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 ESIIT/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 ESIIT/C.1 BRY com

Progr. Máquina IV: Procs. x86_64 / Datos

- **Procedimientos (x86-64)**
- **Arrays***
  - Uni-dimensionales
  - Multi-dimensionales (anidados)
  - Multi-nivel
- **Estructuras**
  - Ubicación
  - Acceso

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

x86-64, Registros Enteros

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%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

** Consultar Wikipedia: "X86 calling conventions", x86-64 System V ABI*

x86-64, Registros

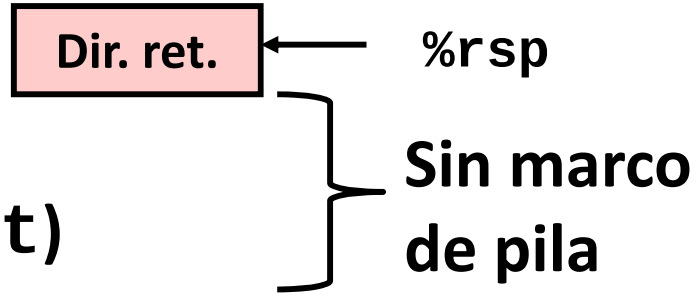
- **Argumentos pasados a funciones *vía* registros**
  - Si más de 6 parámetros enteros, pues pasar resto en pila
  - Estos registros pueden usarse también como salva-invocante
- **Todas las referencias a marco pila *vía* puntero pila**
  - Elimina la necesidad de actualizar **`%ebp/%rbp`**
- **Otros 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 (**xp**) en **%rdi**, 2º (**yp**) en **%rsi**
  - Punteros de 64-bit
- **No requiere operaciones en pila (salvo `ret`)**
- **Evita marco pila**
  - Puede mantener toda la información local en registros



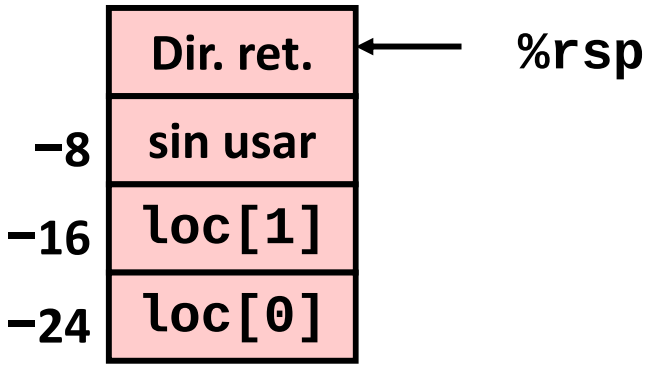
* ya visto en lección 2.1 pág.47

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*

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 MOVSXD 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*

* *"NonLeaf" (no-hoja) en inglés*

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

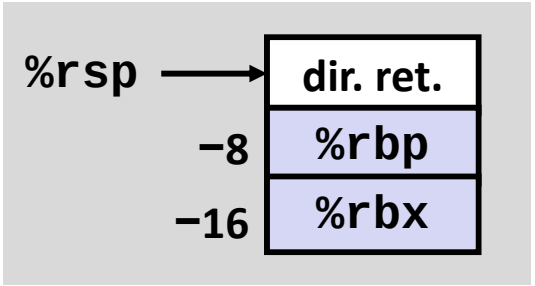
```
swap_ele_su:
    movq    %rbx, -16(%rsp)         # Salvar %rbx
    movq    %rbp, -8(%rsp)          # Salvar %rbp
    subq    $16, %rsp               # Reservar* marco pila
    movslq  %esi,%rax               # Extender i (signo)
    leaq    8(%rdi,%rax,8), %rbx    # &a[i+1] (salva invocado)
    leaq    (%rdi,%rax,8), %rbp     # &a[i]   (salva invocado)
    movq    %rbx, %rsi              # 2º argumento
    movq    %rbp, %rdi              # 1er 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*

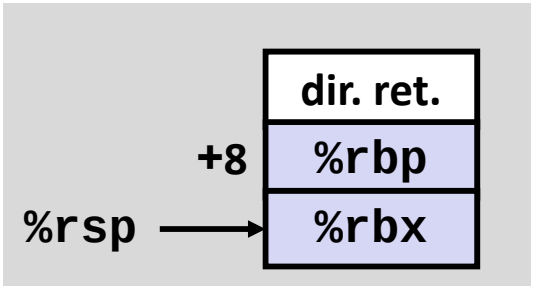
Comprendiendo Marco Pila x86-64

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




