

# Computer Architecture (Practical Class)

## Heterogeneous Data Structures

Luís Nogueira

Departamento de Engenharia Informática  
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2023/2024

In C, we have two ways of combining variables of different types:

- *structures*, declared using the keyword `struct`, aggregate multiple variables under the same name;
- *unions*, declared using the keyword `union`, allow a single variable to use several data types.

Today, we will discuss *structures in C and Assembly*.

- A *structure* is:
  - A continuous memory region (similarly to arrays), composed of different *members*
  - Members are accessed by their name and can have different types
- We can have structures in C by:
  - defining a single structure in a variable;
  - declaring a structure type;
  - declaring a new data type.

Here, we define a *single structure* in a variable called "c1".

#### Structure in C - Defining a structure held by a variable

```
#include <stdio.h>

/* define a structure in a variable c1 */
struct{
    unsigned char n_wheels;
    unsigned int n_passengers;
    float fuel_consumption;
}c1;

int main(){
    /* init and access the data in c1 */
    c1.n_wheels = 4;
    c1.n_passengers = 2;
    c1.fuel_consumption = 12.5;
    printf("The car has %hd wheels, %d passengers.
        Consumption is %f 1/100 km.\n",
        c1.n_wheels,c1.n_passengers,c1.fuel_consumption);
    return 0;
}
```

- The structure has only one instantiation (in c1); no other instantiations of the structure exist.



Here, we declare a *structure type* called "s\_car".

#### Structure in C - Declaring a structure type

```
#include <stdio.h>

/* declare a structure type called s_car */
struct s_car{
    unsigned char n_wheels;
    unsigned int n_passengers;
    float fuel_consumption;
};

int main(){
    /* define two structures of type s_car */
    struct s_car c1, c2;

    /* init and access the data in c1 and c2 */
    c1.n_wheels = 4; c1.n_passengers = 2; c1.fuel_consumption = 12.5;
    c2.n_wheels = 5; c2.n_passengers = 4; c2.fuel_consumption = 20.0;
    printf("Car 2 has %hd wheels, %d passengers.
        Consumption is %f l/100 km.\n",
        c2.n_wheels,c2.n_passengers,c2.fuel_consumption);
    return 0;
}
```

- To instantiate the *s\_car structure*, we define new variables using the name of the structure after the keyword *struct*.

Here, we declare a new *data type* called "t\_car" using the keyword "typedef".

#### Structure in C - Declaring a data type for a structure

```
#include <stdio.h>

/* declare a data type called t_car */
typedef struct{ /* we can also write 'typedef struct s_car' */
    unsigned char n_wheels;
    unsigned int   n_passengers;
    float          fuel_consumption;
}t_car;      /* this is the name of the structure data type */

int main(){
    /* define two cars */
    t_car c1, c2;

    /* init and access the data in c1 and c2 */
    c1.n_wheels = 4; c1.n_passengers = 2; c1.fuel_consumption = 12.5;
    c2.n_wheels = 5; c2.n_passengers = 4; c2.fuel_consumption = 20.0;
    printf("Car 2 has %d wheels, %d passengers.\n",
           c2.n_wheels,c2.n_passengers,c2.fuel_consumption);
    return 0;
}
```

- To instantiate the *t\_car data type*, we define new variables using the name of the data type.

- We can define pointers to structures

```
struct s_car *ptr; /* pointer to structure type */  
t_car *ptr; /* pointer to a structure data type */
```

- To access the fields of a structure referenced by a pointer, use ->

```
ptr->n_wheels  
ptr->n_passengers  
ptr->fuel_consumption
```

instead of:

```
(*ptr).n_wheels  
(*ptr).n_passengers  
(*ptr).fuel_consumption
```

### Notes:

- To obtain the address of a structure, use the & operator
- Structures are very similar to arrays (continuous block of memory), **except in this aspect** (must use & to get address)



