

Programación a Nivel-Máquina IV: Procedimientos x86-64, Datos

Estructura de Computadores
Semana 5

Bibliografía:

[BRY11] Cap.3 Computer Systems: A Programmer's Perspective. Bryant, O'Hallaron. Pearson, 2011
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

Guía de trabajo autónomo (4h/s)

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

- x86-64, Procedures, Data Structures, Observations.
 - 3.13.4 - .13.6 pp.316-325
- Array Allocation & Access, Heterogeneous Data Structures (hasta Unions)
 - 3.8 – 3.9.1 pp.266-278

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

- Probl. 3.50 - 3.53 pp.318, 323-25
- Probl. 3.35 – 3.39 pp.267-68, 270, 272, 277

Bibliografía:

[BRY11] Cap.3

Computer Systems: A Programmer's Perspective. Bryant, O'Hallaron. Pearson, 2011

Signatura ESIT/C.1 BRY com

Progr. Máquina IV: Procs. x86_64 / Datos

- **Procedimientos (x86-64)**
- **Matrices***
 - Uni-dimensionales
 - Multi-dimensionales (anidadas)
 - Multi-nivel
- **Estructuras**
 - Ubicación
 - Acceso

* “arreglos” en centro y sudamérica. Hay autores españoles que distinguen entre “vectores” (1D) y “matrices” (2D) 3

x86-64, registros enteros

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

- El doble de registros
- Accesibles como 8, 16, 32, 64 bits

x86-64, registros enteros :

Convenios de uso

%rax	Valor retorno
%rbx	Salva invocado
%rcx	Argumento #4
%rdx	Argumento #3
%rsi	Argumento #2
%rdi	Argumento #1
%rsp	Puntero de pila
%rbp	Salva invocado

%r8	Argumento #5
%r9	Argumento #6
%r10	Salva invocante
%r11	Salva invocante
%r12	Salva invocado
%r13	Salva invocado
%r14	Salva invocado
%r15	Salva invocado

x86-64, registros

- **Argumentos pasados a funciones *vía* registros**
 - Si más de 6 parámetros enteros, pasar resto en pila
 - Estos registros pueden usarse también como salva-invocante

- **Todas las referencias a marco pila *vía* puntero pila**
 - Elimina la necesidad de actualizar `%ebp/%rbp`

- **Otros registros**
 - 6 salva invocado
 - 2 salva invocante
 - 1 valor de retorno (también usable como salva invocante)
 - 1 especial (puntero pila)

x86-64, swap* long int

```
void swap_l(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret
```

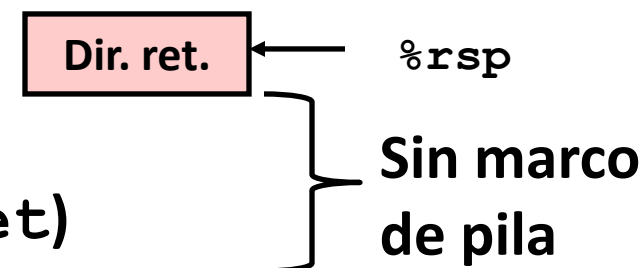
■ Operandos pasados en registros

- Primero (**x**p) en %rdi, 2º (**y**p) en %rsi
- Punteros de 64-bit

■ No requiere operaciones en pila (salvo ret)

■ Evita marco pila

- Puede mantener toda la información local en registros



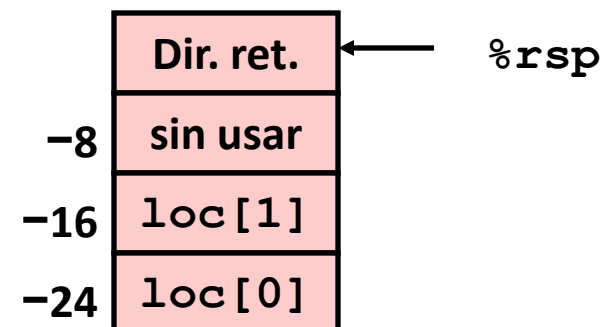
x86-64, Locales en la zona roja*

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
    movq    (%rdi), %rax
    movq    %rax, -24(%rsp)
    movq    (%rsi), %rax
    movq    %rax, -16(%rsp)
    movq    -16(%rsp), %rax
    movq    %rax, (%rdi)
    movq    -24(%rsp), %rax
    movq    %rax, (%rsi)
    ret
```

■ Evita cambios del puntero pila

- Puede almacenar toda la info. dentro de una pequeña ventana más allá del puntero de pila (debajo)



* ver misma Wikipedia pág.5, o “Red Zone”. Zona de 128-bytes bajo %rsp 8

x86-64, Procs. padre* sin marco pila

```
/* Swap a[i] & a[i+1] */  
void swap_ele(long a[], int i)  
{  
    swap(&a[i], &a[i+1]);  
}
```

- No hay valores que mantener mientras swap es invocado
- No necesita regs. salva-invocado
- Instr. rep insertada como no-op
 - Basado en recomendación de AMD††

swap_ele:

```
†   movslq %esi,%rsi           # Sign extend i  
    leaq   8(%rdi,%rsi,8), %rax # &a[i+1]  
    leaq   (%rdi,%rsi,8), %rdi  # &a[i] (1st arg)  
    movq   %rax, %rsi          # (2nd arg)  
    call   swap  
  