```
subq    $16, %rsp              # Reservar marco pila
```

● ● ●




```
movq    (%rsp), %rbx           # Restaurar %rbx
movq    8(%rsp), %rbp          # 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)
- **Arrays**
  - Uni-dimensionales
  - Multi-dimensionales (anidados)
  - 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)

Punto Flotante

- Almacenados & manipulados en registros punto 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*

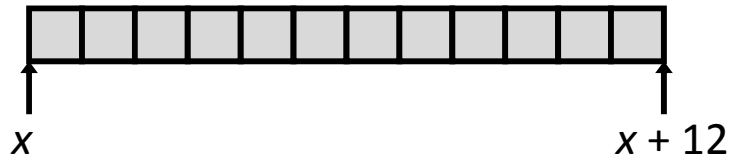
Ubicacion* de Arrays

- **Principio Básico**

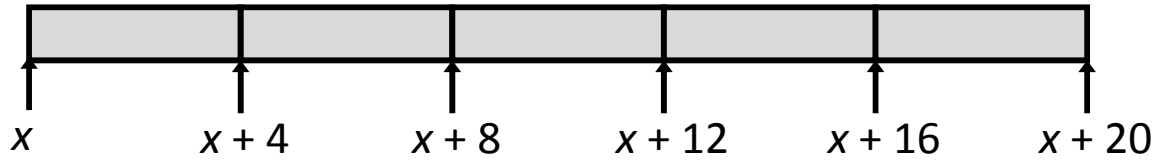
  T **A[***L***];**

  - Array de tipo *T* y longitud *L*
  - Se reserva* región contigua de *L* * **sizeof**(*T*) bytes

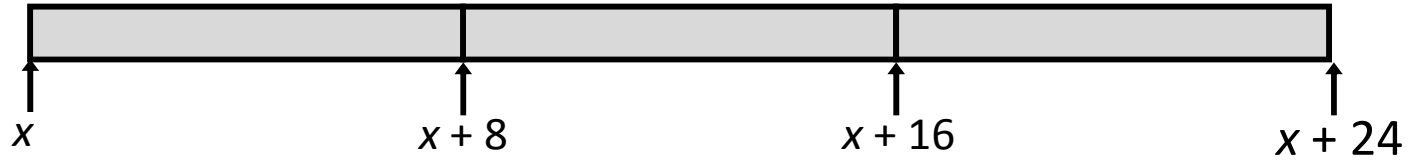
char string[12];



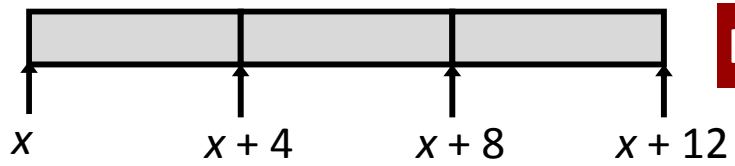
int val[5];



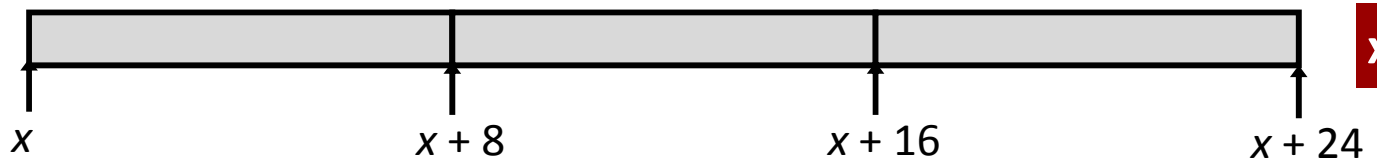
double a[3];



char *p[3];



