

# Memory allocation

Luca Tornatore - I.N.A.F. 

**“Foundation of HPC” course**



**DATA SCIENCE &  
SCIENTIFIC COMPUTING**

**2021-2022 @ Università di Trieste**

# Outline



Memory  
Padding



Overhead of  
Memory Allocation

# Memory allocation has a memory cost

```
typedef struct
{
    char    char_field;
    double  double_field;
    int     int_field;
    char    char_field2;
} STRUCT_A;
```

```
typedef struct
{
    char    char_field;
    double  double_field;
    char    char_field2;
    int     int_field;
} STRUCT_B;
```

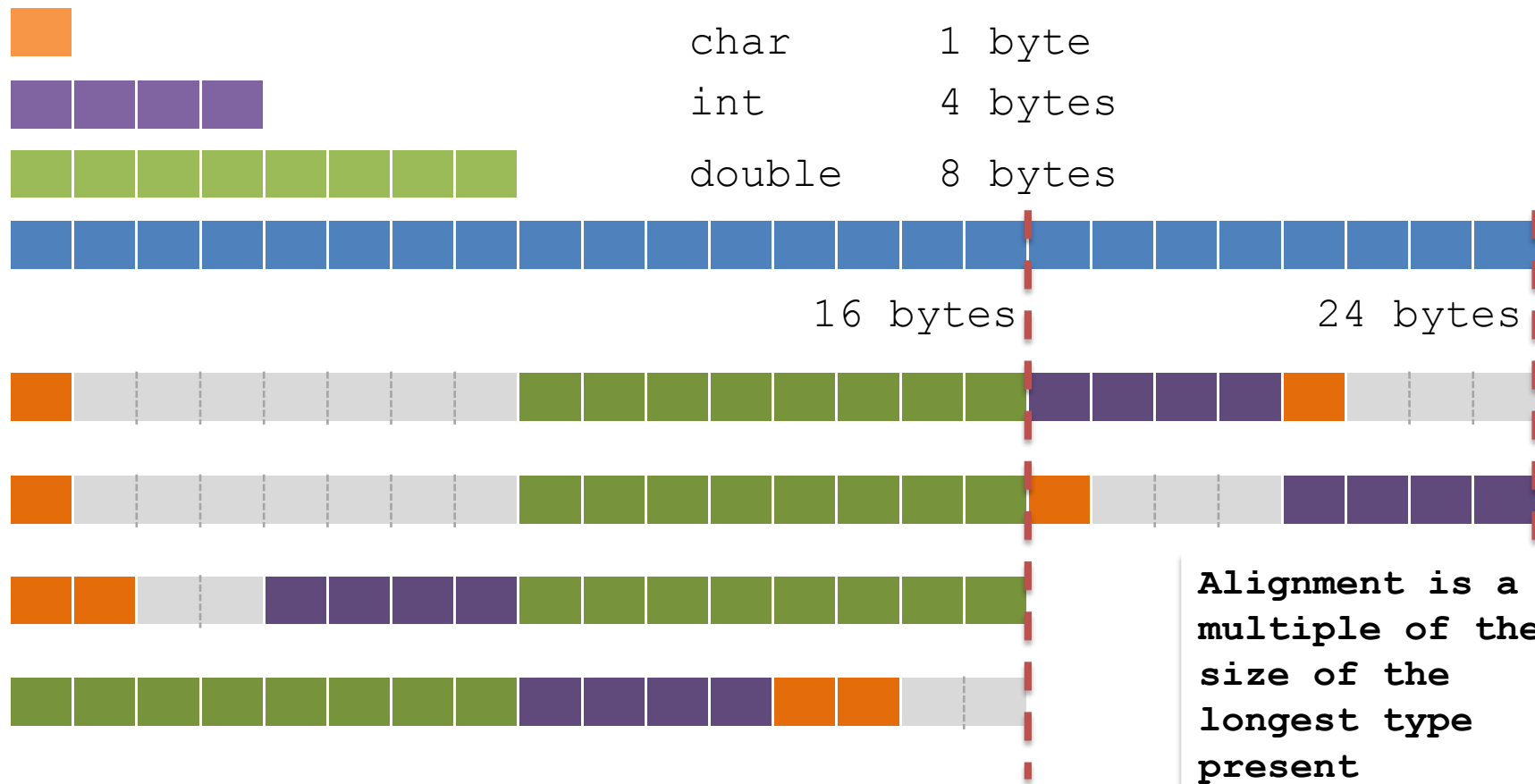
```
typedef struct
{
    char    char_field;
    char    char_field2;
    int     int_field;
    double  double_field;
} STRUCT_C;
```

```
typedef struct
{
    double  double_field;
    int     int_field;
    char    char_field;
    char    char_field2;
} STRUCT_D;
```

