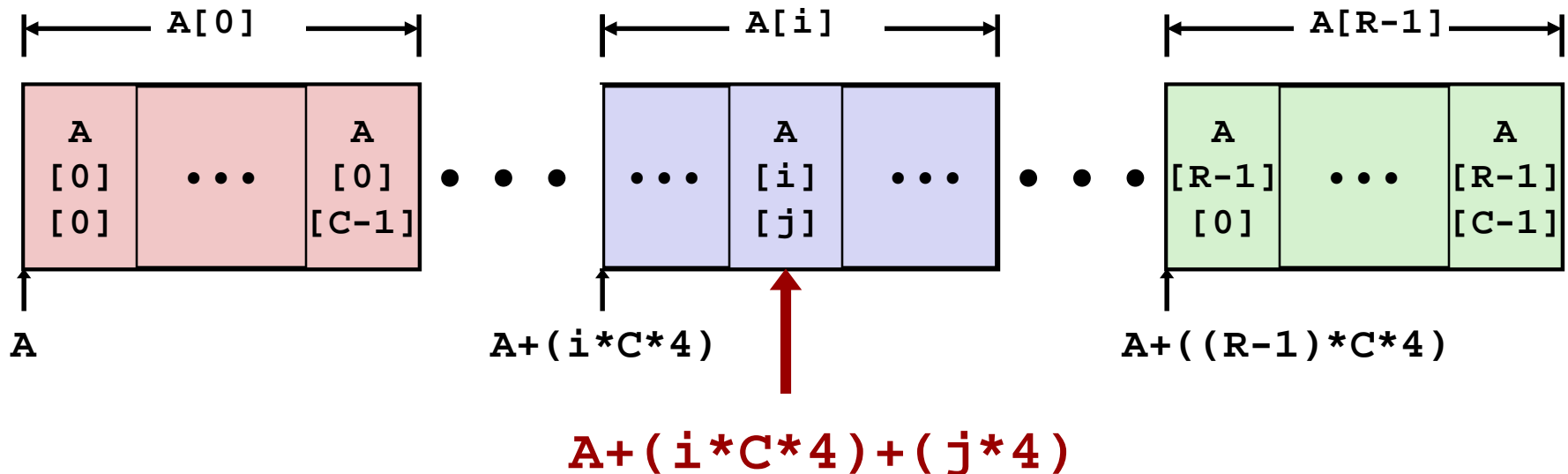
*† es direccionamiento relativo a Contador de Programa %rip, existe en x86-64, no en x86.
Activado por defecto desde Ubuntu 17.10
#Position-independent_executables*

Acceso a Elementos en Arrays Anidados

■ Elementos del Array

- $A[i][j]$ es elemento de tipo T , que requiere K bytes
- Dirección $A + i * (C * K) + j * K = A + (i * C + j) * K$

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



$$A + (i * C * 4) + (j * 4)$$

Código Acceso Elementos Arrays Anidados

```
int get_pgh_digit
(size_t index, size_t digit)
{
    return pgh[index][digit];
}
```

```
get_pgh_digit:
                                # digit en %rax
                                # index en %rdi
leaq  (%rdi,%rdi,4), %rax      # 5*index
addq  %rax, %rsi               # 5*index+digit

†movl  pgh(,%rsi,4), %eax      # pgh + 4 * (5*index+digit)
ret
```

■ Elementos del Array

- `pgh[index][dig]` es `int`
- Dirección: $pgh + 20 * idx + 4 * dig = pgh + 4 * (5 * idx + dig)$
- Código x86_64 calcula la dirección $pgh + 4 * ((idx + 4 * idx) + dig)$

Código Acceso Elementos Arrays Anidados

```
int get_pgh_digit
(int index, int digit)
{
    return pgh[index][digit];
}
```

```
get_pgh_digit:
    movslq %esi, %rax          # digit = %rax
    movslq %edi, %rdi         # index = %rdi
    leaq  (%rdi,%rdi,4), %rsi  # 5*index
    addq  %rax, %rsi           # 5*index+digit
    leaq  pgh(%rip), %rax      # pgh
    movl  (%rax,%rsi,4), %eax  # pgh + 4 * (5*index+digit)
    ret
```

■ Elementos del Array

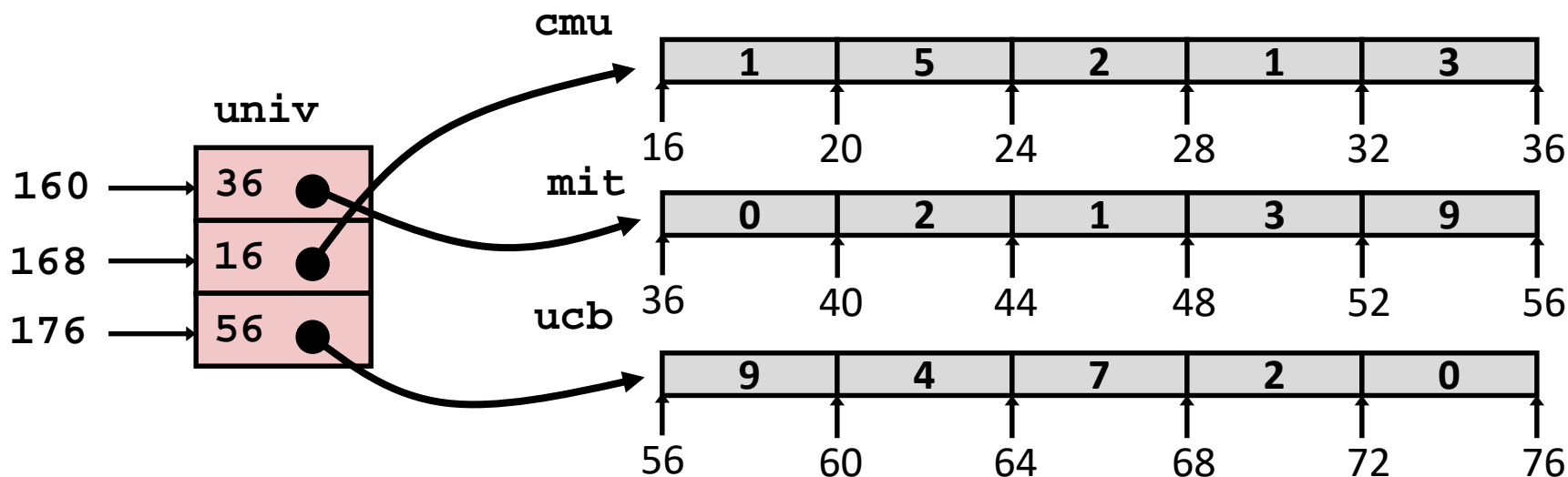
- `pgh[index][dig]` es `int`
- Dirección: $pgh + 20 * idx + 4 * dig = pgh + 4 * (5 * idx + dig)$
- Código x86_64 calcula la dirección $pgh + 4 * ((idx + 4 * idx) + dig)$