IA32



x86-64

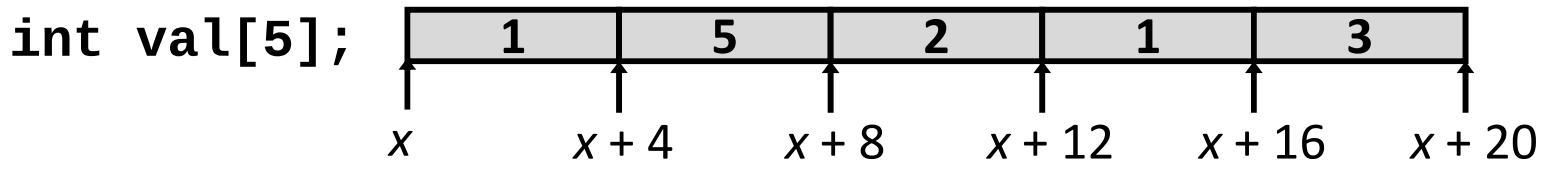
* *"allocat(ion)/(ed)" en inglés*

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

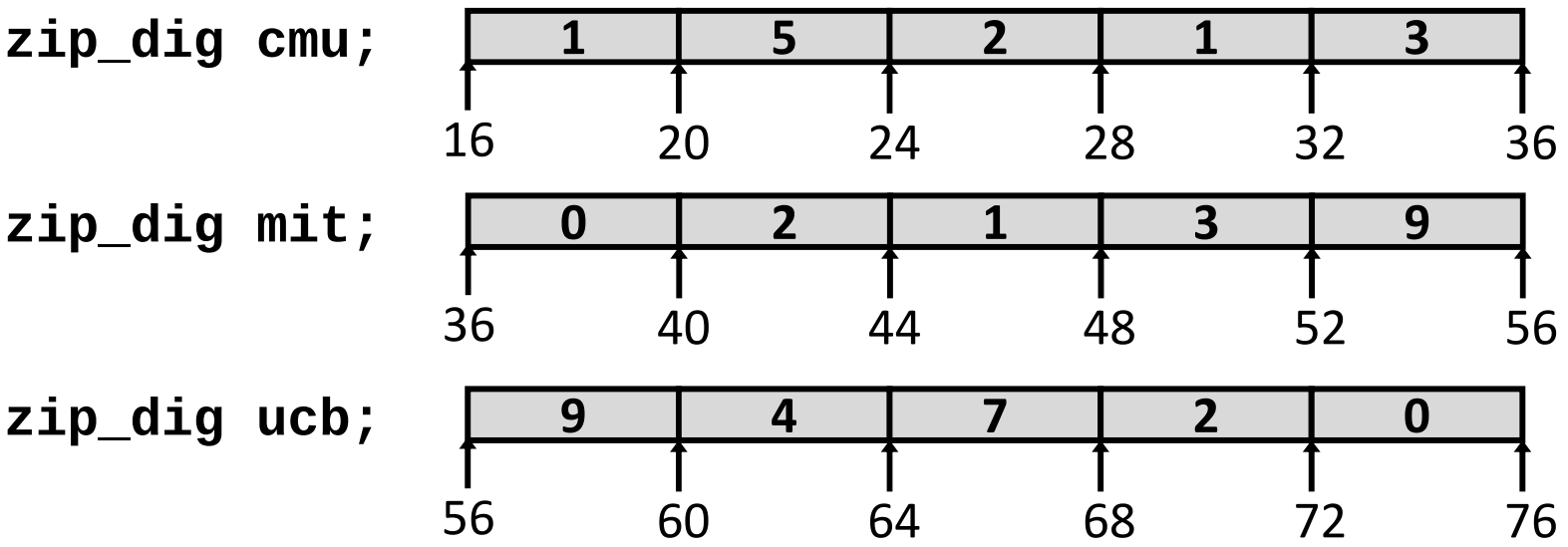


Referencia*	Tipo	Valor	
`val[4]`	`int`	3	
`val`	`int *`	*x*	
`val+1`	`int *`	*x* + 4	(aritmética de punteros)
`&val[2]`	`int *`	*x* + 8	
`val[5]`	`int`	??	
`*(val+1)`	`int`	5	(aritmética de punteros)
`val + i`	`int *`	*x* + 4 *i*	(aritmética de punteros)