Is there any difference among the above C structures ?

...just have a look at the live results..

# Memory allocation has a memory cost : padding



Alignment is a multiple of the size of the longest type present

# Memory allocation has a memory cost : padding

```
gcc ... -fpack_struct[=n]
```

An instruction to pack them all and in the bitfield chain them down.

**To be used carefully:** it generates code *binary-incompatible* with code generated without the option (offset are different) and *sub-optimal code*.

Normally used for non-default binary interface (reduces the data stream).

If given,  $n$  must be a (small) power of two.

```
__attribute__ ( (packed) ) ;
```

Inline in the structure definition, at each field you want not to waste any byte.

The same words of caution than above hold.

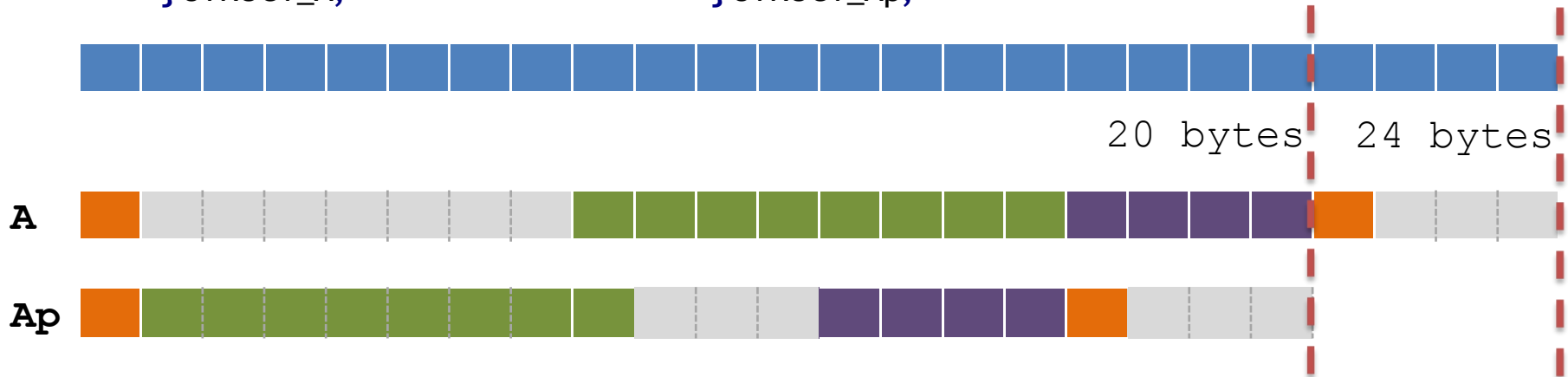
```
typedef struct
{
    char   char_field;
    double double_field;
    int    int_field;
    char   char_field2;
} STRUCT_A;
```

```
typedef struct
{
    char   char_field;
    double double_field __attribute__ ((packed));
    int    int_field;
    char   char_field2;
} STRUCT_Ap;
```

# Memory allocation has a memory cost : padding

```
typedef struct
{
    char   char_field;
    double double_field;
    int     int_field;
    char    char_field2;
} STRUCT_A;
```

```
typedef struct
{
    char   char_field;
    double double_field __attribute__((packed));
    int     int_field;
    char    char_field2;
} STRUCT_Ap;
```

