Packing, Padding y Alignment en C++

TPV2

Samir Genaim

¿Cuál es el tamaño de un struct?

```
struct T {
    int a;
    char b;
    long c;
    char d;
};

int main(int, char**) {
    std::cout << sizeof(T) << std::endl;
    std::cout << sizeof(int)+sizeof(long)+2*sizeof(char) << std::endl;
}
```

Este programa escribe:
24
14

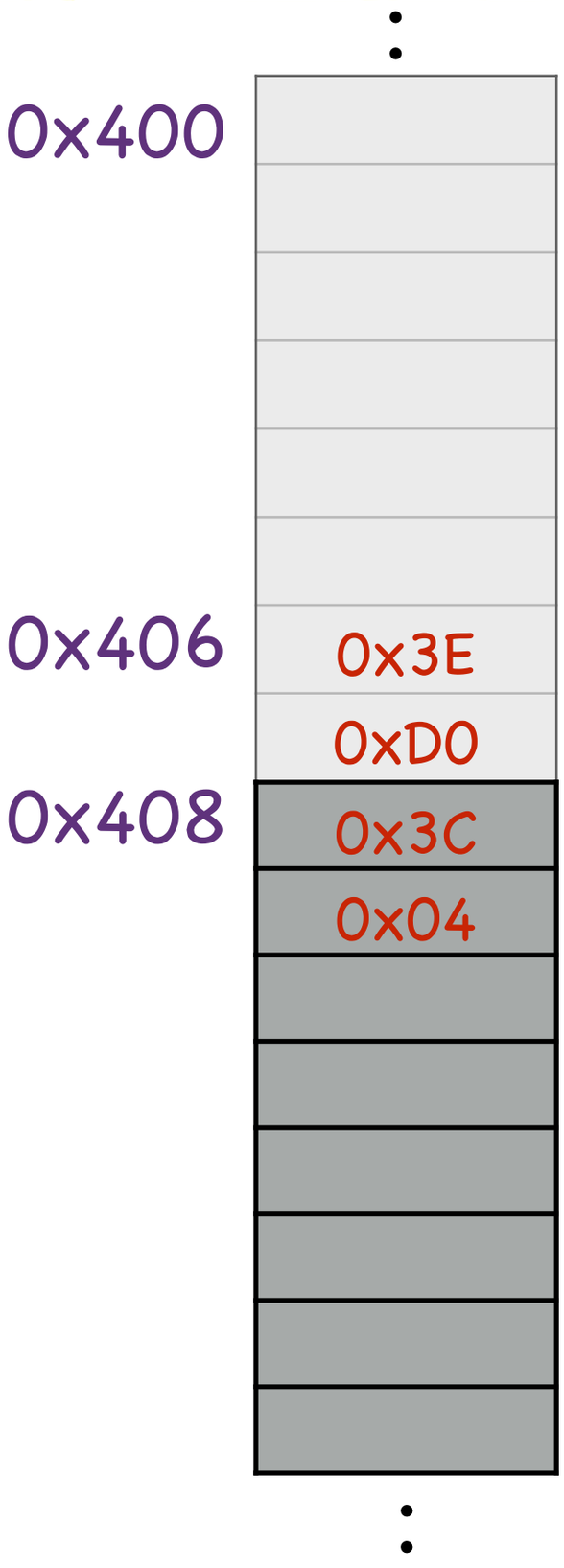
¿Porque el tamaño del struct T es más de la suma de los tamaños de sus miembros?

Antes de empezar ..

Suponemos lo siguiente sobre la arquitectura, la memoria y el acceso a datos en la memoria:

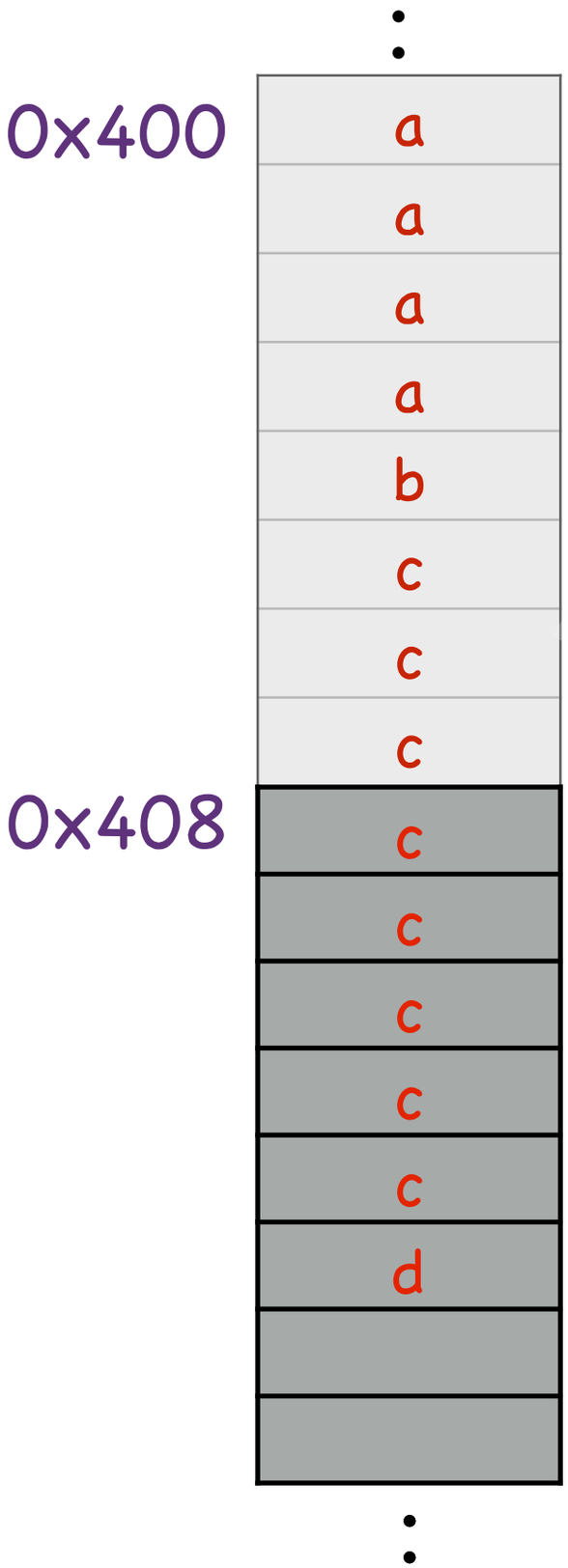
- La memoria es una secuencia de palabras (WORDs)
- El tamaño de una palabra puede ser 2, 4, o 8 bytes depende de la arquitectura (en arquitecturas de 32bit es 4 y en arquitecturas de 64 bits es 8).
- Para traer datos desde la memoria al CPU/CACHE ,p.ej. acceder al valor de una variable, normalmente traemos palabras completas (por razones de eficiencia/diseño).
- Durante la ejecución de un programa, para acceder a datos que ocupan N bytes (el valor de una variable) a partir de la dirección de memoria M, tenemos que traer todas las palabras que incluyen las direcciones desde N hasta N+M-1.

Ejemplo (y problema) de Acceso



- 0x400 y 0x408 son las direcciones de dos palabras en la memoria.
- El valor de una variable x de tipo int está en la dirección 0x406 con valor 0x3ED03C04 (1053834244 en decimal).
- El valor de x abarca 2 palabras.
- Para traer el valor de x necesitamos 2 accesos a la memoria, uno para traer la palabra 0x400 y otro para 0x408.
- Si el valor de x empezara en la dirección 0x400, 0x401, 0x402, 0x403, o 0x404 podríamos hacerlo con un sólo acceso a la palabra 0x400.

Ejemplo (y problema) de Acceso



Para structs esto puede ser incluso más complicado.

```
struct T {
    int a;
    char b;
    long c;
    char d;
};

T x;
```

Si tenemos unos structs seguidos en la memoria esto puede complicarse más, por ejemplo en este caso 'a' y 'c' abarcarán 2 palabras.

Type Alignment y Padding

- Para evitar este problema, el compilador tiene condiciones (type alignment) que indican en qué direcciones en la memoria se pueden almacenar valores de cada tipo, con el objetivo de minimizar el numero de accesos a la memoria.
- Si un miembro del struct no cumple esas condiciones, el compilador añade paddings para que las cumpla, que son "espacios" extra dentro del struct ...

```
struct T {
    int a;
    char b;
    long c;
    char d;
};
```

```
struct T {
    int a;
    char b;
    char pad_1[3]
    long c;
    char d;
    char pad_2[7]
};
```

padding

Type Alignment para Struct

Informalmente, las condiciones sobre direcciones que se pueden usar para cada tipo, y como hacer padding, son las siguientes:

1. alignof(T) es una función que ayuda a decidir si se puede almacenar un valor de tipo T en la dirección N. Para tipos primitivos (char, int, etc.) alignof(T) = sizeof(T). La condición que se tiene que cumplir es N % alignof(T) == 0, es decir N es divisible por alignof(T).

2. Antes de cada miembro (de tipo T o T[]) del struct, el compilador añade padding para que su dirección N (contando desde 0, el inicio del struct) cumpla la condición N % alignof(T) == 0.