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

Ejemplo de Arrays

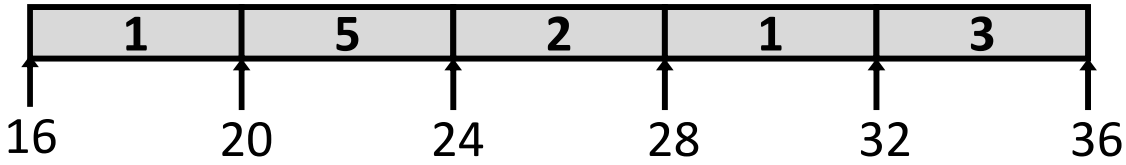
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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



- **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

Ejemplo de Acceso a Arrays

```
zip_dig cmu;
```




```
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 del array
- El registro %eax contiene el índice al array
- El dígito deseado está en 4*%eax + %edx
- Usar referencia a memoria (%edx,%eax,4) *

** en sentido manuales Intel IA-32, "direccionamiento a memoria"*

Ejemplo de Bucle sobre Array (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);
}
```

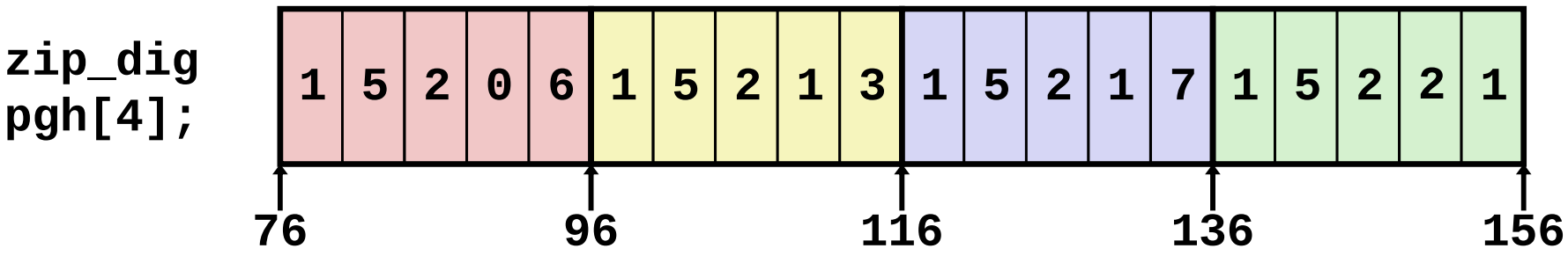
```
void zincr_v(zip_dig z) {
  void *vz = z;
  int i = 0;
  do {
    (*((int *) (vz+i)))++;
    i += ISIZE;
  } while (i != ISIZE*ZLEN);
}
```

```
                              # 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
```

** puntero a void cuenta de 1 en 1, por eso i+=ISIZE;*

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")**

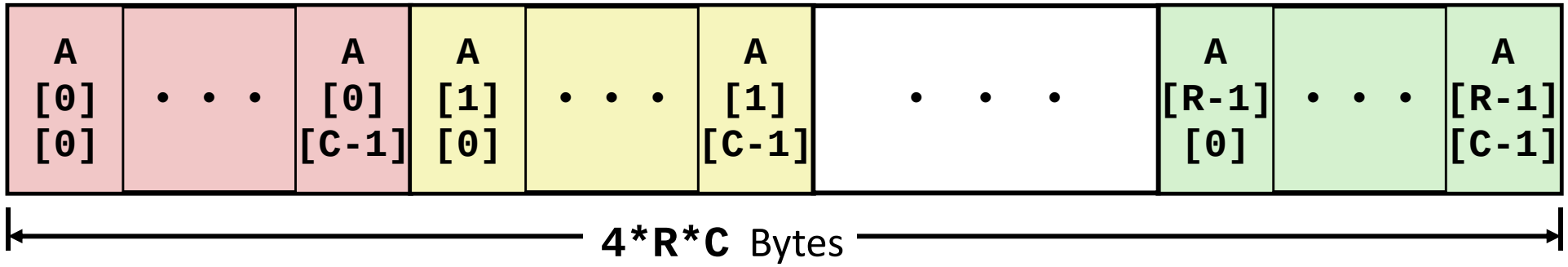
** también existe "column-major order", usado por FORTRAN (y consecuentemente por LAPACK, MATLAB, R, Octave…)*