††  rep                               # No-op  
    ret
```

† “Move with Sign-extend Long to Quad”, mnemotécnico MOVSLQ según Intel

†† problema predicción saltos Opteron y Athlon 64 (2003-2005) en RET 1B tras flowctrl.

Software Optimization Guide for AMD64 Family 10-12h (2010-2011) recomienda RET 0

Software Optimization Guide for AMD64 Family 15h (2011) no menciona ningún problema 9

x86-64, Ejemplo de marco de pila

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
    (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Mantiene los valores de `&a[i]` y `&a[i+1]` en registros salva-invocado
- Debe ajustar marco de pila para salvar estos registros

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi, %rax
    leaq    8(%rdi, %rax, 8), %rbx
    leaq    (%rdi, %rax, 8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

Comprendiendo marco pila x86-64

swap_ele_su:

movq	%rbx, -16(%rsp)	# Salvar %rbx
movq	%rbp, -8(%rsp)	# Salvar %rbp
subq	\$16, %rsp	# Reservar* marco pila
movslq	%esi, %rax	# Extender i (signo)
leaq	8(%rdi, %rax, 8), %rbx	# &a[i+1] (salva invocado)
leaq	(%rdi, %rax, 8), %rbp	# &a[i] (salva invocado)
movq	%rbx, %rsi	# 2º argumento
movq	%rbp, %rdi	# 1º argumento
call	swap	
movq	(%rbx), %rax	# Traerse a[i+1]
imulq	(%rbp), %rax	# Multiplicar por a[i]
addq	%rax, sum(%rip)	# Sumar a sum **
movq	(%rsp), %rbx	# Restaurar %rbx
movq	8(%rsp), %rbp	# Restaurar %rbp
addq	\$16, %rsp	# Liberar* marco
ret		

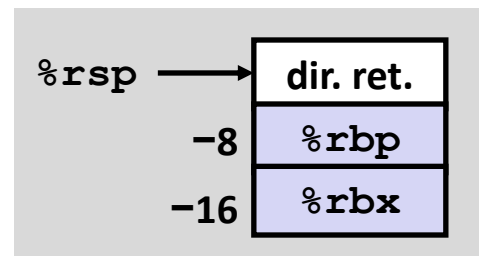
* "allocate/deallocate" en inglés

** direccionamiento relativo a Contador de Programa %rip, existe en x86-64, no en x86 11

Comprendiendo marco pila x86-64

```
movq    %rbx, -16(%rsp)
movq    %rbp, -8(%rsp)
```

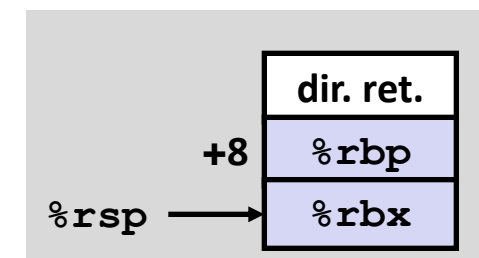
```
# Salva%rbx
# Salva%rbp
```