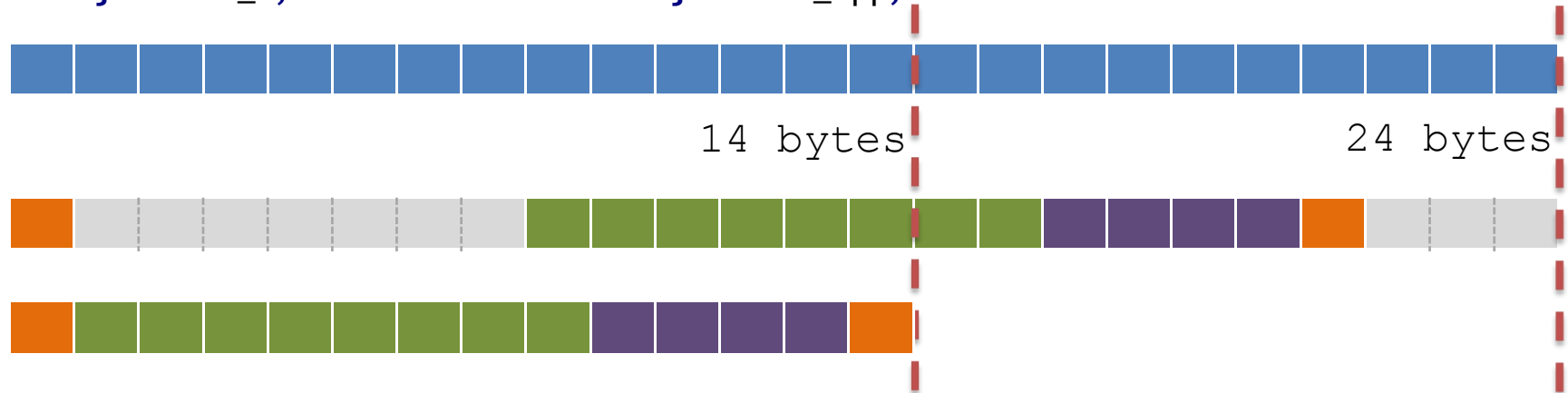




# Memory allocation has a memory cost : padding

```
typedef struct
{
    char   char_field;
    double double_field;
    int     int_field;
    char    char_field2;
} STRUCT_A;
```

```
typedef struct
{
    char   char_field;
    double double_field __attribute__((packed));
    int     int_field __attribute__((packed));
    char    char_field2;
} STRUCT_App;
```



...just have a look at the live results..

# Memory allocation has a memory cost : padding

General remark:

in order to have **best memory access performances**, it is usually better to have **data aligned** to the natural alignment of your machine, which typically is 32 or 64 bits.

```
__attribute__ ((aligned (n))) ;
```

You may achieve this in your data structures by using



# Memory allocation has a memory cost : padding

System's `malloc()` allocator returns memory addresses multiples of 8 or 16 for 32- and 64-bits systems.

If you different needs (for instance: using AVXN instructions), you can use special calls:

```
void * posix_memalign (void **memptr, size_t alignment, size_t size)
```

```
void * aligned_alloc (size_t alignment, size_t size)
```

*introduced in ISO C11 and hence may have better portability to non-POSIX systems*

# Outline



## Overhead of Memory Allocation

# Memory allocation has a memory cost

Everytime you allocate some memory chunk on the heap, the system has to **keep track** of it, in order to

- maintain a precise account of used and available memory
- be able to “register back” that chunk into the free memory when you don’t need it anymore (you `free()` it).