Arrays Multidimensionales (Anidados)

- **Declaración**
  - *T* **A**[*R*][*C*];
  - Array 2D de (elementos de) tipo *T*
  - *R* filas (rows), *C* columnas
  - Elems. tipo *T* requieren *K* bytes
- **Tamaño Array**
  - *R* * *C* * *K* bytes
- **Disposición**
  - Almacenamiento por filas

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

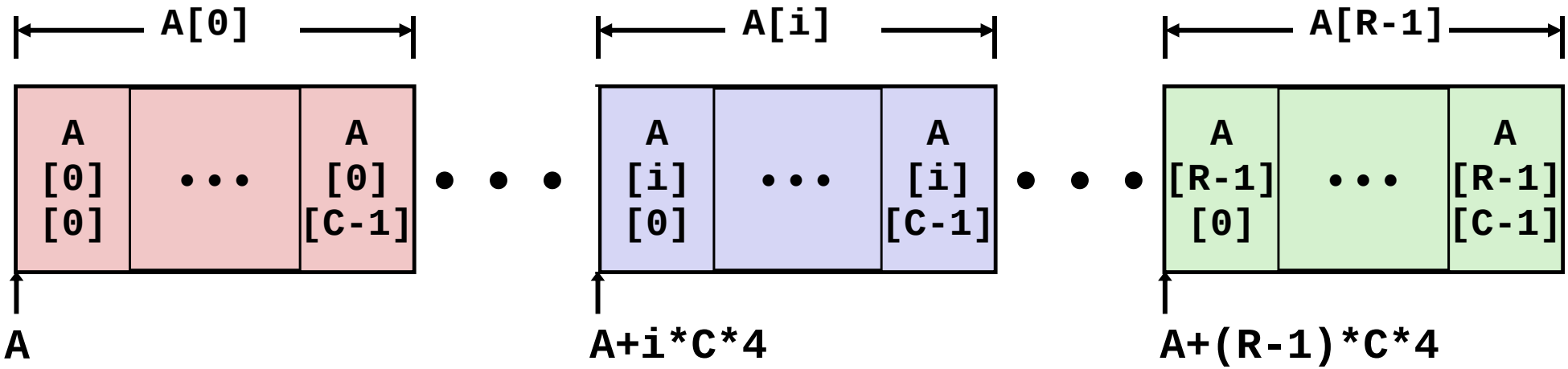
```
int A[R][C];
```



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(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 }};
```

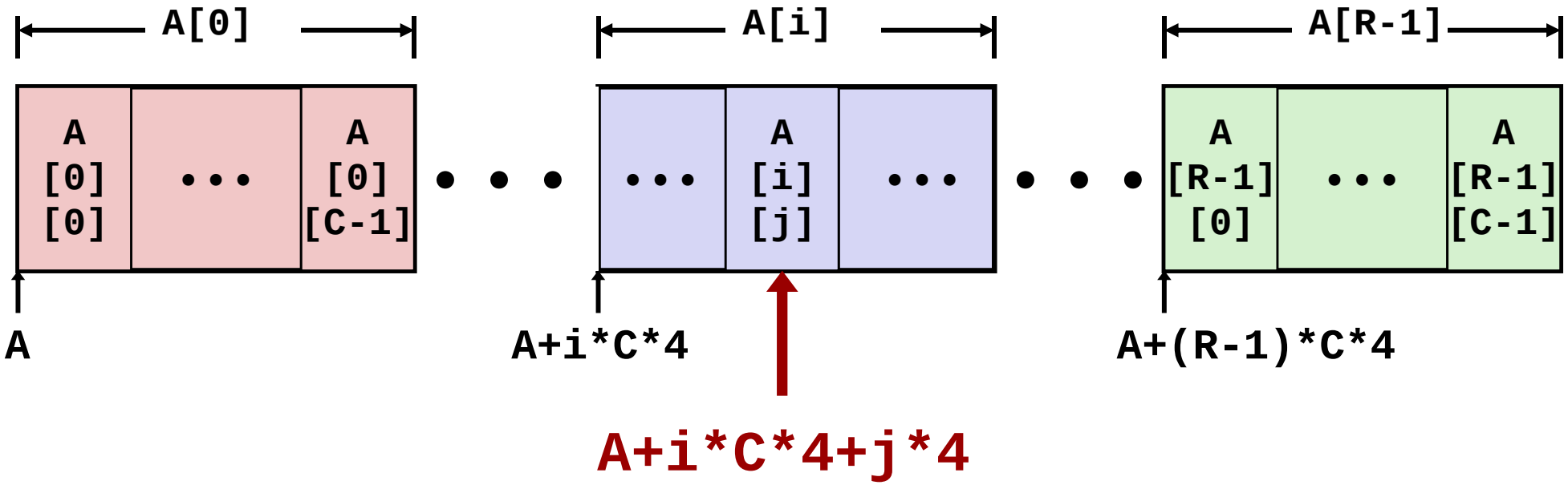
```
                              # %eax = index
   leal (%eax,%eax,4),%eax    # 5 * index