Ejemplo de Array Multi-Nivel†

```
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};
```

- Variable `univ` denota un array de 3 elementos
- Cada elemento un puntero
 - 8 bytes
- Cada puntero apunta a un array de `int`'s



† no se suele decir "multinivel",
es más usual "array de punteros". 23

Acceso a Elementos en Array Multi-Nivel

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

get_univ_digit:

```
movq    univ(,%rdi,8), %rax    # p = *(univ+8*index)
movl    (%rax,%rsi,4), %eax    # return *(p+4*digit)
ret
```

■ Cuentas

- Acceso a elemento `Mem[Mem[univ+8*index]+4*digit]`
- Debe hacer dos lecturas de memoria
 - Primero obtener puntero al array[†] fila
 - Entonces acceder elemento dentro del array[†]

[†] se suele decir “vector” (1D)
como opuesto a “matriz” (2D) 24

Acceso a Elementos en Array Multi-Nivel

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

```
get_univ_digit:
† salq    $2, %rsi          # 4*digit
  addq    univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
  movl    (%rsi), %eax      # return *p
  ret
```

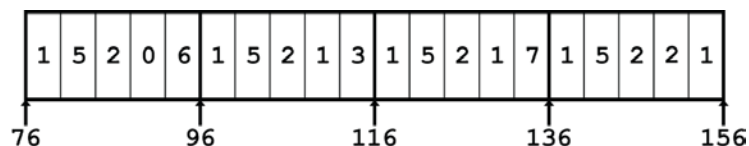
■ Cuentas

- Acceso a elemento `Mem[Mem[univ+8*index]+4*digit]`
- Debe hacer dos lecturas de memoria
 - Primero obtener puntero al array[†] fila
 - Entonces acceder elemento dentro del array[†]

Acceso a Elementos en Arrays

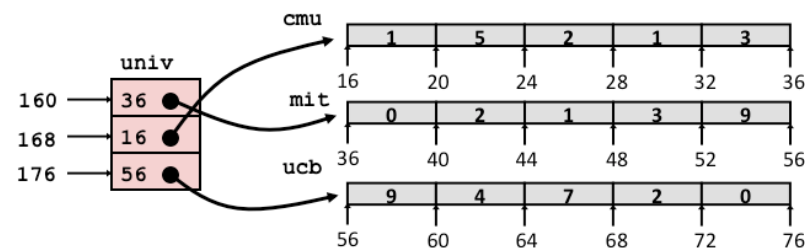
Array anidado

```
int get_pgh_digit
(size_t index, size_t digit)
{
    return pgh[index][digit];
}
```



Array Multi-nivel

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



Accesos parecen similares en C, pero cuentas muy diferentes:

$\text{Mem}[\text{pgh} + 20 * \text{index} + 4 * \text{digit}]$

$\text{Mem}[\text{Mem}[\text{univ} + 8 * \text{index}] + 4 * \text{digit}]$

Código para Matriz N X N

■ Dimensiones fijas

- Se conoce valor de N en tiempo de compilación

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element A[i][j] */
int fix_ele(fix_matrix A,
           size_t i, size_t j)
{
    return A[i][j];
}
```

■ Dimensiones variables, indexado explícito

- Forma tradicional de implementar arrays dinámicos

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

■ Dimensiones variables, indexado implícito

- Soportado ahora[†] por gcc

```
/* Get element A[i][j] */
int var_ele(size_t n, int A[n][n],
           size_t i, size_t j) {
    return A[i][j];
}
```

[†] Ver p.ej.: <http://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html>

Acceso a Matriz 16 X 16

■ Elementos del Array

- `int A[16][16];`
- Dirección $A + i * (C * K) + j * K$
- $C = 16, K = 4$

```
/* Get element A[i][j] */
int fix_ele(fix_matrix A, size_t i, size_t j)
{
    return a[i][j];
}
```

```
# A en %rdi, i en %rsi, j en %rdx
salq    $6, %rsi           # 64*i
addq    %rsi, %rdi         # A + 64*i
movl    (%rdi,%rdx,4), %eax # M[A + 64*i + 4*j]
ret
```

Acceso a Matriz n X n

■ Elementos del Array

- `int A[n][n];`
- Dirección `A + i * (C * K) + j * K`
- `C = n, K = 4`
- Hay que realizar multiplicación entera

```
/* Get element A[i][j] */
int var_ele(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
imulq    %rdx, %rdi          # n*i
leaq     (%rsi,%rdi,4), %rax  # A + 4*n*i
movl     (%rax,%rcx,4), %eax  # A + 4*n*i + 4*j
ret
```

Ejemplo: Accesos a un Array

```
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
    zip_dig pgh[PCOUNT] =
        {{1, 5, 2, 0, 6},
         {1, 5, 2, 1, 3 },
         {1, 5, 2, 1, 7 },
         {1, 5, 2, 2, 1 }};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
        pgh[0][0] +
        linear_zip[7] +
        *(linear_zip + 8) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

```
linux> ./array
result: 9
```

Ejemplo: Accesos a un Array

```
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
    zip_dig pgh[PCOUNT] =
        {{1, 5, 2, 0, 6},
         {1, 5, 2, 1, 3 },
         {1, 5, 2, 1, 7 },
         {1, 5, 2, 2, 1 }};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
        pgh[0][0] +
        linear_zip[7] +
        *(linear_zip + 8) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