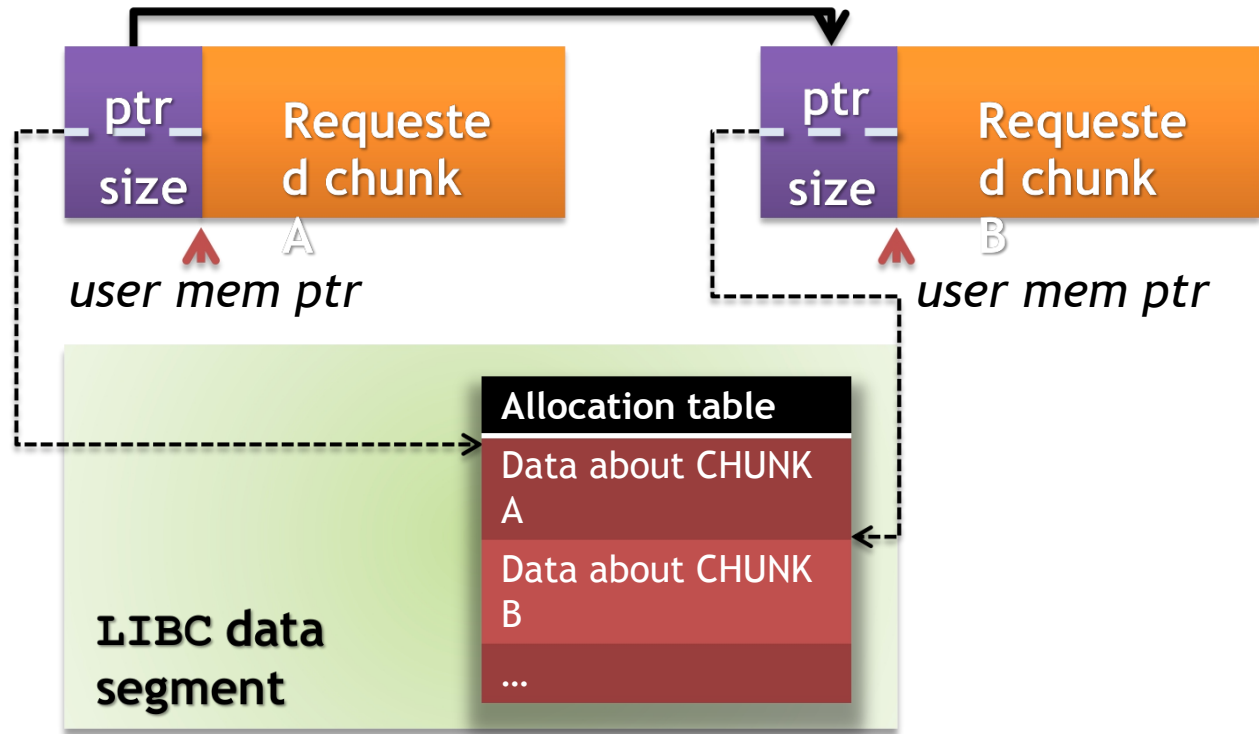
So, in addition to allocate *padded* memory layouts, which amounts to consume more memory than you formally request, there is also an additional “**administrative**” **overhead** whose amount is strongly system- and implementation-dependent (normally few bytes).

# Memory allocation has a memory cost

LIBC prepends an header to the chunk of data it allocates upon your request.

In that header, whose dimension is implementation-dependent, there are several data it uses to check the consistency of memory chain and to recover the data about that same allocation.

Every detail is strongly  
implementation-dependent !



# Memory allocation has a memory cost

Allocating a huge amount of small data (like in text-book linked-list techniques) might be not the optimal strategy:

- you incur in some waste of memory due to padding, depending on the layout of data;
- for sure in a larger amount of data needed to trace every chunk you require.

How much large is the latter overhead ?

Again, that is implementation-dependent.

Let's measure it on your computer and for different implementation  
(i.e. *different compilers and libc*)

**...just have a look at the live results..**

| # of allocated blocks | Requested Mem | Additionally located | Ptr Overhead | System overhead |
|-----------------------|---------------|----------------------|--------------|-----------------|
| 10M                   | 400 MB        | 24 %                 | 80 MB        | 18 %            |
| 1M                    | 400 MB        | 1.3 %                | 8 MB         | 3.2 %           |
| 100k                  | 400 MB        | 0.09 %               | 0.8 MB       | 0.27 %          |
| 10k                   | 400 MB        | 0.0 %                | 80 K         | 0.05%           |
| 1k                    | 400 MB        | 0.12 %               | 8 K          | 0.125 %         |
| 100                   | 400 MB        | 0.033 %              | 0.8 K        | 0.034 %         |
| 10                    | 400 MB        | 0.006 %              | 0.1 K        | 0.006 %         |