   leal pgh(,%eax,4),%eax     # pgh + (20 * index)
```

- **Vector Fila**
  - **`pgh[index]`** es array de 5 **`int`**'s
  - Dirección de comienzo **`pgh+20*index`**
- **Código IA32**
  - Calcula y devuelve dirección
  - La calcula como **`pgh + 4*(index+4*index)`**

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];
```



Código Acceso Elementos Arrays Anidados

```
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)
```

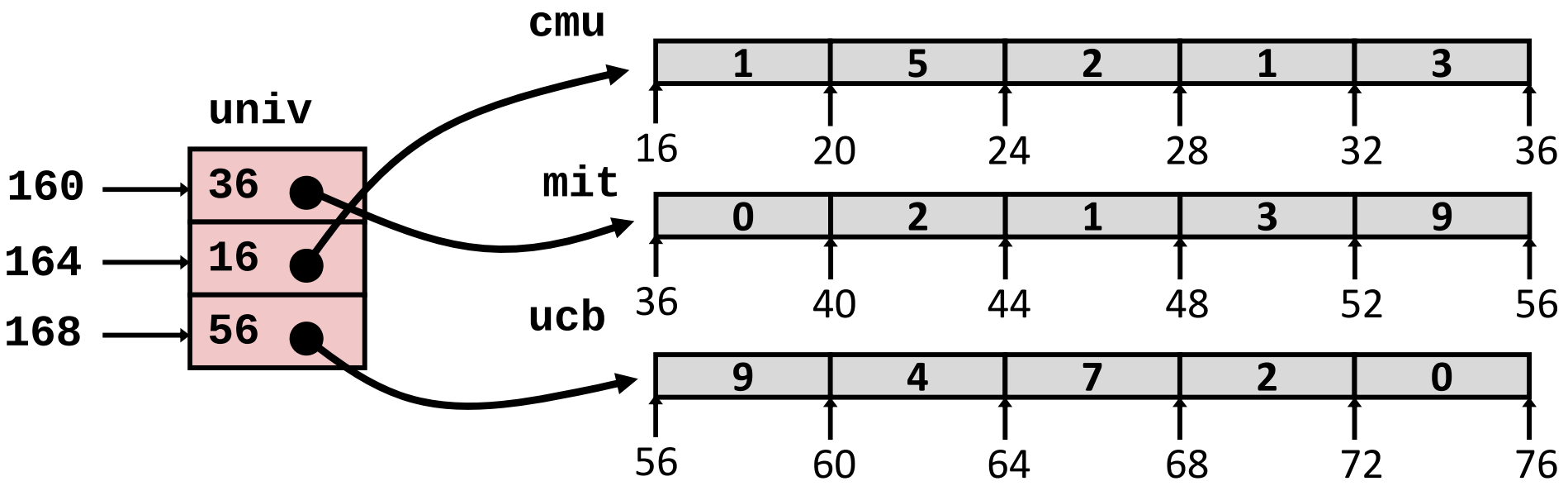
- **Elementos del Array**
  - `pgh[index][dig]` es `int`
  - Dirección: `pgh + 20*index + 4*dig`
    - = `pgh + 4*(5*index + dig)`
- **Código IA32**
  - Calcula la dirección `pgh + 4*((index+4*index)+dig)`