```
subq    $16, %rsp
```

```
# Reservar marco pila
```

● ● ●



```
movq    (%rsp), %rbx
movq    8(%rsp), %rbp
```

```
# Restaurar %rbx
# Restaurar %rbp
```

```
addq    $16, %rsp
```

```
# Liberar marco
```

Características interesantes del marco pila

■ Reserva el marco entero de una vez

- Todos los accesos a pila pueden hacerse relativos a `%rsp`
- Hacerlo decrementando puntero pila
- Reserva puede posponerse, ya que es seguro usar temporalmente la zona roja

■ Liberación sencilla

- Incrementar puntero pila
- No se necesita puntero marco/base

Resumen Procedimientos (x86-64)

■ Uso intensivo de registros

- Paso de parámetros
- Más temporales ya que hay más registros

■ Uso mínimo de la pila

- A veces ninguno
- Reservar/liberar el bloque entero

■ Muchas optimizaciones complicadas

- Qué tipo de marco de pila usar
- Diversas técnicas de reserva

Progr. Máquina IV: Procs. x86_64 / Datos

- Procedimientos (x86-64)
- **Matrices**
 - Uni-dimensionales
 - Multi-dimensionales (anidadas)
 - Multi-nivel
- Estructuras

Tipos de datos básicos

■ Enteros

- Almacenados & 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	l	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

■ Coma flotante

- Almacenados & manipulados en registros coma flotante

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

** sintaxis ASM AT&T Linux

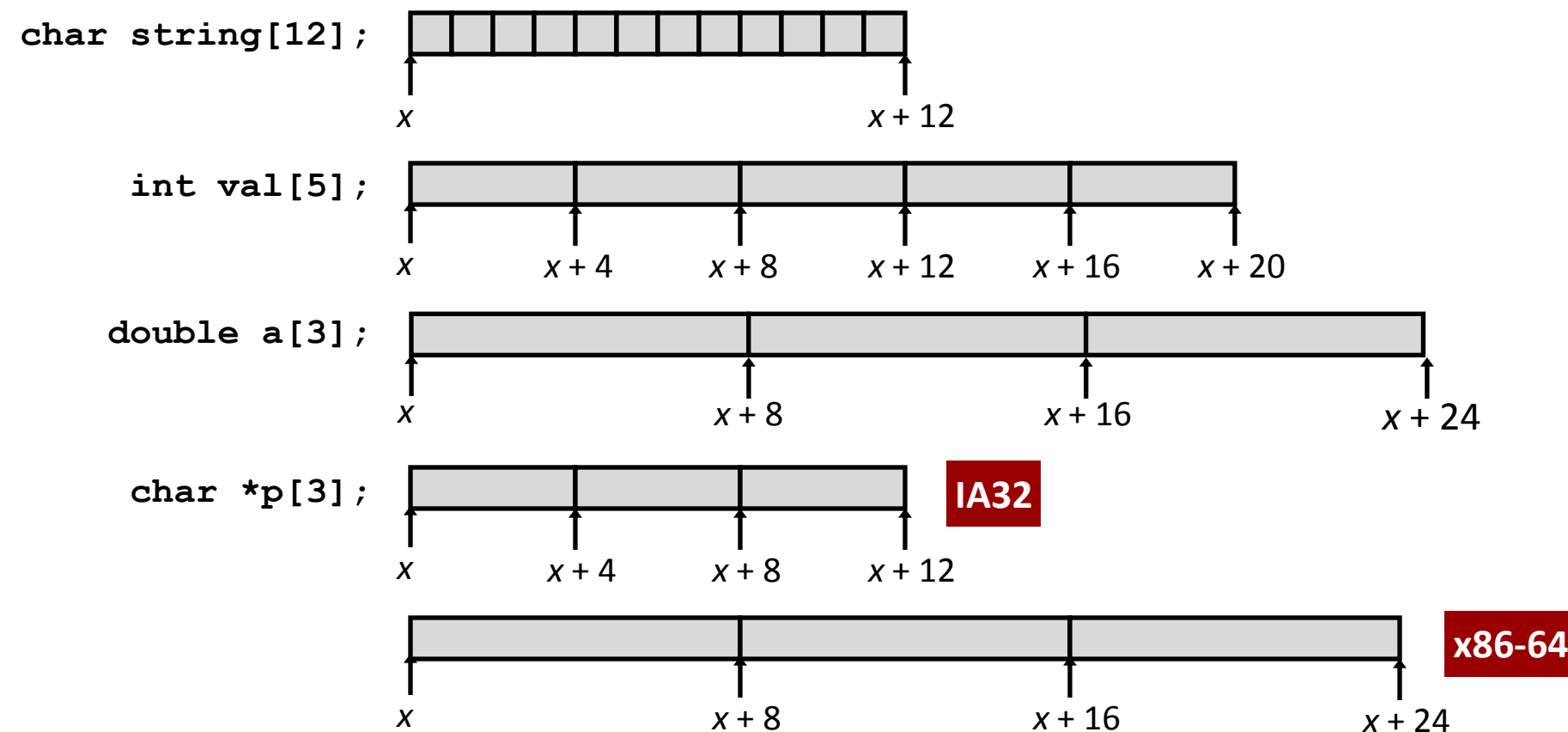
* y de los flags ó “condition codes” consultados en ASM 16

Ubicación* de matrices

■ Principio básico

$T \ A[L];$

- Matriz de tipo T y longitud L
- Se reserva* región contigua de $L * \text{sizeof}(T)$ bytes



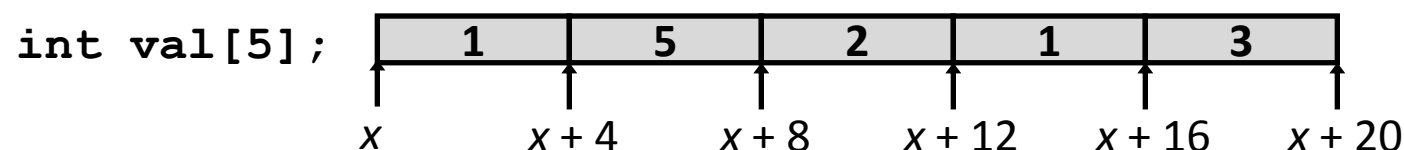
* "allocat(ion)/(ed)" en inglés

Acceso a matrices

■ Principio básico

$T \ A[L] ;$

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



■ Referencia*

Tipo

Valor

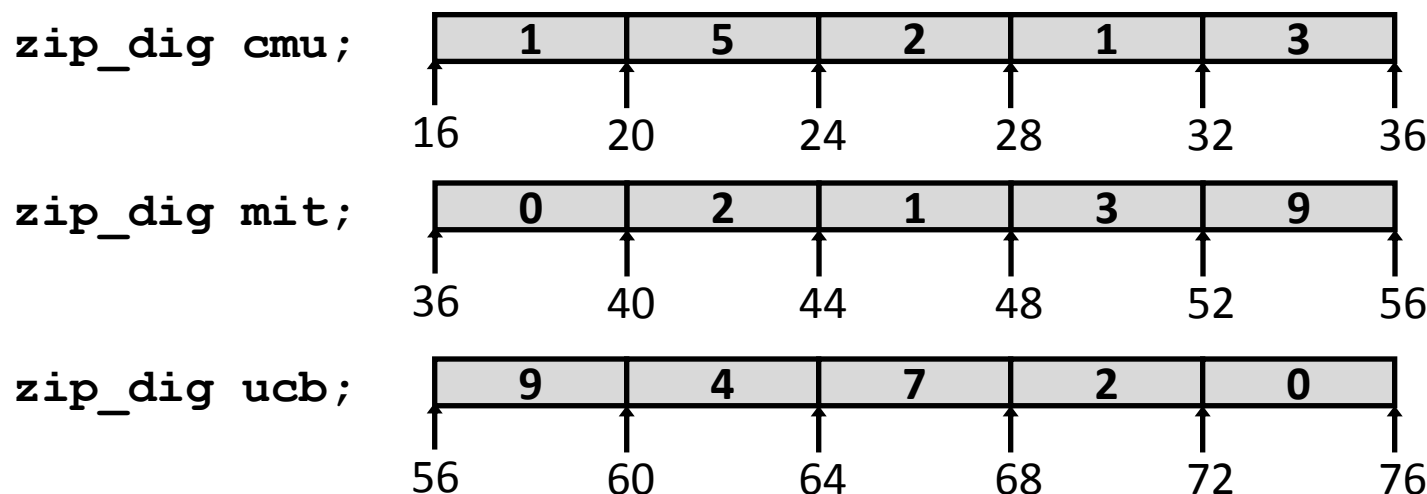
<code>val[4]</code>	<code>int</code>	3	
<code>val</code>	<code>int *</code>	x	
<code>val+1</code>	<code>int *</code>	$x + 4$	(aritmética de punteros)
<code>&val[2]</code>	<code>int *</code>	$x + 8$	
<code>val[5]</code>	<code>int</code>	??	
<code>*(val+1)</code>	<code>int</code>	5	(aritmética de punteros)
<code>val + i</code>	<code>int *</code>	$x + 4 i$	(aritmética de punteros)

* otros autores usan “(de)reference” en sentido mucho más estricto, para indicar el tipo puntero (o la operación de seguir el puntero) 18

Ejemplo de matrices

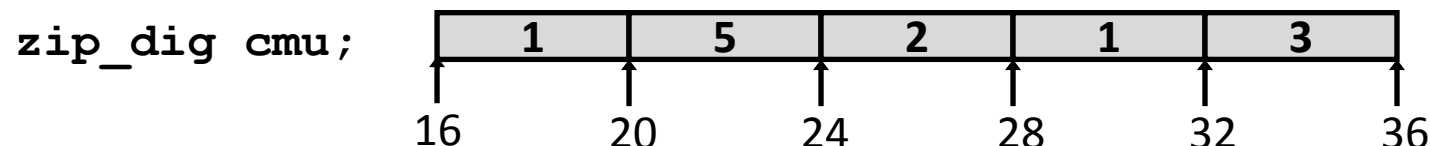
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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



- Declaración “`zip_dig cmu`” equivalente a “`int cmu[5]`”
- Las matrices del ejemplo se ubicaron en bloques sucesivos de 20 bytes
 - En general no está garantizado que suceda

Ejemplo de acceso a matrices



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

IA32

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```

- El registro `%edx` contiene dirección inicio de matriz
- El registro `%eax` contiene el índice a la matriz
- El dígito deseado está en $4 * \%eax + \%edx$
- Usar referencia a memoria `(%edx, %eax, 4)` *

Ejemplo de bucle sobre matriz (IA32)

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

