

Informatics

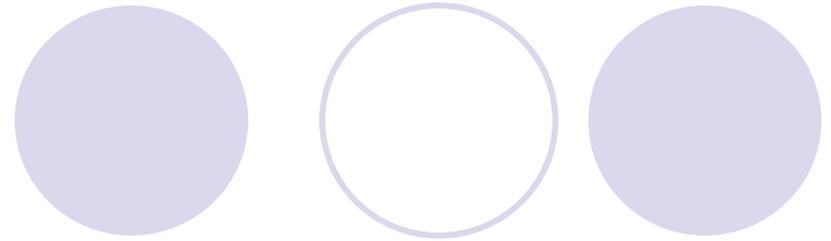
Ingeniería en Electrónica y Automática Industrial

Pointers

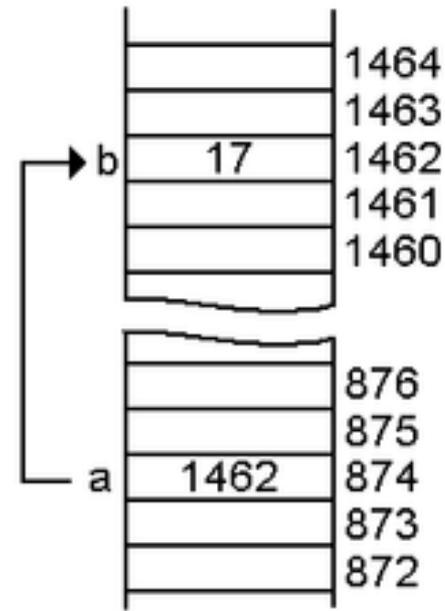
Pointers in C language

- Definition
- Pointers operators: «*» and «&»
- Declaration and initialization
- Operations with pointers
 - Assignment operations
 - Pointer arithmetic
- Pointers types
 - Generic
 - Null
 - Constant
- Arrays and pointers
 - Pointer to array
 - Arrays of pointers
- Pointer to pointer
- Pointers to structures and unions

Pointer definition (I)



- A **pointer** is a *variable* that contains the *memory address* of another *variable*
 - It is an *indirection*: the variable can be accessed *indirectly*
 - It is said that a pointer *points* to the variable
 - Example:
Pointer **a** *points* to variable **b**



From wikibooks.org

Pointer definition (II)

- Pointers are a very important tool in C language
 - They provide fast and efficient access to arrays
 - They facilitate working with linked lists
 - They facilitate information exchange between functions
 - They are essential to
 - Assign memory dynamically
 - Manage files
- Pointers must be used with a lot of care to avoid making serious mistakes very difficult to find

Pointers operators: «*» and «&»

- The **address operator** «&» returns the *memory address* of its operand
 - It can just be applied to variables and array elements

```
punt = &var;
```

- The **indirection operator** «*» applied to a pointer accesses the value of the variable the pointer points to
 - It can be used as any other variable without limitations

```
*punt = 7.98
```

- Both operators «*» and «&»
 - Are associated from left to right
 - Have higher precedence than arithmetic/logic operations

Pointers declaration and initialization (I)

- The declaration of a pointer variable assigns the necessary memory to store an address

```
datatype *pointername;
```

- `datatype` is the type of the variable to which the pointer points
 - `pointername` is the label of the memory position that stores the variable address
 - `*pointername` refers to the value of that variable
 - The declaration does not reserve any memory for the variable
- The memory size required to store an address is always the same, independently of the data type contained in the address

Pointers declaration and initialization (II)

- To **initialize** a pointer is to make it point to a valid variable
 - Variable must exist prior to pointer initialization
 - This does not mean that the variable must contain valid data

```
float *punt;      /* Pointer declaration */
float var;        /* Variable declaration. They
                  must be of the same type*/
punt = &var;     /* Pointer initialization. var
                  still without valid data*/
*punt = 7.98;    /* Variable initialization
                  Equivalent to var = 7.98; */
```

Operations with pointers (I)

- Just operations that can be made with addresses:
 - **Assignment** operations
 - **Arithmetic**: addition, subtraction, increment and decrement
- **Assignment** operations
 - Pointer to pointer:
 - Both will point to the same address
 - Both must be of the same type

```
int data, *punt1, *punt2; /* Declarations */  
punt1 = &data; /* punt1 initialization */  
punt2 = punt1; /* punt2 points to data*/
```

