Other Material

Pedro Ribeiro

DCC/FCUP

2024/2025

Assorted Material

- This set of slides covers very briefly a couple of aspects we will not have time to explore in more detail, namely:
  - C Libraries
  - Global Variables
  - Structs
  - Pointers

C Libraries

- The C language has provides many useful functions that you can call using an `#include` directive.
- You already used some (e.g. `stdio.h` with functions such as `printf` and `scanf`) but there are many other examples
- You can check a full list on the documentation.

  E.g.: cppreference: C Standard Library Headers
  - `stdlib.h` (see) - memory management, program utilities, string conversions, random numbers, algorithms
  - `stdio.h` (see) - input/output
  - `string.h` (see) - string handling
  - `ctype.h` (see) - determining character types
  - `math.h` (see) - mathematical functions
  - `limits.h` (see) - ranges of integer types
  - `float.h` (see) - ranges of floating point types
  - `time.h` (see) - time/data utilities
  - (...)

Global Variables

- In C variables exist only within their **scope**
  e.g. variables of a function only exist within that function
- If a variable is declared outside of all functions it is a **global variable** and can be accessed anywhere (use them with care...)

```c
#include <stdio.h>

int a; // global variable

void function() {
  a++; // a can be accessed here
}

int main(void) {
  a = 42;       // a can be accessed here
  function(); // an also inside the function
  printf("%d\n", a);

  return 0;
}
```

```
43
```

Structs

- A structure is a user-defined data type that can be used to group items of possibly different types into a single type.
- The `struct` keyword is used to define a structure.
- The items in the structure are called its **members**

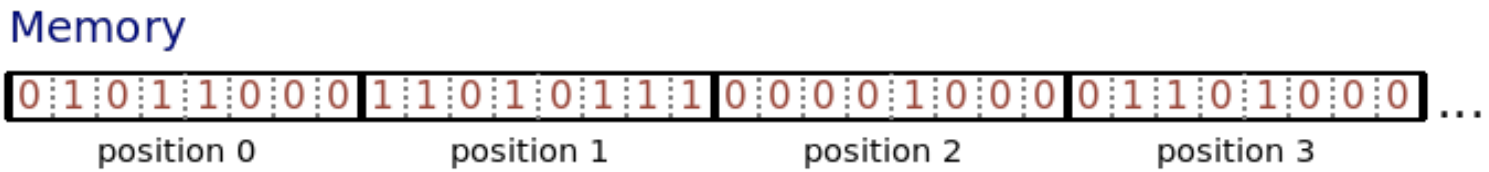
```
struct Point {
  int x, y;
  char name;
};
```

- To access an element we can use the `.` operator:

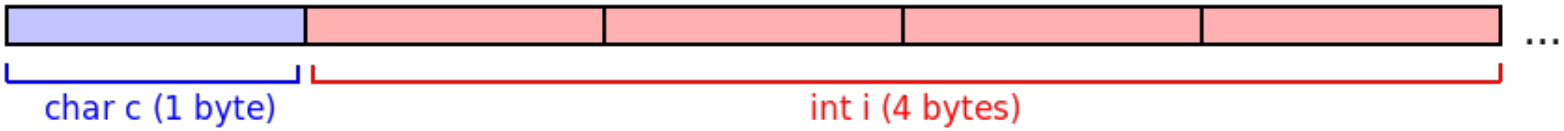
```
  struct Point p;

  p.x = 1;
  p.y = 42;
  p.name = 'A';
```

Pointers

- We can think of the memory in a computer almost as a "large array of bytes" (8 bits each):



- When you declare a variable you are "reserving" a space in memory to contain that variable (and telling how to interpret the bits):



- A **pointer** is a variable that stores a **memory address**

  Instead of holding a direct value, it holds the address (position) where the value is stored in memory.

Pointers - Operators

- There are 2 important **operators** to work with pointers:
  - `*` **Dereferencing operator** - used to declare a pointer variable and access the value stored in the address
  - `&` **Address operator** - used to returns the address of a variable

```c
int i = 42;

// pointer variable ptr with the address of variable i
int *ptr = &i;
// Value of variable m
printf("Variable i: %d\n", i);
// Memory address of variable m (hexadecimal format)
printf("Memory address of variable m is: %p\n", &i);
// Memory address using pointer
printf("Memory address using pointer ptr: %p\n", ptr);
// Content of i using pointer
printf("Value stored in i using pointer: %d\n", *ptr);
```

```
Variable i: 42
Memory address of variable m is: 0x7ffeaad7174c
Memory address using pointer ptr: 0x7ffeaad7174c
Value stored in i using pointer: 42
```

Pointers - Usage Example

- Image you want to create a function that actually changes the contents of the variables passed as arguments

  Incorrect version of a swap function

```c
#include <stdio.h>

void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  int a = 10;
  int b = 42;
  printf("Before swap | a=%d, b=%d\n", a, b);
  swap(a, b);
  printf(" After swap | a=%d, b=%d\n", a, b);

  return 0;
}
```

```
Before swap | a=10, b=42
 After swap | a=10, b=42
```

Pointers - Usage Example

- Image you want to create a function that actually changes the contents of the variables passed as arguments

  Correct version of a swap function

```c
#include <stdio.h>

void swap(int *a, int *b) {  // uses pointers
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int a = 10;
  int b = 42;
  printf("Before swap | a=%d, b=%d\n", a, b);
  swap(&a, &b); // we now pass the addresses
  printf(" After swap | a=%d, b=%d\n", a, b);

  return 0;
}
```

```
Before swap | a=10, b=42
 After swap | a=42, b=10
```

Pointers, scanf and arrays

- Now you know why you need to use `&` before a variable in `scanf`

  You are passing its **address** and letting `scanf` modify its contents:

```c
int n;
scanf("%d", &n);
```

- But why don't we need to pass the address when working with arrays?

```c
char str[100];
scanf("%s", str);
```

  Because a variable *pointing* to an array is already (for all practical effects) a pointer!

- the `[]` operator is in its essence *"syntactic sugar"*
  - `a+i` is equivalent to `&a[i]`
  - `a[i]` is equivalent to `*(a+i)`

Pointers and Arrays

- This is also the reason why when passing an array as argument you can change its contents on the function:

```c
#include <stdio.h>

void increment(int v[], int n) {
  for (int i=0; i<n; i++)
    v[i]++;
}

int main(void) {
  int v[] = {1,2,3,4};
  increment(v, 4);
  for (int i=0; i<4; i++) printf("%d ", v[i]);
  printf("\n");

  return 0;
}
```

```
2 3 4 5
```

Pointers and Arrays

- You can even use `int *v` instead of `int v[]`:

```c
#include <stdio.h>

void increment(int *v, int n) { // notice the usage of int *
  for (int i=0; i<n; i++)
    v[i]++;
}

int main(void) {
  int v[] = {1,2,3,4};
  increment(v, 4);
  for (int i=0; i<4; i++) printf("%d ", v[i]);
  printf("\n");

  return 0;
}
```

```
2 3 4 5
```

There is much more to know about pointers, but for today this is enough