```
                                # edx = z  
    movl    $0, %eax            # %eax = i  
.L4:                            # loop:  
    addl    $1, (%edx,%eax,4)    # z[i]++  
    addl    $1, %eax            # i++  
    cmpl    $5, %eax            # i:5  
    jne     .L4                 # if !=, goto loop
```

Ejemplo de bucle con puntero (IA32)

```
void zincr_p(zip_dig z) {
    int *zend = z+ZLEN;
    do {
        (*z)++;
        z++;
    } while (z != zend);
}
```

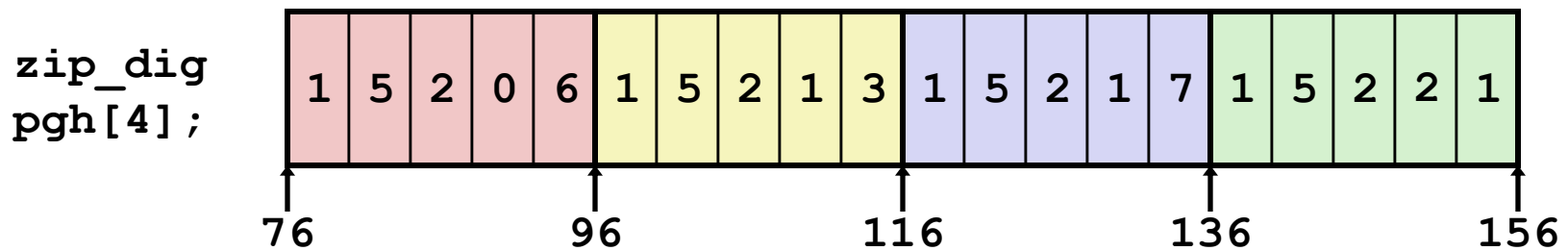


```
void zincr_v(zip_dig z) {
    void *vz = z;
    int i = 0;
    do {
        (*((int *) (vz+i)))++;
        i += ISIZE;
    } while (i != ISIZE*ZLEN);
}
```

	# edx = z = vz
movl \$0, %eax	# i = 0
.L8:	# loop:
addl \$1, (%edx,%eax)	# Increment vz+i
addl \$4, %eax	# i += 4
cmpl \$20, %eax	# Compare i:20
jne .L8	# if !=, goto loop

Ejemplo de matriz anidada

```
#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`: matriz de 4 elementos, ubicados contiguamente
 - Cada elemento es una matriz de 5 `ints`, ubicados contiguamente
- **Garantizado almacenamiento por filas (“row-major order”)**

Matrices multidimensionales (anidadas)

