Programación a Nivel Máquina

Estructura de Computadores. Tema 2.

(III) Procedimientos

Ejemplo ilustrativo de Recursividad

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) +
               pcount_r(x >> 1);
}
```

```
pcount_r:
        movl    $0, %eax
        testq   %rdi, %rdi
        je      .L6
        pushq %rbx
        movq    %rdi, %rbx
        andl    $1, %ebx
        shrq    %rdi
        call    pcount_r
        addq    %rbx, %rax
        popq    %rbx
.L6:
        rep; ret
```

Consideraciones de la Recursividad

- Se pueden almacenar valores tranquilamente en el **marco de pila** local y en **registros salva-invocado**.
- Poner argumentos 7+ de la función en tope de pila.
- Devolver resultado en **%rax**.

(IV) Datos

Arrays

Unidimensionales

```
T A[L];
```

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

Multi-dimensionales (Anidados)

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

- Array 2D de tipo **T** con **R** filas y **C** columnas
- Tamaño Array **R * C * sizeof(T)** bytes
- **Vectores Fila:** A[i] es un array de C elementos con dirección de comienzo **A + i * (C * sizeof(T))**

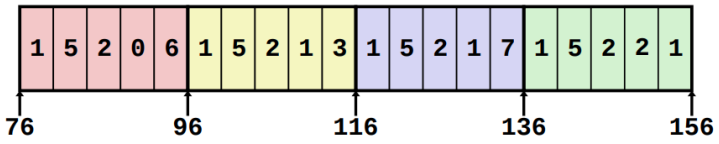
Intel	ASM‡	Bytes	C
byte	**b**	1	**[unsigned] char**
word	**w**	2	**[unsigned] short**
double word	**l**	4	**[unsigned] int**
quad word	**q**	8	**[unsigned] long int** (x86-64)

```
#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 }};
```

Ejemplo de **Array Anidado** →

➔ "zip_dig pgh[4]" equivalente a "int pgh[4][5]"

`zip_dig pgh[4];`




```
get_pgh_zip:
                                # index en %rdi
   leaq  (%rdi,%rdi,4), %rax    # 5 * index

†  leaq   pgh(,%rax,4), %rax    # pgh + (20 * index)
   ret
```

← Acceder al elemento (vector) index guardado en %rdi

- **Elementos del Array:** A[i][j] elemento de tipo T, que requiere K bytes. Dirección:

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

Multi-nivel *(todo igual menos:)*

- **Elementos del Array:** A[i][j] elemento de tipo T, que requiere K bytes. Dirección:

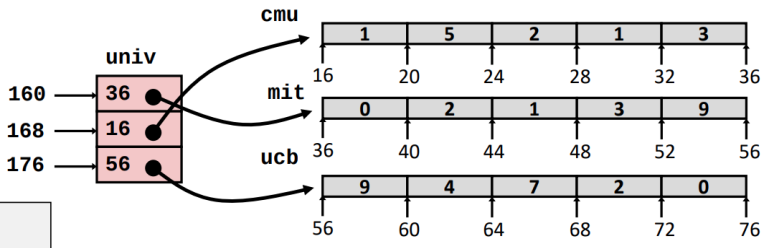
Mem[A + i * 8)] + j * K

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};
```

Acceder al elemento [index][digit] guardado en [%rdi][%rax]




```
get_univ_digit:

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

Estructuras

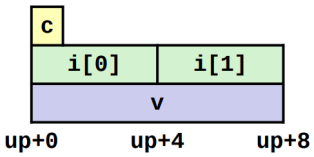
Ubicación

- Estructuras representadas como un bloque de memoria contigua.
- Campos ordenados según la declaración.

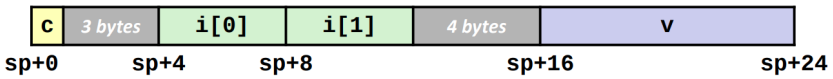
Acceso

- Un **puntero** indica el primer byte de la estructura.
- Acceder a los elementos mediante sus desplazamientos.

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




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



Alineamiento

- Si el tipo de datos primitivo requiere **K bytes** ⇒ La dirección debe ser **múltiplo de K.**
- **Motivación:** A la memoria se accede en trozos de 4 ú 8 bytes ⇒ **Eficiencia**
- El compilador inserta huecos en la estructura para asegurar un correcto alineamiento.
- Si el requisito de alineamiento **máximo es K** el **struct** entero debe ocupar **múltiplo de K**.
- **Ahorro de Espacio**: Poner primero los tipos de datos grandes.

Uniones

- **Reservar** de acuerdo al **elemento más grande** ⇒ Sólo puede usarse un campo a la vez.
- Si se cambia un elemento ⇒ Se cambia el resto que comparta memoria.
- **IMPORTANTE**: Little Endian → Se guardan primero los LSB's (de 2 en 2)