Ejemplo de 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
  - 4 bytes
- Cada puntero apunta a un array de `int`'s



** es más frecuente decir "array de punteros"*

Acceso a Elementos en Array 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]
```

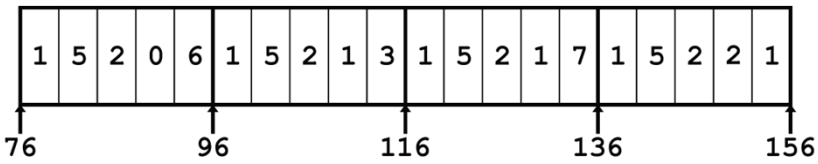
- **Cuentas (IA32)**
  - Acceso a elemento **`Mem[Mem[univ+4*index]+4*dig]`**
  - Debe hacer dos lecturas de memoria
    - Primero obtener puntero al array* fila
    - Entonces acceder elemento dentro del array*

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

Acceso a Elementos en Arrays

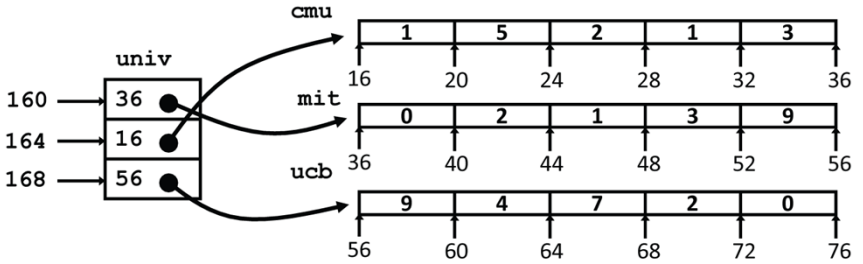
Array anidado

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



Array Multi-nivel*

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



Accesos parecen similares en C, pero cuentas muy diferentes:

`Mem[pgh+20*index+4*dig]`

`Mem[Mem[univ+4*index]+4*dig]`

** "array de punteros"*

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 arrays dinámicos

```
#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 del Array**
  - 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)
```

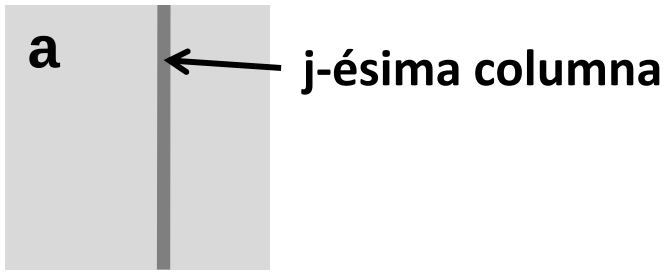
Acceso a Matriz n X n

- **Elementos del Array**
  - 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 Arrays Dims. Fijas




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

- **Operación**
  - Pasar por todos los elementos de la columna j
- **Optimización**
  - Direccionando los sucesivos elementos de una misma columna (separados *C***K*)

```
/* 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 Arrays 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 Arrays Dims. Variables

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

Registro	Valor
%ecx	ajp
%edi	dest
%edx	i
%ebx	4*n
%esi	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];
}
```