■ Declaración

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

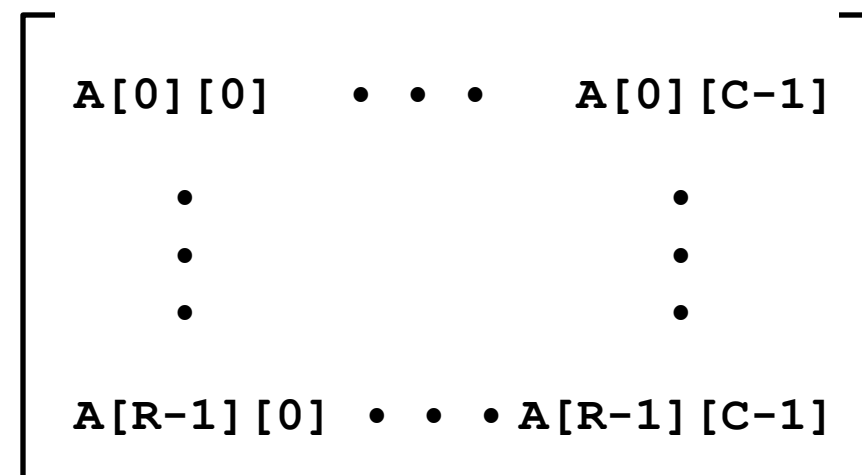
- Matriz 2D de (elems. de) tipo T
- R filas (rows), C columnas
- Elems. tipo T requieren K bytes

■ Tamaño matriz

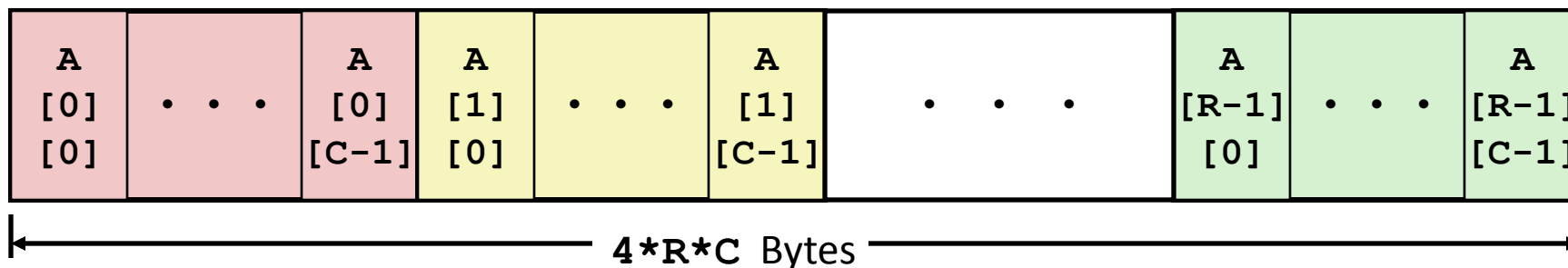
- $R * C * K$ bytes

■ Disposición

- Almacenamiento por filas



`int A[R][C];`

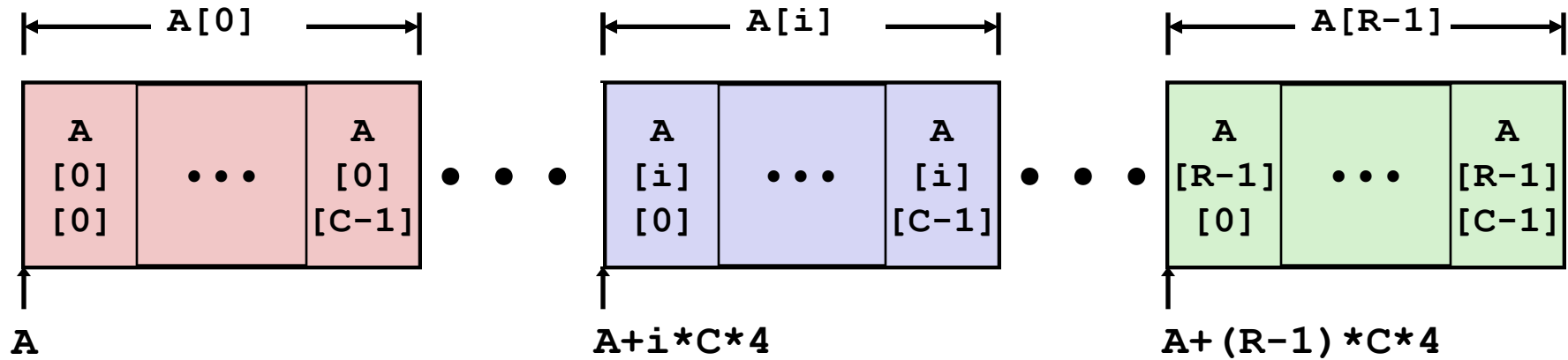


Acceso a filas en matrices anidadas

■ Vectores fila

- $A[i]$ es una matriz 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 matrices anidadas

```
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 }};
```

```
                                # %eax = index
leal (%eax,%eax,4),%eax # 5 * index
leal pgh(,%eax,4),%eax  # pgh + (20 * index)
```