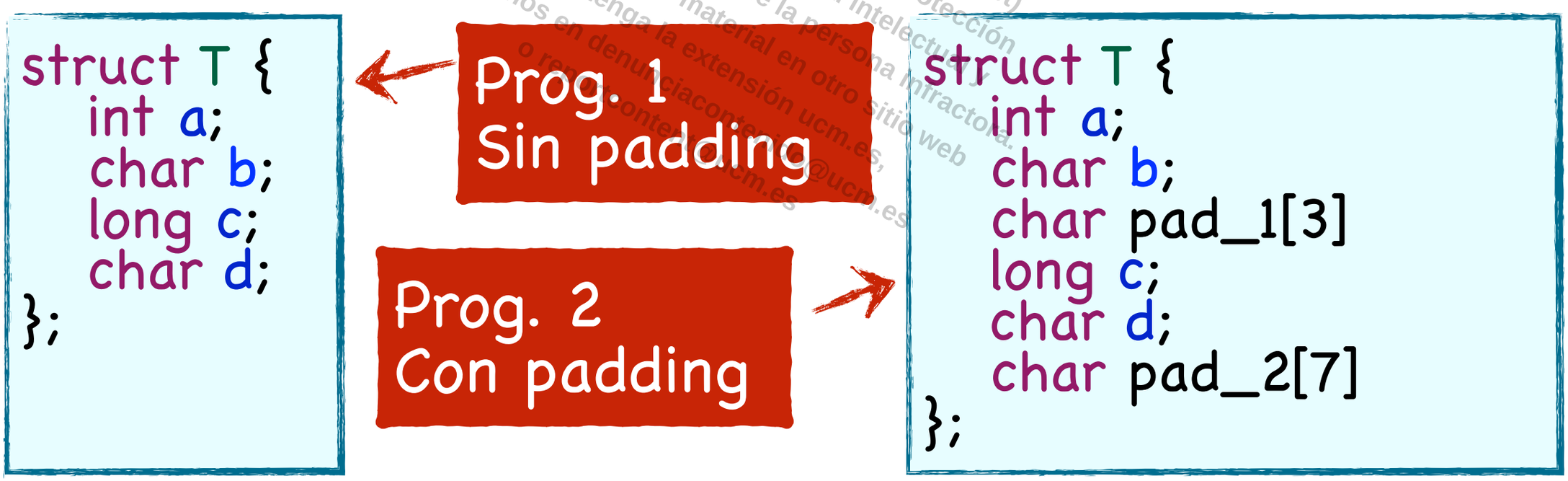
3. Al final de un struct T, el compilador añade padding para que su tamaño sizeof(T) sea divisible por el tamaño del miembro con tamaño máximo M, y define alignof(T)=M.

Además, cuando definimos una variable de tipo T (primitive, array, struct, etc), su dirección en la memoria es siempre divisible por alignof(T). !El mecanismo de padding puede ser distinto en compiladores distintos!

Problemas de Padding

Usando padding, el compilador genera código más eficiente, y en algunos caso mucho más eficiente, sólo por reducir el número de accesos a la memoria. Pero en algunos contextos esto puede causar otros problemas ...

Ejemplo: tenemos un programa y lo compilamos con 2 compiladores distintos, por ejemplo para Windows y para Linux, uno usa padding y el otro no los usa (o usa mecanismo distinto de padding). Nosotros no sabemos nada porque el padding es implícito ...



Problemas de Padding

```
struct T {
    int a;
    char b;
    long c;
    char d;
};
```

Prog. 1
Sin padding

Prog. 2
Con padding

```
struct T {
    int a;
    char b;
    char pad_1[3]
    long c;
    char d;
    char pad_2[7]
};
```

- El programa 1 tiene un struct x de tipo T y lo escribe en un archivo usando f.write(&x, sizeof(x)). En otra parte lea esa información a un struct z usando f.read(&z, sizeof(z)). Aquí todo es correcto porque sizeof(x) y sizeof(z) son iguales, usamos el mismo tipo T.
- Pero si el programa 2 intenta a leerlo tendremos un problema, porque sizeof(z) es distinto, y la información no corresponde al mismo struct aunque en principio estamos usando el mismo programa.
- Esto puede pasar también cuando usamos structs para intercambiar información en red ...

Packing (Desactivar Padding)

Una posible solución es desactivar el mecanismo de padding y usar packing, es decir pedir al compilador que deje los structs como están declarados …

```
#pragma pack(push,1)


struct T {
   int a;
   char b;
   long c;
   char d;
};


…

#pragma pack(pop)
```

Cambia el valor de 'pack' a 1, es decir pide al compilador que no haga padding desde este punto en adelante. El 'push' recuerda el valor actual de 'pack' (se puede quitar el 'push' si no queremos recordarlo).

Restaura el valor anterior de 'pack'. No hace falta si no usamos 'push' arriba.

Packing (Otra forma)

Otra forma es usar lo siguiente, pero al mejor no funciona en algún compilador como Visual Studio

```
struct __attribute__((__packed__)) T2 {
   int a;
   char b;
   long c;
   char d;
};
```

Más Información

Muy buena referencia sobre el tema:

The Lost Art of Structure Packing

http://www.catb.org/esr/structure-packing/

Resumen

- No desactives el mecanismo de padding, porque puede mejorar las prestaciones de tu programa.
- Recuerda que el mecanismo de padding puede ser distinto en compiladores distintos.
- En el caso que usas structs para intercambiar datos entre programas, desactivarlo sólo para esos structs y usarlos sólo para enviar y recibir datos pero no para almacenarlos en la memoria. Pero también hay que usar tipos de tamaño fijo (cono uint32_t, etc.) para que los 2 lo interpretan de igual manera. Pero ...
- ... la mejor forma para intercambiar datos es usar "Serialization", para transformar el contenido (de un struct) en una secuencia de bytes y vice versa.