```
.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
```

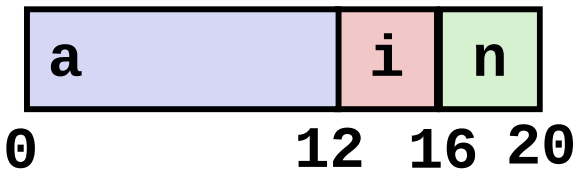
Progr. Máquina IV: Procs. x86_64 / Datos

- Procedimientos (x86-64)
- Arrays
  - Uni-dimensionales
  - Multi-dimensionales (anidados)
  - Multi-nivel
- **Estructuras**
  - Ubicación
  - Acceso

Ubicación* de Estructuras

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

Disposic† Memoria

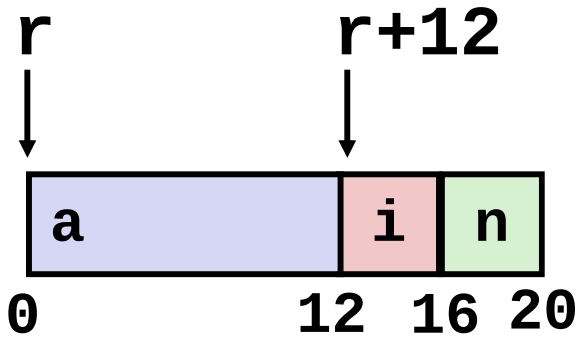


- **Concepto**
  - Se reserva* región contigua de memoria
  - Referencia a miembros de la estructura mediante sus nombres
  - Los miembros pueden ser de tipos diferentes

** "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

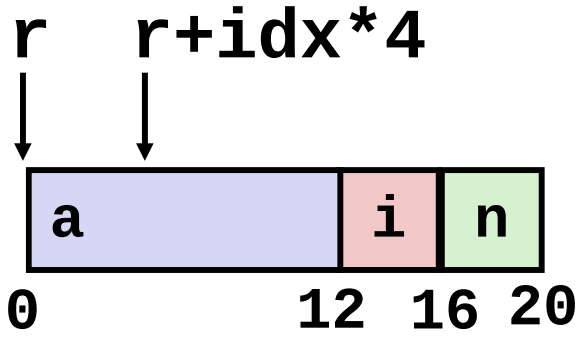
```
                          # %edx = val
                          # %eax = r
movl %edx, 12(%eax)       # Mem[r+12] = val
```

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

Si se declara "struct rec R;", entonces "R.a" es array, "R.a[0]" y "R.i" son enteros, "R.n" es puntero, y "R.n->a" vuelve a ser array

Generando Puntero a Miembro Estructura

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



- **Generando Puntero a un Elemento del Array**
  - 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];
}
```

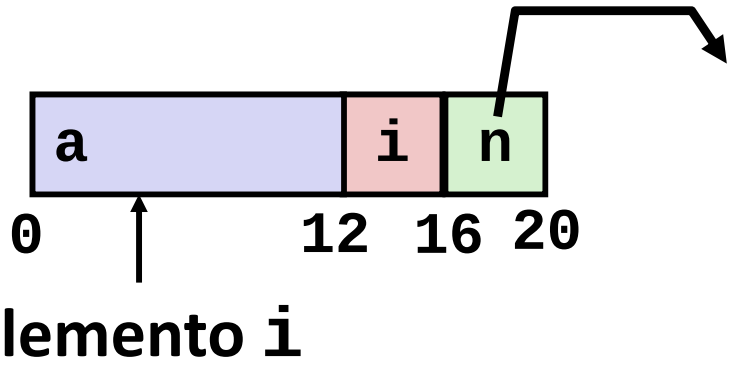
```
movl   12(%ebp), %eax   # Get idx
sall   $2, %eax         # idx*4
addl   8(%ebp), %eax    # r+idx*4
```

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;
  }
}
```



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
- **Arrays**
  - Uni-dimensionales
  - Multi-dimensionales (anidados)
  - 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 ESIIT/C.1 BRY com