Práctica de Organización del Computador II

Programación orientada a datos - Tercera Parte

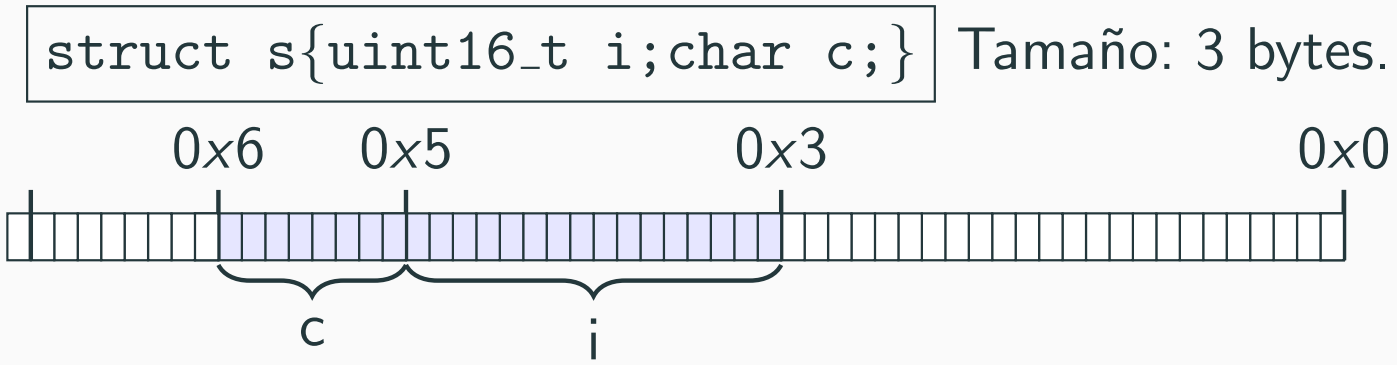
Segundo Cuatrimestre 2023

Organización del Computador II
DC - UBA

Alineación de los datos en memoria

Leyendo un dato desalineado

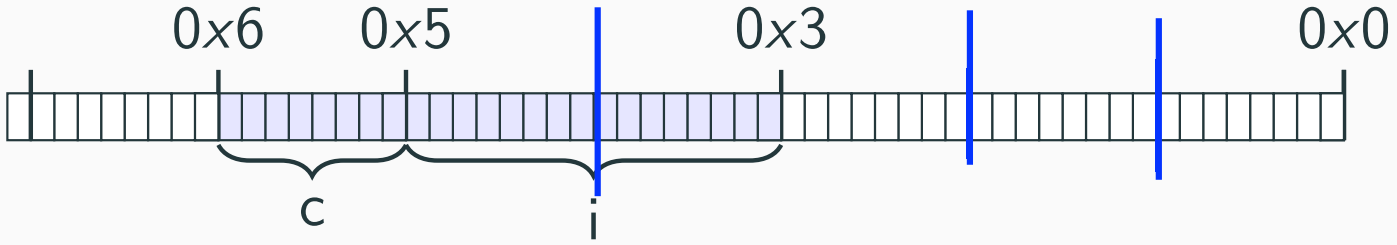
Retomemos la definición de una estructura de la clase anterior, habíamos supuesto cierta ubicación en memoria.



En particular supusimos que los datos se ubicaban de forma contigua, uno al lado del otro.

Leyendo un dato desalineado

Ahora **imaginen que sólo podemos leer direcciones de memoria que sean múltiplo de cuatro y que el tamaño de nuestro buffer de memoria es de 4 bytes**.



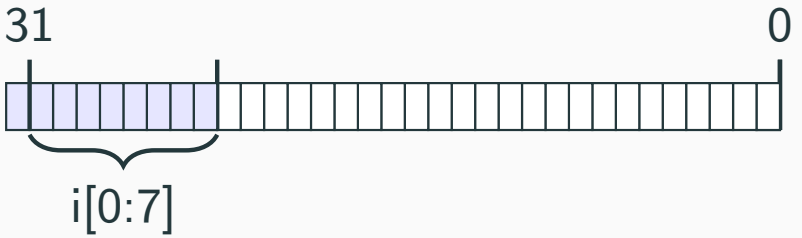
¿Cómo cargamos `i` en `AX` (2 bytes)?