```
linux> ./array
result: 9
```

Programación Máquina IV: Datos

■ Arrays

- Uni-dimensionales
- Multi-dimensionales (anidados)
- Multi-nivel

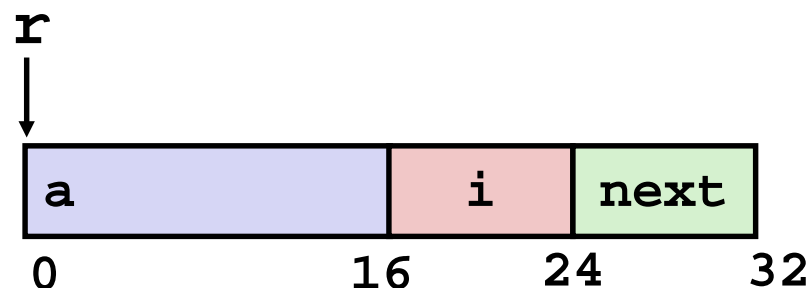
■ Estructuras

- Ubicación
- Acceso
- Alineamiento

■ Uniones

Representación de Estructuras

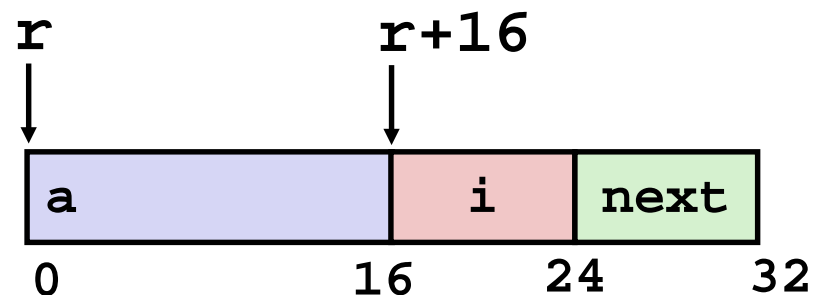
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- **Estructuras representadas como un bloque de memoria**
 - Suficientemente grande como para contener todos los campos
- **Referencia a campos de la estructura mediante sus nombres**
 - Sintaxis struct.field, pointer->field
- **Campos ordenados según la declaración**
 - Incluso si otro orden pudiera producir una representación más compacta
- **Compilador determina posición/tamaño conjunto de los campos**
 - El programa a nivel máquina no entiende las estructuras del código fuente

Acceso a Estructuras

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



■ Accediendo a un Miembro de la Estructura

- Puntero indica primer byte de la estructura[‡]
- Acceder a los elementos mediante sus desplazamientos[‡]

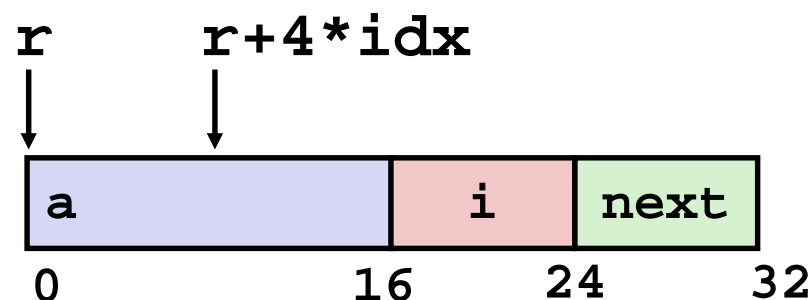
```
void set_i
(struct rec *r,
 size_t val)
{
‡ r->i = val;
}
```

```
set_i:                # r en %rdi,
                    # val en %rsi
movq %rsi, 16(%rdi)  # Mem[r+16] = val
ret
```

[‡] "offset"=compensación, "desplazamiento"
[‡] ptr->fld es abreviación para (*ptr).fld
 Si se declara "struct rec R;," entonces
 "R.a" es array, "R.a[0]" y "R.i" enteros,
 "R.n" puntero, y "R.n->a" otra vez array 34

Generando Puntero a Miembro Estructura

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



■ Generando Puntero a un Elemento del Array

- Desplaz.[†] de cada miembro struct queda determinado en tiempo compilación
- Se calcula $r + 4*idx$

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

```
# r en %rdi, idx en %rsi
leaq (%rdi,%rsi,4), %rax
ret
```

[†] "offset"=compensación, "desplazamiento"

[‡] `ptr->fld` es abreviación para `(*ptr).fld`

Si se declara "struct rec R;", entonces

"R.a" es array, "R.a[0]" y "R.i" enteros,

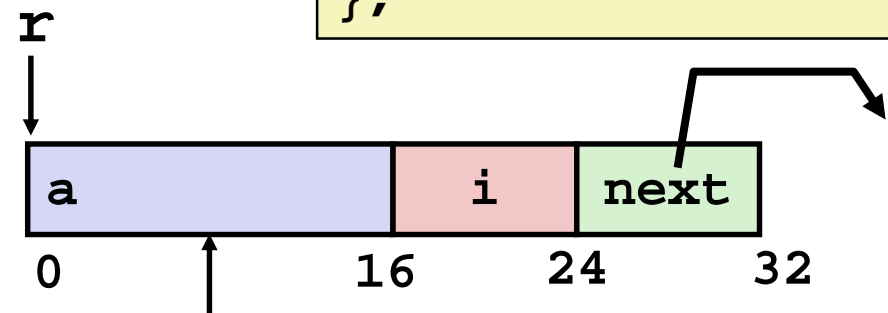
"R.n" puntero, y "R.n->a" otra vez array 35

Siguiendo Lista Encadenada

■ Código C

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

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



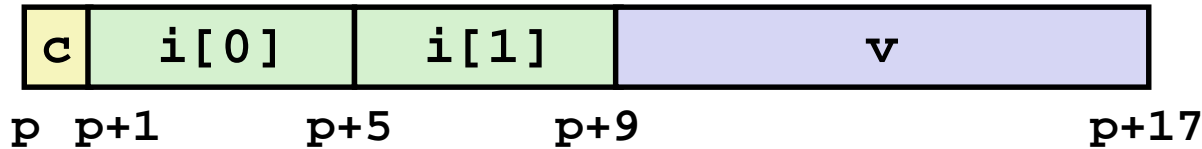
Elemento i

Registro	Valor
<code>%rdi</code>	<code>r</code>
<code>%rsi</code>	<code>val</code>