## Structure in C - Obtaining a pointer to a structure

```
#include <stdio.h>

typedef struct {
    unsigned char n_wheels;
    unsigned int n_passengers;
    float fuel_consumption;
}t_car;

int main(){
    /* define a car */
    t_car c1;
    /* define a pointer to a car */
    t_car *car_ptr = &c1;

    /* init and access the data in c1 using the pointer */
    car_ptr->n_wheels = 4;
    car_ptr->n_passengers = 2;
    car_ptr->fuel_consumption=12.5;
    printf("Car 2 has %d wheels, %d passengers.
          Consumption is %f l/100 km.\n",
          car_ptr->n_wheels,car_ptr->n_passengers,car_ptr->fuel_consumption);
    return 0;
}
```



- Consider the following structure in C

**Example structure in C to store personal data**

```
struct s_person_data{ /* we could also have used typedef ... */  
    char name[80];      /* 80 bytes */  
    char address[120];  /* 120 bytes */  
    int age;           /* 4 bytes */  
};
```

- To store this structure, we need 204 bytes (80+120+4)

**Important note**

We will see some subtleties of structure sizes in a moment

- In Assembly, we refer to members within a structure by their *offset*
- The offset of each member of the structure refers to the displacement, in bytes, from the start of the structure
- To simplify this access, we can define constants for the offset of each member using the `.equ` directive

```
.equ DATA_SIZE, 204      # total size
.equ NAME_OFFSET, 0       # name is at the beginning of the structure
.equ ADDRESS_OFFSET, 80   # the address starts at byte 80
.equ AGE_OFFSET,200       # age starts at byte 200
```

**Function set\_age in C**

```
void set_age(struct  
s_person_data *pdata){  
/* age = 30 */  
pdata->age = 30;  
}
```

**Function set\_age in Assembly**

```
set_age:  
# *pdata on %rdi  
# age = 30  
movl $30, AGE_OFFSET(%rdi)  
ret
```

Example: Access to a structure member

| Structure Declaration                                                          | Access in C                                                                                                           | Access in Assembly                                                                    |
|--------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| <pre>struct rec{<br/>    int x;<br/>    int a[3];<br/>    int *p;<br/>};</pre> | <pre>void set_x(struct rec *r,<br/>           int val){<br/>    /* access by name */<br/>    r-&gt;x=val;<br/>}</pre> | <pre># *r in %rdi<br/># val in %esi<br/># access by offset<br/>movl %esi,(%rdi)</pre> |

Memory layout for the structure rec

| x       | a        | p         |
|---------|----------|-----------|
| 0 ... 3 | 4 ... 15 | 16 ... 23 |

- Note that the address of *x* (the first member of the structure) is equal to the address of *r* (the structure)

Example: Obtaining the address of a structure member

| Structure Declaration                                         | Access in C                                                                                                                   | Access in Assembly                                                                    |
|---------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| <pre>struct rec{     int x;     int a[3];     int *p; }</pre> | <pre>int* find_a(struct rec *r,              unsigned int i){     /* address of a[i] */     return &amp;(r-&gt;a[i]); }</pre> | <pre># *r in %rdi # i in %rsi # %rax = r + 4 + (4 * i) leaq 4(%rdi,%rsi,4),%rax</pre> |

Memory layout for the structure rec

| x       | a        | p         |
|---------|----------|-----------|
| 0 ... 3 | 4 ... 15 | 16 ... 23 |

- To access field *a*, we need to add the appropriate offset to the address of the structure (in this case 4 bytes). Then, we need to access the element *i* within *a*, by adding offset *i* \* 4

Example: Obtaining a pointer and changing a structure member

| Structure Declaration                                         | Access in C                                                                                                                                              | Access in Assembly                                                                                                                                    |
|---------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>struct rec{     int x;     int a[3];     int *p; }</pre> | <pre>void set_p(struct rec *r){     /* address of a[i],      * where i is the value      * of member x */     r-&gt;p = &amp;(r-&gt;a[r-&gt;x]); }</pre> | <pre># *r in %rdi # %rcx = r-&gt;x movslq (%rdi),%rcx # %rax = r + 4 + 4*(r-&gt;x) leaq 4(%rdi,%rcx,4),%rax # r-&gt;p = %rax movq %rax,16(%rdi)</pre> |