■ Vector fila

- `pgh[index]` es array de 5 ints
- Dirección de comienzo `pgh+20*index`

■ Código IA32

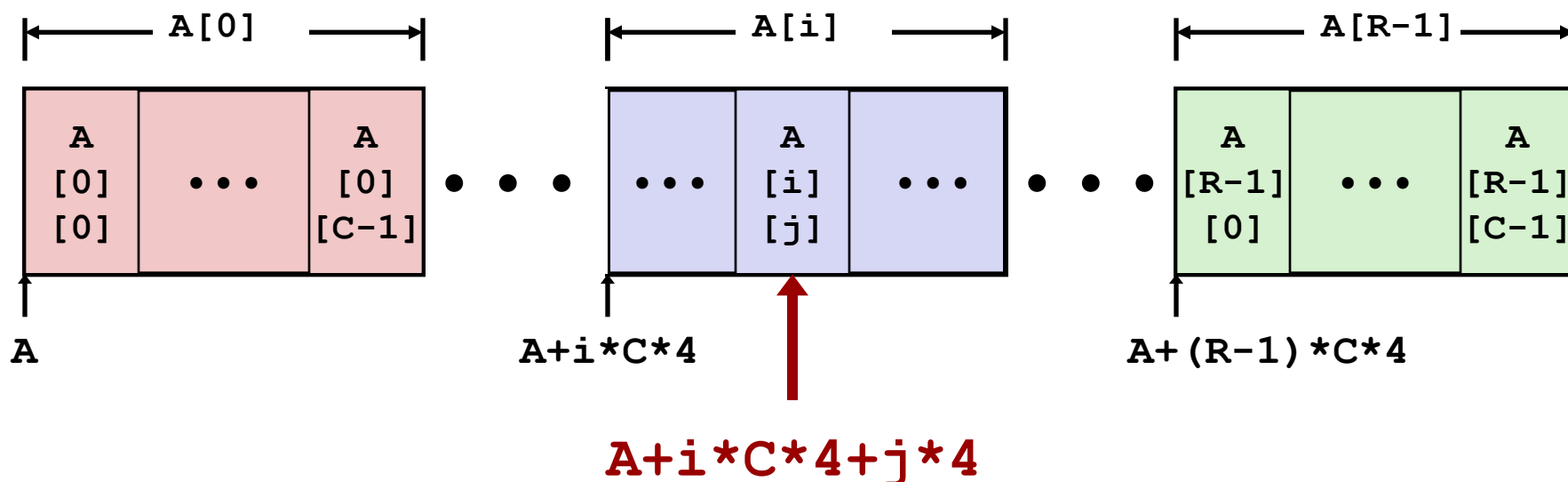
- Calcula y devuelve dirección
- La calcula como `pgh + 4*(index+4*index)`

Acceso a elementos en matrices anidadas

■ Elementos de la matriz

- $A[i][j]$ 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];
```



Código acceso elems. matrices anidadas

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

```
movl    8(%ebp), %eax          # index
leal    (%eax,%eax,4), %eax     # 5*index
addl    12(%ebp), %eax         # 5*index+dig
movl    pgh(,%eax,4), %eax     # offset 4*(5*index+dig)
```

■ Elementos de la matriz

- `pgh[index][dig]` es `int`
- Dirección: `pgh + 20*index + 4*dig`
 - `= pgh + 4*(5*index + dig)`

■ Código IA32

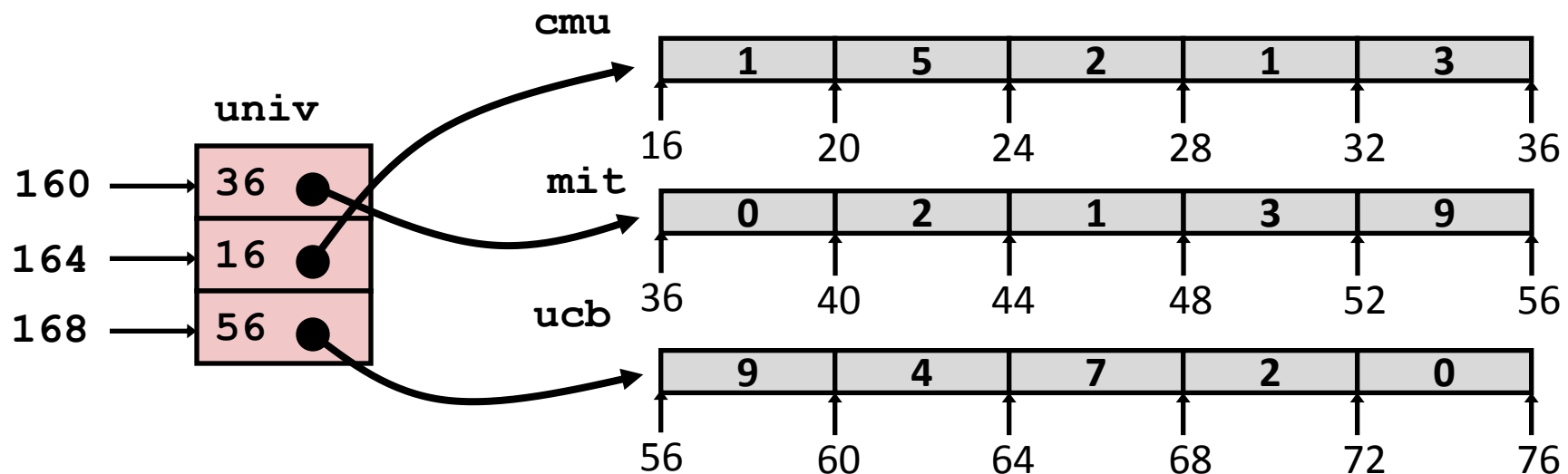
- Calcula la dirección `pgh + 4*((index+4*index)+dig)`

Ejemplo de matriz 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 una matriz de 3 elementos
- Cada elemento un puntero
 - 4 bytes
- Cada puntero apunta a una matriz de `ints`



Acceso a elementos en matriz multi-nivel

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