```
.L11:                                # loop:
    movslq 16(%rdi), %rax              # i = M[r+16]
    movl  %esi, (%rdi,%rax,4)         # M[r+4*i] = val
    movq  24(%rdi), %rdi              # r = M[r+24]
    testq %rdi, %rdi                  # Test r
    jne  .L11                          # if !=0 goto loop
```

Estructuras y Alineamiento

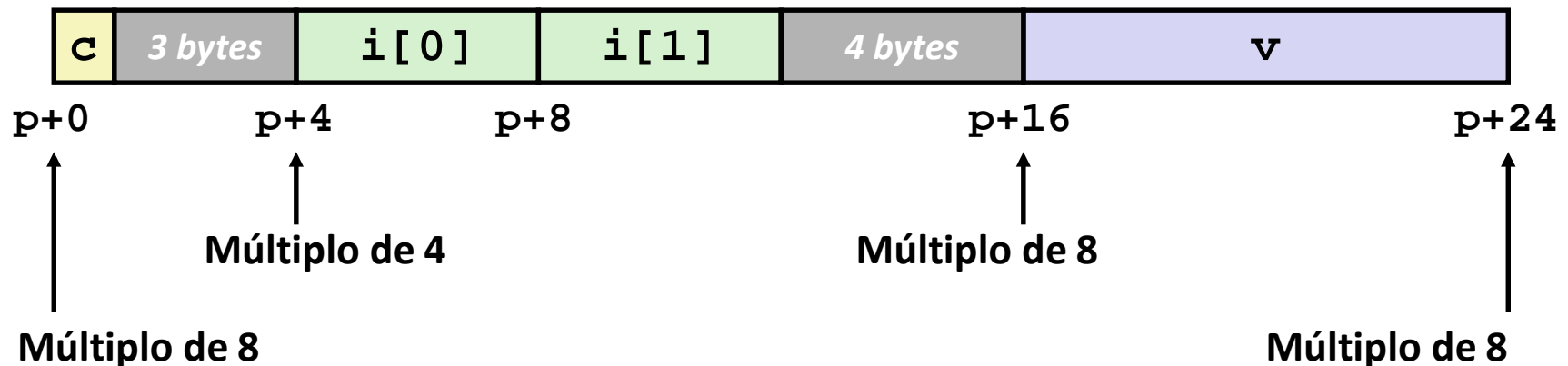
■ Datos Desalineados



```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

■ Datos Alineados

- El tipo de datos primitivo requiere K bytes
- La dirección debe ser múltiplo de K



Principios de Alineamiento

■ Datos Alineados

- El tipo de datos primitivo requiere K bytes
- La dirección debe ser múltiplo de K
- Requisito en algunas máquinas; recomendado en x86-64

■ Motivación para Alinear los Datos

- A la memoria se accede (físicamente) en trozos (alineados) de 4 ú 8 bytes (dependiendo del sistema)
 - Ineficiente cargar o almacenar dato que cruza frontera quad word
 - Mem. virtual muy delicada cuando un dato se extiende a 2 páginas

■ Compilador

- Inserta huecos en estructura para asegurar correcto alineamiento campos

Casos Concretos de Alineamiento

	<i>Linux x86</i>		<i>x86-64</i>		<i>Windows MinGW32</i>		<i>MinGW64</i>	
▪ <i>Tipo de Datos C</i>	<i>tam.alin.</i>		<i>tam.alin.</i>		<i>tam.alin.</i>		<i>tam.alin.</i>	
▪ char	1	1	1	1	1	1	1	1
▪ short	2	2	2	2	2	2	2	2
▪ int	4	4	4	4	4	4	4	4
▪ long	4	4	8	8	4	4	4	4
▪ long long	8	4	8	8	8	8	8	8
▪ float	4	4	4	4	4	4	4	4
▪ double	8	4	8	8	8	8	8	8
▪ long double	12	4	16	16	12	4	16	16
▪ void *	4	4	8	8	4	4	8	8
▪ <i>Regla para memorizar</i>	4x		KB Kx		8x		KB Kx	

ahorro M.

alin.estricto

sin ahorro

sin long 8B

estricto-12B [#Typical alignment of C structs on x86](#)

Casos Concretos de Alineamiento (x86-64)

- **1 byte: char, ...**
 - sin restricciones en la dirección
- **2 bytes: short, ...**
 - el LSB[†] (bit más bajo) de la dirección debe ser 0₂
- **4 bytes: int, float, ...** *(y long en Windows!!!)*
 - los 2 LSB's de la dirección deben ser 00₂
- **8 bytes: double, long, char *, ...**
 - los 3 LSB's de la dirección deben ser 000₂
- **16 bytes: long double (GCC en Linux x86-64)**
 - los 4 LSB's de la dirección deben ser 0000₂

Cumpliendo Alineamiento en Estructuras

■ Dentro de la estructura:

- Deben cumplirse requisitos alinm. de cada elemento

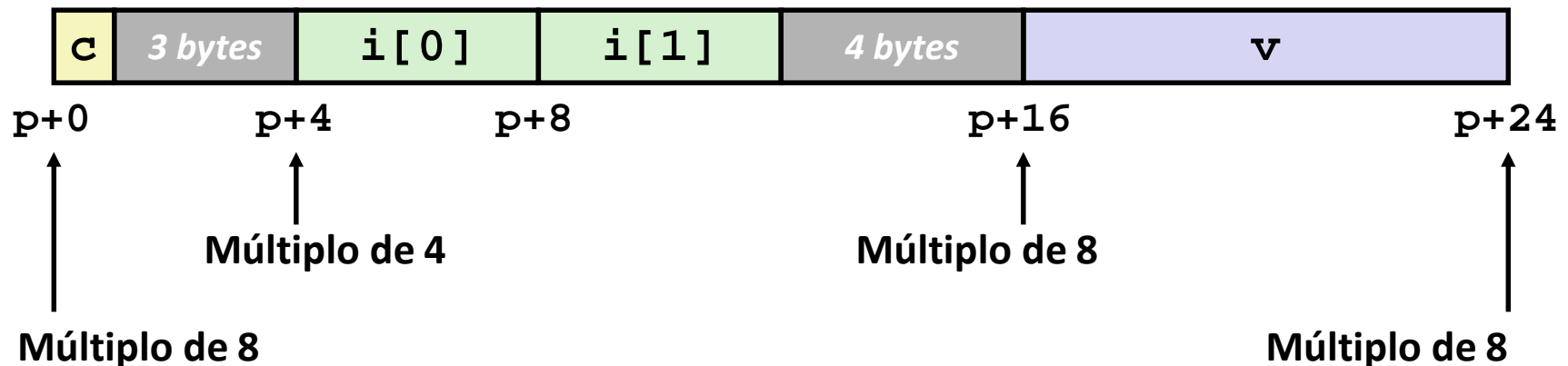
■ Colocación global de la estructura

- Cada estructura tiene un requisito de alineamiento K
 - K = Mayor alineamiento de cualquier elemento
- Dirección inicial y longitud estructura deben ser múltiplos de K

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

■ Ejemplo:

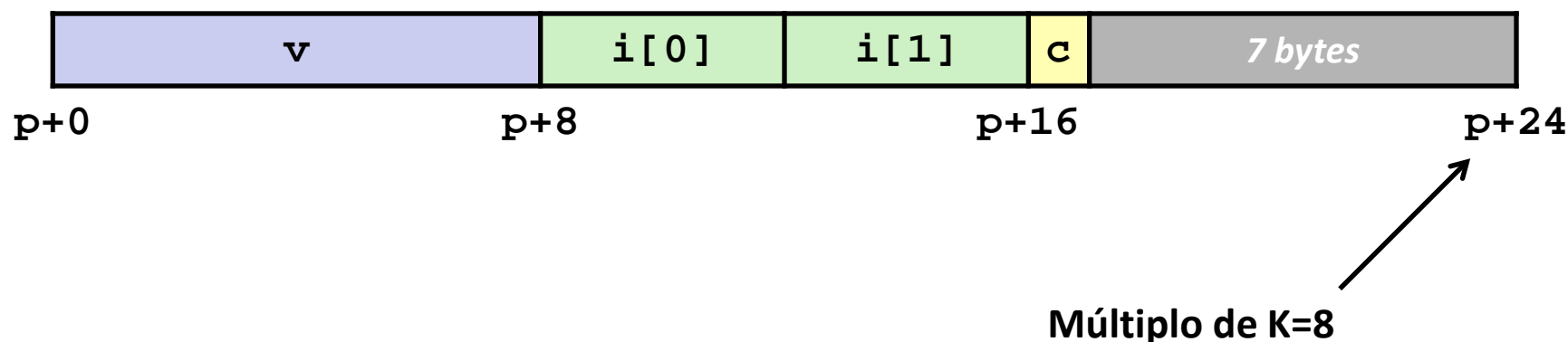
- $K = 8$, debido al elemento `double`



Cumpliendo Requisito Alineamiento Global

- Si el requisito de alineamto. máximo es K
- La struct debe ocupar glob. múltiplo de K

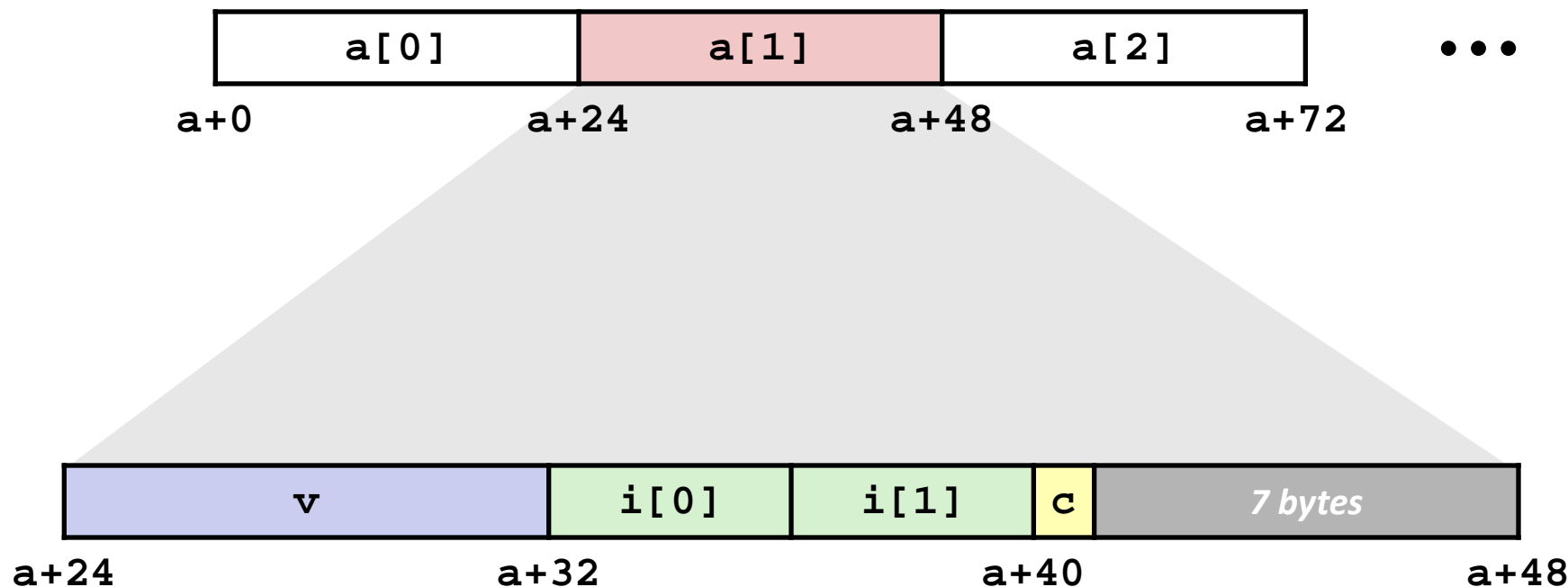
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