Respuesta:

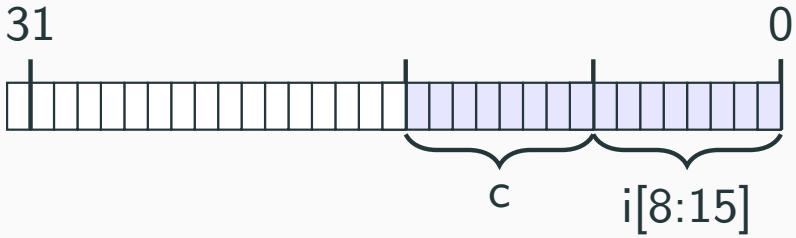
Tenemos que hacer dos lecturas, una para parte baja empezando en `0x0` y otra para la parte alta empezando en `0x4`, y luego tenemos que combinar las dos partes en `AX`.

Leyendo un dato desalineado

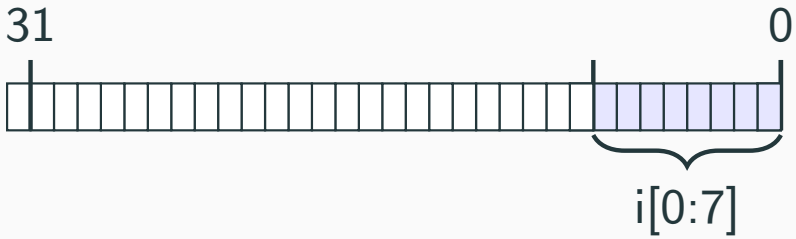
Reconstruyendo `i`.



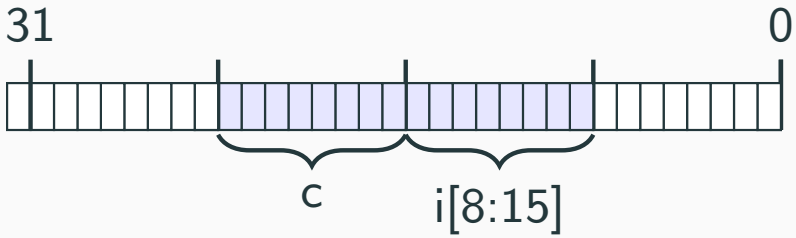

```
MOV R10d, [RBX]
```




```
MOV R11d, [RBX + 4]
```



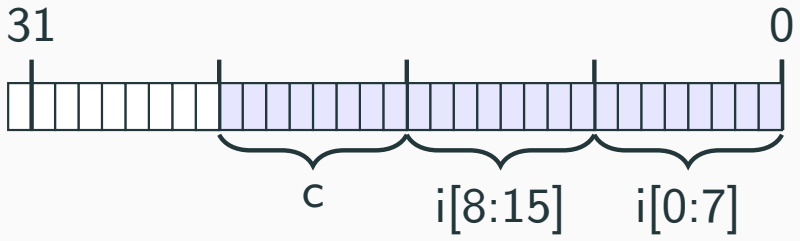

```
SHR R10d, 24
```



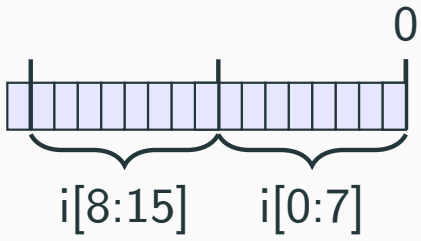

```
SHL R11d, 8
```

Leyendo un dato desalineado

Reconstruyendo `i`.




```
OR R10d, R11d
```




```
MOV AX, R10w
```

¿Por qué el procesador haría lecturas en múltiplos de cuatro si puede direccionar a byte?

Respuesta:

Algunas funciones de `libc` aprovechan los registros largos (`XMM`,`YMM` de 128 o 256 bytes) y utilizan operaciones que presuponen (**contrato de datos**) que los datos están alineados en estas longitudes para no tener que hacer este tipo de reconstrucciones.