Memory layout for the structure rec

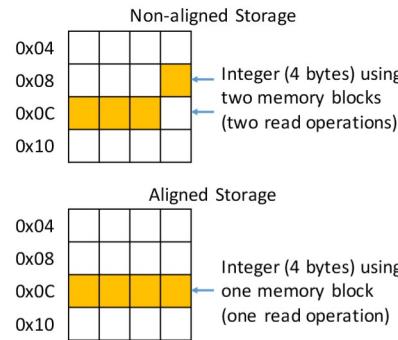
| x       | a        | p         |
|---------|----------|-----------|
| 0 ... 3 | 4 ... 15 | 16 ... 23 |

- Based on the size of each member (according to its data type), and the start address of the structure, we must compute the addresses

**Why**

- Memory is accessed in blocks of fixed size
  - Usually, 4 or 8 bytes, depending on the system
- Storing data at addresses multiple of the block size — *aligned to the block size*
  - will reduce the number of memory accesses

The following example depicts a system accessing memory in blocks of 4 bytes:



Data alignment is performed by the compiler

- The compiler inserts “spaces” in the data stored in memory to ensure that the members are aligned
- This improves the performance of the code
- This is required in some architectures; in x86-64 it is just recommended
  - in x86-64 data can be misaligned, at the cost of a performance penalty
- May be treated differently by different operating systems

The alignment rules are based on the principle that any primitive object of  $K$  bytes must have an address that is a multiple of  $K$

- $K = 1$  byte: `char`
  - No restrictions
- $K = 2$  bytes: `short`
  - The least significant bit must be 0 (that is, the address must be a multiple of 2)
- $K = 4$  bytes: `int, float, ...`
  - The 2 least significant bits must be 0 (that is, the address must be a multiple of 4)
- $K = 8$  bytes: `double, long, long long, char *, ...`
  - The 3 least significant bits must be 0 (that is, the address must be a multiple of 8)

### Important notes

These restrictions hold for most x86-64 operating systems, except that on Windows, the `long` type has size and alignment 4 (The `long long` type has size and alignment 8 on all x86-64 operating systems)

## Particularly important to know about data alignment when dealing with structures in C

- In order to be able to share structures between C and Assembly, we have to be aware of the data alignment made by the compiler

### Inside the structure

- We must satisfy the alignment requirements for each member of the structure
- Each structure has an alignment requirement of  $K$ , which may require implicit internal padding, depending on the previous member

### Placement of the structure in memory

- Given  $K$ , the largest alignment requirement inside the structure:
- The starting address of the structure must be a multiple of  $K$
- The total size of the structure must be a multiple of  $K$ , which may require padding after the last member (external padding)

Total size: 24 bytes

- $K=8$ , due to the member *c* of type long

```
struct S1{
    char a;
    int b[2];
    long c;
};
```

| a | 3 bytes | b        | 4 bytes | c         |
|---|---------|----------|---------|-----------|
| 0 | [gap]   | 4 ... 11 | [gap]   | 16 ... 23 |

Total size: 24 bytes

- $K=8$ , due to the member *b* of type long

```
struct S2{
    int a;
    long b;
    short c;
};
```

| a       | 4 bytes | b        | c        | 6 bytes |
|---------|---------|----------|----------|---------|
| 0 ... 3 | [gap]   | 8 ... 15 | 16 .. 17 | [gap]   |

### Important note

The starting address of both structures must be a multiple of  $K = 8$

Consider the following data type:

```
typedef struct{
    char age;
    short number;
    int grades[10];
    char name[80];
    char address[120];
}Student;
```

- Develop in Assembly the function `void save_grades(Student *s, int *new_grades, int size)` that copies all the elements of the array `new_grades` to the field `grades` of the structure pointed by `s`

**Function save\_grades in Assembly**

```
# void save_grades(Student *s, int *new_grades, int size)
save_grades:
    # *s in %rdi, *new_grades in %rsi, size in %rdx

    addq $4, %rdi          # rdi = &(s->grades[0])
    movslq %rdx, %rdx      # rdx = size
    cmpq $0, %rdx
    jle end

    movl $0, %rcx           # index = 0

loop_grades:
    cmpq %rcx, %rdx
    je end

    movl (%rsi, %rcx, 4), %eax  # place grade to copy on %eax
    movl %eax, (%rdi, %rcx, 4)  # copy grade in %eax to new_grades

    incq %rcx                # index++
    jmp loop_grades

end:
    ret
```