Arrays de Estructuras

- Longitud global estructura[†] múltiplo de K
- Cumplir requisitos alnmt. de cada elemento

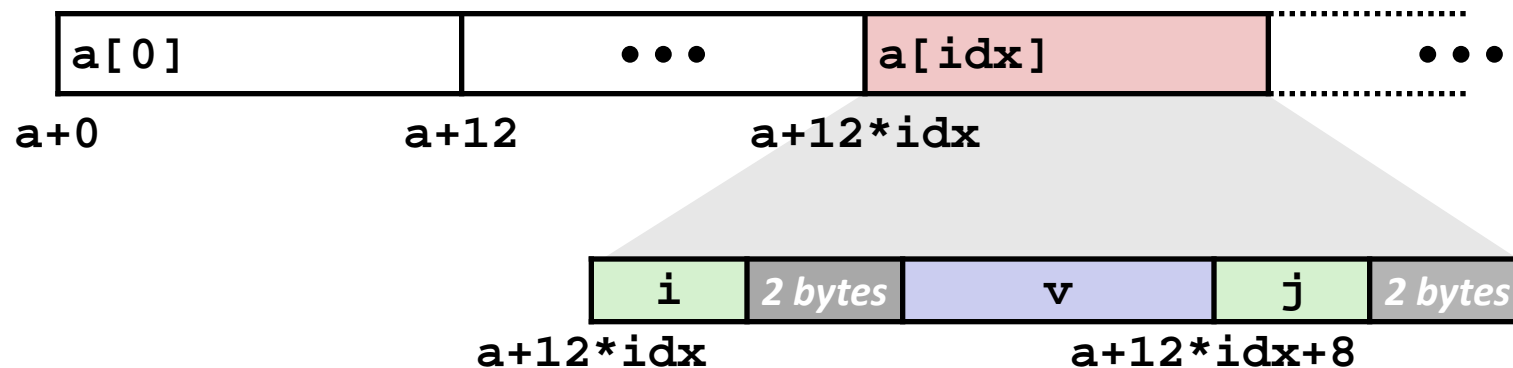
```
struct s2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Acceso a Elementos del Array

- Calcular desplazamiento elem. array: $12i$
 - $\text{sizeof}(S3)*i$, incluyendo espaciadores alineamiento
- Elemento j @ despl. 8 dentro de estructura
- El ensamblador genera desplazamiento $a+8$
 - Resuelto durante enlazado (en tiempo de *link*)

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```



```
short get_j(size_t idx)
{
    return a[idx].j;
}
```

```
# idx en %rdi
leaq (%rdi,%rdi,2),%rax # 3*idx
† movzwl a+8(,%rax,4),%eax
```

† "Move with Zero-extend Word to Long", mnemotécnico MOVZX según Intel

Ahorro de Espacio

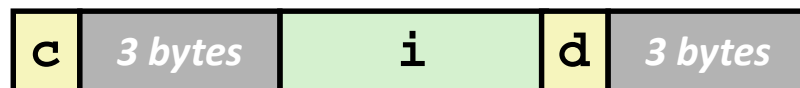
- Poner primero los tipos de datos grandes

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```



```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- Efecto (K=4)



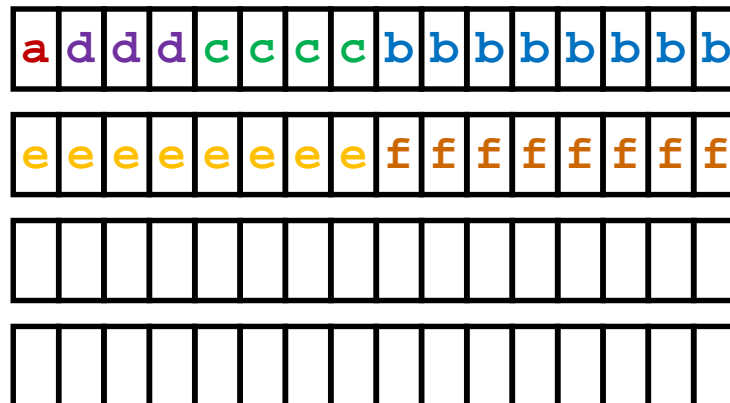
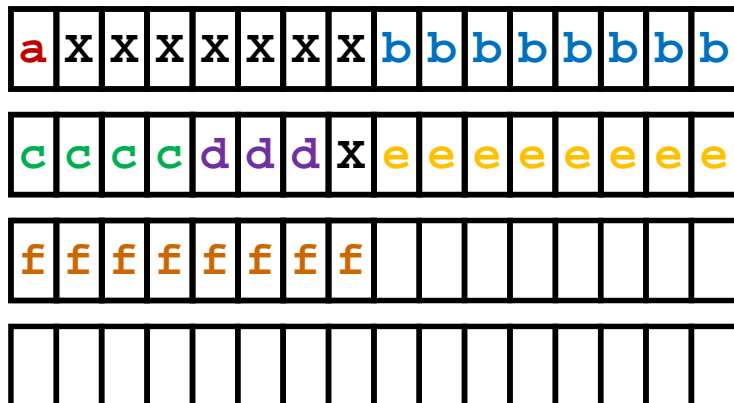
Ejemplo: Ejercicio sobre structs

- Indicar cómo se ubicaría en memoria la siguiente estructura (gcc Linux x86_64), marcando las posiciones ocupadas por cada campo con su nombre, y las de relleno (tanto interno como global) con una X. Repetir el ejercicio reordenando los campos para obtener el máximo ahorro de memoria posible. (ver ~ Ex.Probl.Sep'13)

```
struct foo {
    char a;
    long b;
    float c;
    char d[3];
    int *e;
    short *f;
} mystruct1;
```



```
struct bar {
    char a;
    char d[3];
    float c;
    long b;
    int *e;
    short *f;
} mystruct2;
```