Datos alineados a n bytes

Vamos a decir que un dato está **alineado a n bytes** si la posición en la que comienza es múltiplo de **n**. Vamos a decir que un tamaño de dato está **alineado a n bytes** si es múltiplo de **n**.

Dato `x` alineado a `n` bytes $\rightarrow$ posicion múltiplo de `n`

Tamaño de `x` alineado a `n` bytes $\rightarrow$ `sizeof(x)` múltiplo de `n`

Datos alineados a n bytes

Se presupone que las estructuras en `C` compiladas con `GCC` cumplen con el siguiente contrato:

- **El tamaño de la estructura está alineado al tamaño del atributo más grande**
  (si el atributo más grande es de 4 bytes, la estructura debe tener un tamaño de múltiplo de cuatro).
- **Cada atributo está alineado según su tamaño**
  (si un atributo es de dos bytes va a estar siempre en posiciones pares).

Datos alineados a n bytes

¿Pero cómo se asegura el compilador que las estructuras cumplen este contrato?

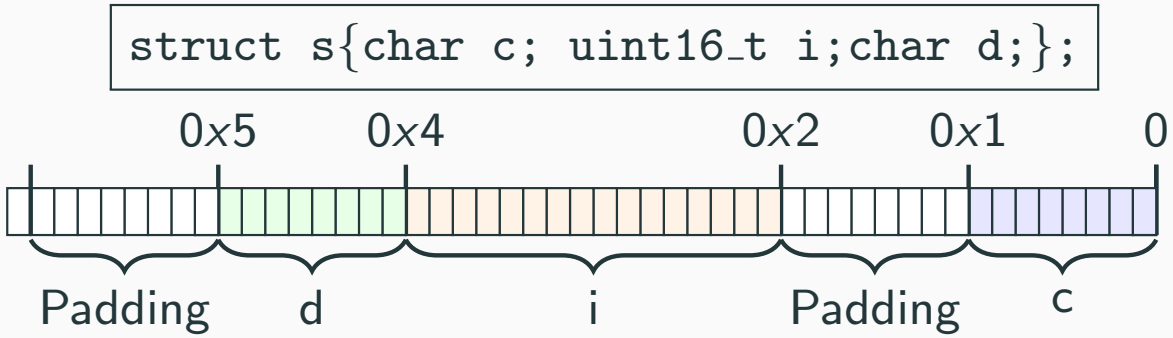
Respuesta:

Va a dejar espacios vacíos entre los atributos para respetar la alineación de los mismos, y también va a dejar espacio al final de una estructura si hace falta.

A estos espacios los llamaremos **padding**.

Structs bien alineados

Estudiemos cómo se alinea el siguiente struct:

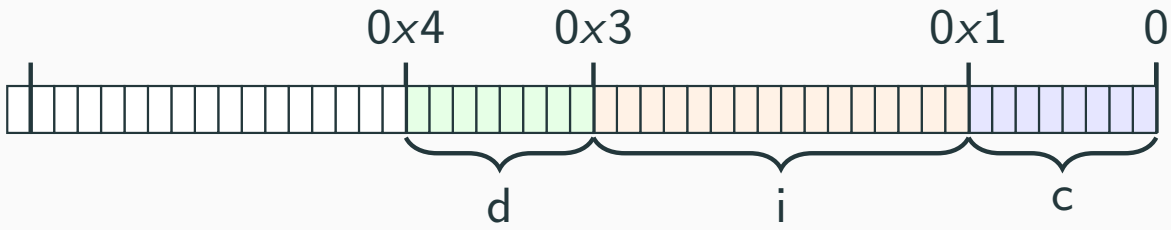


El padding junto a `c` se debe a que `i` tiene que posicionarse en una dirección alienada a su tamaño (2 bytes), el padding luego de `d` se debe a que el tamaño de la estructura debe quedar alineado al tamaño de su atributo más grande (2 bytes).