```
movl    8(%ebp), %eax          # index
movl    univ(,%eax,4), %edx     # p = univ[index]
movl    12(%ebp), %eax         # dig
movl    (%edx,%eax,4), %eax     # p[dig]
```

■ Cuentas (IA32)

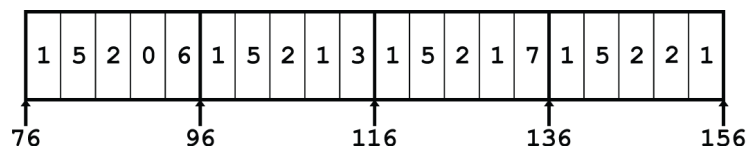
- Acceso a elemento **Mem[Mem[univ+4*index]+4*dig]**
- Debe hacer dos lecturas de memoria
 - Primero obtener puntero a la matriz* fila
 - Entonces acceder elemento dentro de la matriz*

* en estos casos, otros autores prefieren decir “vector” (1D), como opuesto a “matriz” (2D) 30

Acceso a elementos en matrices

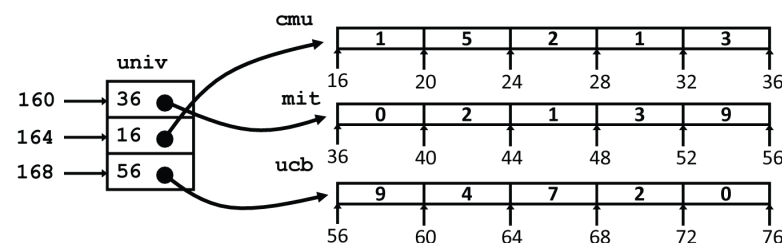
Matriz anidada

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



Matriz multi-nivel*

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



Accesos parecen similares en C, pero cuentas muy diferentes:

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

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

Cód. 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, int i, int j)
{
    return a[i][j];
}
```

■ Dimensiones variables, indexado explícito

- Forma tradicional de implementar matrices dinámicas

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

■ Dimensiones variables, indexado implícito

- Soportado ahora* por gcc

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

* ver <http://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html>
ISO C99, gcc lo soporta en C90/C89 y C++ desde antes v-2.95 1999

Acceso a matriz 16 X 16

■ Elementos de la matriz

- Dirección $A + i * (C * K) + j * K$
- $C = 16, K = 4$

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

```
movl    12(%ebp), %edx    # i  
sall    $6, %edx         # i*64  
movl    16(%ebp), %eax    # j  
sall    $2, %eax         # j*4  
addl    8(%ebp), %eax     # a + j*4  
movl    (%eax,%edx), %eax # *(a + j*4 + i*64)
```

Acceso a matriz n X n

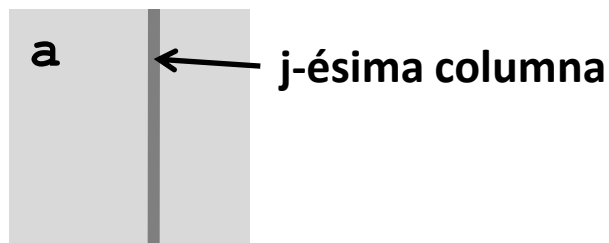
■ Elementos de la matriz

- Dirección $A + i * (C * K) + j * K$
- $C = n, K = 4$

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

```
movl    8(%ebp), %eax    # n  
sall    $2, %eax        # n*4  
movl    %eax, %edx      # n*4  
imull   16(%ebp), %edx   # i*n*4  
movl    20(%ebp), %eax   # j  
sall    $2, %eax        # j*4  
addl    12(%ebp), %eax   # a + j*4  
movl    (%eax,%edx), %eax # *(a + j*4 + i*n*4)
```

Optimizando acceso a matrices dims. fijas



■ Operación

- Pasar por todos los elementos de la columna j

■ Optimización

- Direccionando los sucesivos elementos de una misma columna (separados $C \cdot K$)

```
#define N 16  
typedef int fix_matrix[N][N];
```

```
/* Retrieve column j from array */  
void fix_column  
    (fix_matrix a, int j, int *dest)  
{  
    int i;  
    for (i = 0; i < N; i++)  
        dest[i] = a[i][j];  
}
```

Optimizando acceso a matrices dims. fijas

■ Optimización

- Calcular $ajp = \&a[i][j]$
 - Inicialmente $= a + 4*j$
 - Incrementos de $4*N$

Registro	Valor
%ecx	ajp
%ebx	dest
%edx	i

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

```
.L8:                                # loop:
    movl    (%ecx), %eax            #   Read *ajp
    movl    %eax, (%ebx,%edx,4)     #   Save in dest[i]
    addl    $1, %edx               #   i++
    addl    $64, %ecx              #   ajp += 4*N
    cmpl    $16, %edx              #   i:N
    jne     .L8                    #   if !=, goto loop
```

Optimizando acceso a matrices dims. variabl.