Programación Máquina IV: Datos

■ Arrays

- Uni-dimensionales
- Multi-dimensionales (anidados)
- Multi-nivel

■ Estructuras

- Ubicación
- Acceso
- Alineamiento

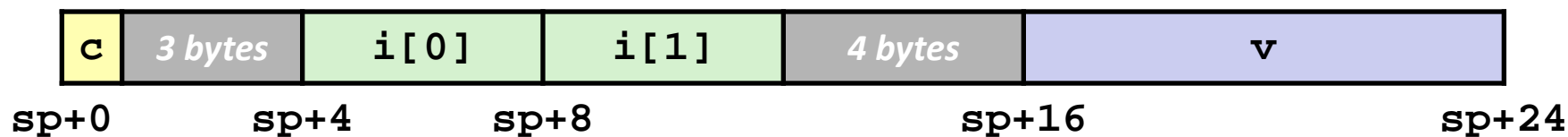
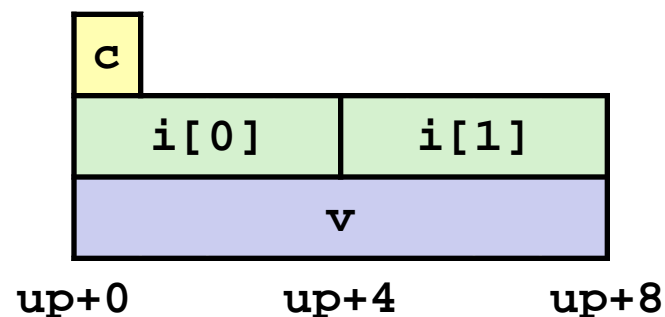
■ Uniones

Ubicación† de Uniones

- Reservar† de acuerdo al elemento más grande
- Sólo puede usarse un campo a la vez

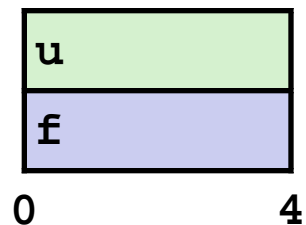
```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Uso de Uniones para Acceder Patrones Bit

```
typedef union {  
    float f;  
    unsigned u;  
} bit_float_t;
```



```
float bit2float(unsigned u)  
{  
    bit_float_t arg;  
    arg.u = u;  
    return arg.f;  
}
```

¿Lo mismo que (float) u ?

```
unsigned float2bit(float f)  
{  
    bit_float_t arg;  
    arg.f = f;  
    return arg.u;  
}
```

¿Lo mismo que (unsigned) f ?

Ordenamiento de Bytes[†]: un Repaso

■ Idea

- Palabras short/long/quad, almacenadas en mem. como 2/4/8B consecutivos
- ¿Cuál es el byte más (menos) significativo?
- Puede causar problemas al intercambiar datos binarios entre máquinas

■ Big Endian[‡] (extremo mayor)

- El byte más significativo está en la dirección más baja (“viene primero”)
- Sparc

■ Little Endian[‡] (extremo menor)

- El byte menos significativo está en la dirección más baja
- Intel x86, ARM con Android e IOS

■ Bi Endian

- Se puede configurar de cualquiera de las dos formas
- ARM

[†] “byte ordering” en inglés, se refiere al orden de bytes en palabras, no a ordenar un array de bytes = “sorting”

[‡] “big/little endian” = “partidario extremo mayor/menor”, ver libro, §2.1.3, Aside: Origin of “endian” 50

Ejemplo de Ordenamiento de Bytes

```
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```

32-bit

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

64-bit

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

Ejemplo de Ordenamiento de Bytes (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

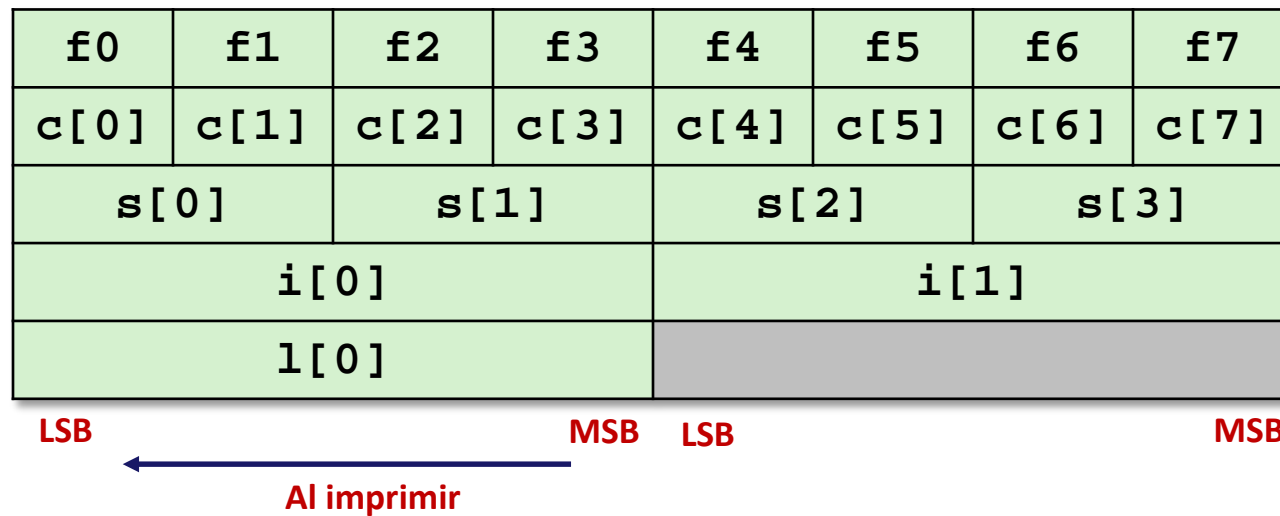
printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Ordenamiento de Bytes en IA32

Little Endian



Salida :

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]

