Computer Architecture (Practical Class)
Heterogeneous Data Structures

Luís Nogueira

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2022/2023

Heterogenous data structures in C

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.

In the example below, we define a *single structure* in a variable called "c1".

Structure in C - Defining a structure held by a variable

```c
#include <stdio.h>

/* define a structure in a global variable c1 */
struct {
  int   n_wheels;
  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 %d wheels, %d passengers.Consumption is %f l/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.

Defining structures in C (2/3)

In the example below, we declare a *structure type* called "s_car".

Structure in C - Declaring a structure type

```c
#include <stdio.h>

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

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

  /* init and acess 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. 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`.

Defining structures in C (3/3)

In the example below, we declare a new *data type* called "t_car" using the keyword "typedef".

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

```c
#include <stdio.h>

/* declare a data type called t_car */
typedef struct { /* we could write 'typedef struct s_car' */
   int    n_wheels;
   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. Consumption is %f l/100 km.\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.

Pointers to structures in C (1/2)

- We can define pointers to structures

```c
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 ->

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

instead of:

```c
(*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)

Pointers to structures in C (2/2)

Structure in C - Obtaining a pointer to a structure

```c
#include <stdio.h>

typedef struct {
    int   n_wheels;
    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("The car has %d wheels, %d passengers. Consum. is %f l/100 km.\n",
            car_ptr->n_wheels,car_ptr->n_passengers,car_ptr->fuel_consumption);

    return 0;
}
```
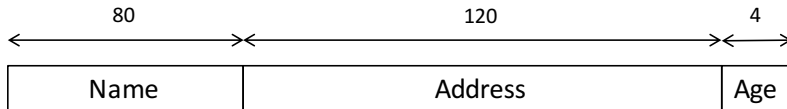
Storing structures in memory

- 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)

80	120	4
Name	Address	Age

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

```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

```
struct rec{
  int x;
  int a[3];
  int *p;
};
```

```
void set_x(struct rec *r,
              int val){
  /* access by name */
  r->x=val;
}
```

```
# *r in %rdi
# val in %esi

# access by offset
movl %esi,(%rdi)
```

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 struture)

Example: Obtaining the address of a structure member

Structure Declaration

```
struct rec{
 int x;
 int a[3];
 int *p;
}
```

Access in C

```
int* find_a(struct rec *r,
                unsigned int i){
  /* address of a[i] */
  return &(r->a[i]);
}
```

Access in Assembly

```
# *r in %rdi
# i in %esi

# %rax = r + 4 + (4 * i)
leaq 4(%rdi,%rsi,4),%rax
```

Memory layout for the structure `rec`

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

- To access field *a*, we need to add the 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

```
struct rec{
 int x;
 int a[3];
 int *p;
}
```

Access in C

```
void set_p(struct rec *r){
  /* address of a[i],
  where i is the value
  of member x */
  r->p = &(r->a[r->x]);
}
```

Access in Assembly

```
# *r in %rdi
# %rcx = r->x
movslq (%rdi),%rcx
# %rax = r + 4 + 4*(r->x)
leaq 4(%rdi,%rcx,4),%rax
# r->p = %rax
movq %rax,16(%rdi)
```
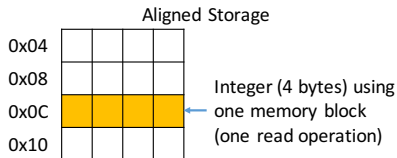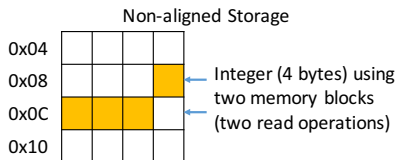
Memory layout for the structure `rec`

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

Data alignment

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:

Non-aligned Storage

0x04
0x08
0x0C
0x10

Integer (4 bytes) using
two memory blocks
(two read operations)

Aligned Storage

0x04
0x08
0x0C
0x10

Integer (4 bytes) using
one memory block
(one read operation)

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

Data alignment rules in x86-64

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

- $\mathbb{K} = 1$ byte: char
  - No restrictions

- $\mathbb{K} = 2$ bytes: short
  - The least significant bit must be 0 (that is, the address must be a multiple of 2)

- $\mathbb{K} = 4$ bytes: int, float, ...
  - The 2 least significant bits must be 0 (that is, the address must be a multiple of 4)

- $\mathbb{K} = 8$ 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 $\mathbb{K}$, which may require implicit internal padding, depending on the previous member

Placement of the structure in memory

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

Total size: 24 bytes

- $\mathbb{K}=8$, 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

- $\mathbb{K}=8$, 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 $\mathbb{K} = 8$

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

Practice problem

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 %edx

   addq $4, %rdi              # rdi = &(s->grades[0])
   movslq %edx, %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
```