- Calcular $ajp = \&a[i][j]$
 - Inicialmente $= a + 4*j$
 - Incrementos de $4*n$

```
/* Retrieve column j from array */
void var_column
(int n, int a[n][n],
 int j, int *dest)
{
    int i;
    for (i = 0; i < n; i++)
        dest[i] = a[i][j];
}
```

Registro	Valor
%ecx	ajp
%edi	dest
%edx	i
%ebx	4*n
%esi	n

```
.L18:                                # loop:
    movl    (%ecx), %eax             #   Read *ajp
    movl    %eax, (%edi,%edx,4)      #   Save in dest[i]
    addl    $1, %edx                #   i++
    addl    %ebx, %ecx              #   ajp += 4*n
    cmpl    %edx, %esi              #   n:i
    jg      .L18                    #   if >, goto loop
```

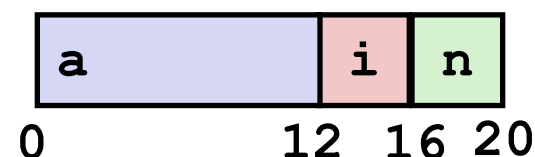
Progr. Máquina IV: Procs. x86_64 / Datos

- Procedimientos (x86-64)
- Matrices
 - Uni-dimensionales
 - Multi-dimensionales (anidadas)
 - Multi-nivel
- Estructuras
 - Ubicación
 - Acceso

Ubicación* de estructuras

```
struct rec {  
    int a[3];  
    int i;  
    struct rec *n;  
};
```

Disposic† Memoria



■ Concepto

- Se reserva* región contigua de memoria
- Referencia a miembros de la estructura mediante sus nombres
- Los miembros pueden ser de tipos diferentes

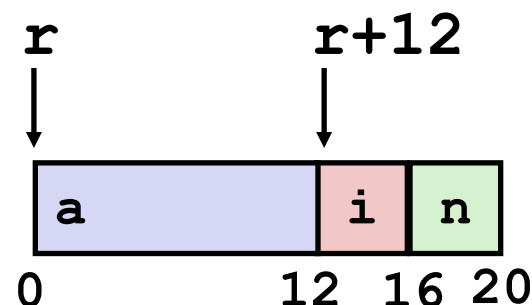
* "allocat(ion)/(ed)" en inglés

† "Memory Layout" = Disposición en Memoria. Para tamaño también se dice:

"Memory Footprint" = Ocupación de (Huella en) Memoria

Acceso a estructuras

```
struct rec {
    int a[3];
    int i;
    struct rec *n;
};
```



■ 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,
      int val)
{
    r->i = val;
}
```

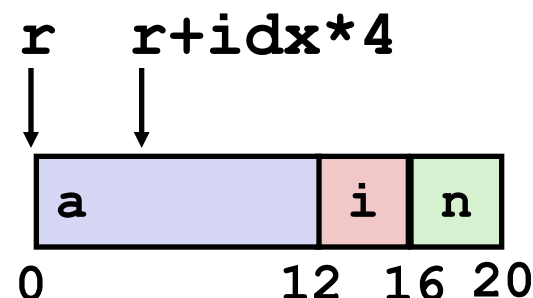
Ensamblador IA32

```
# %edx = val
# %eax = r
movl %edx, 12(%eax) # Mem[r+12] = val
```

* ptr->fld es abreviación para (*ptr).fld

Generando puntero a miembro estructura

```
struct rec {
    int a[3];
    int i;
    struct rec *n;
};
```



■ Generando puntero a un elemento de la matriz

- Desplaz. de cada miembro struct queda determinado en tiempo compilación
- Argumentos
 - `Mem[%ebp+8]: r`
 - `Mem[%ebp+12]: idx`

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

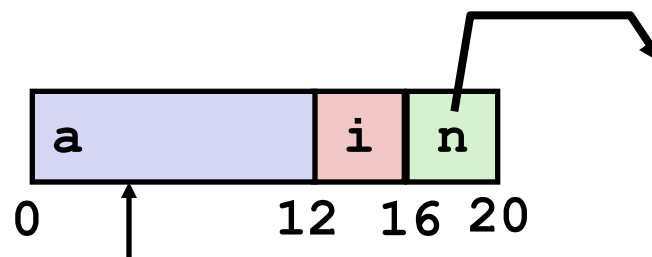
```
movl    12(%ebp), %eax    # Get idx
sall    $2, %eax          # idx*4
addl    8(%ebp), %eax     # r+idx*4
```

Siguiendo lista encadenada

■ Código C

```
struct rec {
    int a[3];
    int i;
    struct rec *n;
};
```

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



Elemento i

Registro	Valor
%edx	r
%ecx	val

```
.L17:                                # loop:
    movl    12(%edx), %eax            # r->i
    movl    %ecx, (%edx,%eax,4)      # r->a[i] = val
    movl    16(%edx), %edx           # r = r->n
    testl   %edx, %edx               # Test r
    jne     .L17                     # If != 0 goto loop
```

Resumen

■ Procedimientos en x86-64

- Marco pila es relativo a puntero pila
- Parámetros pasados en registros

■ Matrices

- Uni-dimensionales
- Multi-dimensionales (anidadas)
- Multi-nivel

■ Estructuras

- Ubicación
- Acceso

Guía de trabajo autónomo (4h/s)

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

- x86-64, Procedures, Data Structures, Observations.
 - 3.13.4 - .13.6 pp.316-325
 - Probl. 3.50 - 3.53 pp.318, 323-25
- Array Allocation & Access, Heterogeneous Data Structures (hasta Unions)
 - 3.8 – 3.9.1 pp.266-278
 - Probl. 3.35 – 3.39 pp.267-68, 270, 272, 277

Bibliografía:

[BRY11] Cap.3

Computer Systems: A Programmer's Perspective. Bryant, O'Hallaron. Pearson, 2011

Signatura ESIT/[C.1 BRY com](#)