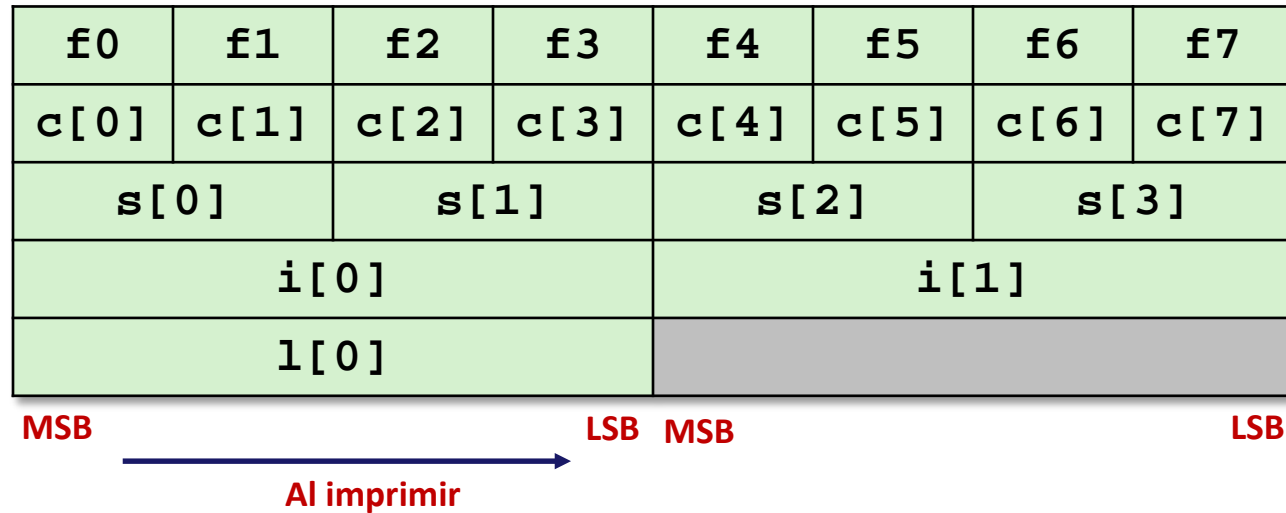
Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]

Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]

Long 0 == [0xf3f2f1f0]

Ordenamiento de Bytes en Sun

Big Endian



Salida en Sun:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]

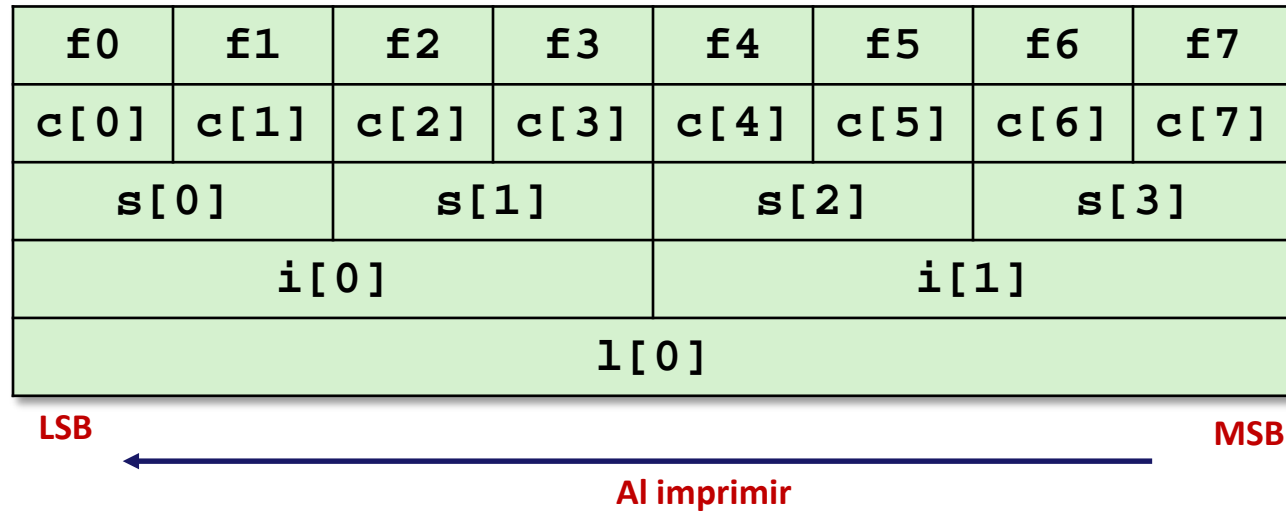
Shorts 0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]

Ints 0-1 == [0xf0f1f2f3,0xf4f5f6f7]

Long 0 == [0xf0f1f2f3]

Ordenamiento de Bytes en x86-64

Little Endian



Salida en x86-64:

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
Ints 0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [0xf7f6f5f4f3f2f1f0]

Resumen de Tipos Compuestos en C

■ Arrays

- Reserva de memoria contigua para almacenar elementos
- Se usa aritmética de indexación para localizar elementos individuales
- Puntero al primer elemento
- Sin chequeo de límites

■ Estructuras

- Reserva de una sola región de memoria, campos van en el orden declarado
- Se accede usando desplazamientos determinados por el compilador
- Puede requerir relleno interno y externo para cumplir con el alineamiento

■ Combinaciones

- Se pueden anidar representación estructura y array arbitrariamente
- Relleno externo estructuras garantiza alineamiento en arrays de structs

■ Uniones

- Declaraciones superpuestas
- Forma de soslayar el sistema de promoción de tipos en C

Guía de trabajo autónomo (4h/s)

■ **Estudio:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Array Allocation and Access
 - § 3.8 pp.291-301
- Heterogeneous Data Structures
 - § 3.9 pp.301-312

■ **Ejercicios:** del Cap.3 CS:APP (Bryant/O'Hallaron)

- Probl. 3.36 – 3.40 § 3.8, pp.292,294,295,298₂
- Probl. 3.41 – 3.45 § 3.9, pp.304,305,308,311₂

Bibliografía:

[BRY16] Cap.3

Computer Systems: A Programmer's Perspective 3rd ed. Bryant, O'Hallaron. Pearson, 2016

Signatura ESIT/C.1 BRY com