Structs bien alineados

Podemos indicar explícitamente que queremos que los atributos de nuestra estructura se encuentren en posiciones contiguas con el modificador `__attribute__((__packed__))`.

```
struct s{char c; uint16_t i;char d;}__attribute__((__packed__));
```

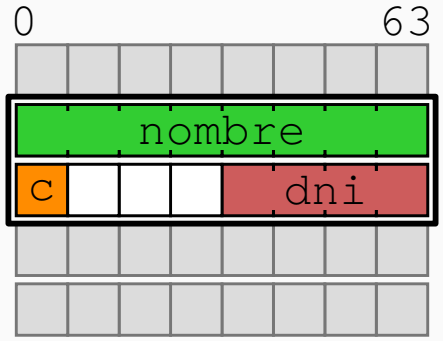


¿Cuándo trabajemos en ASM con una estructura cómo sabemos que tipo de alienación tiene?

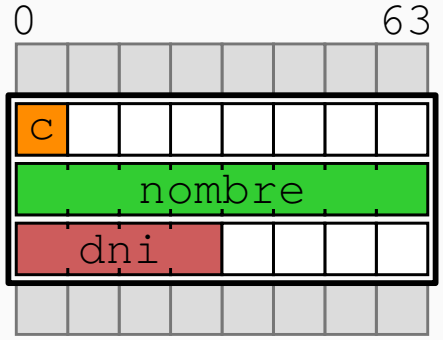
Respuesta: Es un contrato de uso del dato, no sabemos sólo con observar el dato cómo fue compilado, debemos definir con antelación si es un dato empaquetado (packed) o no.

Ejemplos: → SIZE ⇒ OFFSET

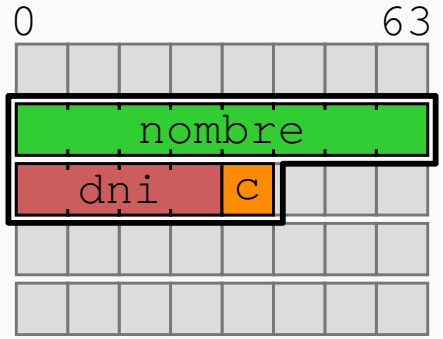
```
struct alumno {
 char* nombre;      → 8      ⇒ 0
 char comision;     → 1      ⇒ 8
 int dni;           → 4      ⇒ 12
};                           ⇒ 16
```




```
struct alumno2 {
 char comision;     → 1      ⇒ 0
 char* nombre;      → 8      ⇒ 8
 int dni;           → 4      ⇒ 16
};                           ⇒ 24
```




```
struct alumno3 {
 char* nombre;      → 8      ⇒ 0
 int dni;           → 4      ⇒ 8
 char comision;     → 1      ⇒ 12
} __attribute__((packed));   ⇒ 13
```



Uso

Definición:

```
struct estudiante {
        char* nombre;
        char comision;
        int dni;
};
```

Uso en C:

```
struct estudiante  alu1;
alu1.nombre = 'carlos';
alu1.dni = alu.dni + 10;
alu1.comision = 'a';

struct estudiante  *alu2;
alu2->nombre = 'carlos';
alu2->dni = alu.dni + 10;
alu2->comision = 'a';
```

Uso en ASM:

```
%define off_nombre 0
%define off_comision 8
%define off_dni 12


mov rsi, ptr_struct
mov rbx, [rsi+off_nombre]
mov al, [rsi+off_comision]
mov edx, [rsi+off_dni]
```

Uso: Mejor

Definición:

```
typedef struct estudiante {
        char* nombre;
        char comision;
        int dni;
}estudiante_t;
```

Uso en C:

```
estudiante_t  alu1;
alu1.nombre = 'carlos';
alu1.dni = alu.dni + 10;
alu1.comision = 'a';

estudiante_t  *alu2;
alu2->nombre = 'carlos';
alu2->dni = alu.dni + 10;
alu2->comision = 'a';
```

Uso en ASM:

```
%define off_nombre 0
%define off_comision 8
%define off_dni 12


mov rsi, ptr_struct
mov rbx, [rsi+off_nombre]
mov al, [rsi+off_comision]
mov edx, [rsi+off_dni]
```

¿Consultas?