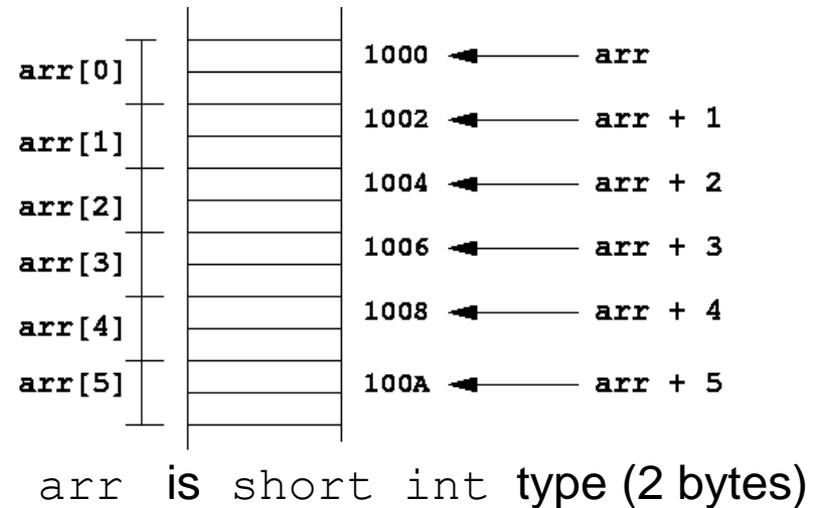
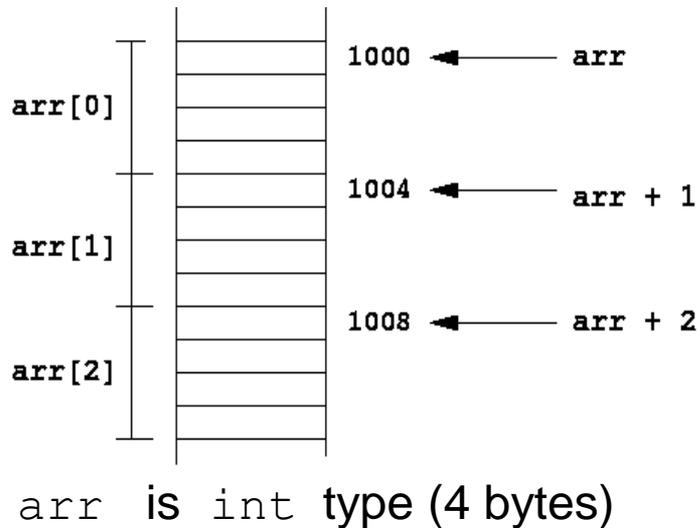
Operations with pointers (II)

- **Arithmetic operations:** Let `arr` be a pointer and `n` an integer

- **Addition, Subtraction, Increment/Decrement**

`arr+n`, `arr-n`, `arr++`, `arr--`

- **Pointer arithmetics just considers addresses**
(pointer arithmetic \neq ordinary arithmetic)



Pointer types

- **Generic pointer** does not point to any data type yet

```
void *pointername;
```

- It is declared generic and later can point to any kind of data
- **Null pointer** points to address NULL (= 0)

```
datatype *pointername = NULL;
```

- NULL is a constant defined in `stdio.h`
- It is used because address 0 is not valid
- **Constant pointer** always points to the same address

```
datatype *const pointername;
```

- The content of the address do may change though

Arrays and pointers (I)

- Every thing that can be done with arrays can also be done with pointers
 - Pointer versions are generally faster and more used
- The array identifier is a pointer to its first element
- To access element M in an array of N elements, $0 \leq M < N$
 - With arrays

```
elementM = arrayname[M];
```

- With pointer

```
elementM = *(arrayname+M);
```

Since the name of an array is a synonym of the location of the initial element

Arrays and pointers (II)

- A **pointer to an array of characters** points to the first element
 - It can be initialized in declaration

```
char *pointername = "string";
```

- `pointername` contains the address of the first character
 - `string` is a string of characters ending with `'\0'`
- Functions receive a string as a pointer to the first element of the string (pass by reference)

```
char *message = "Reading error";  
puts(message);
```

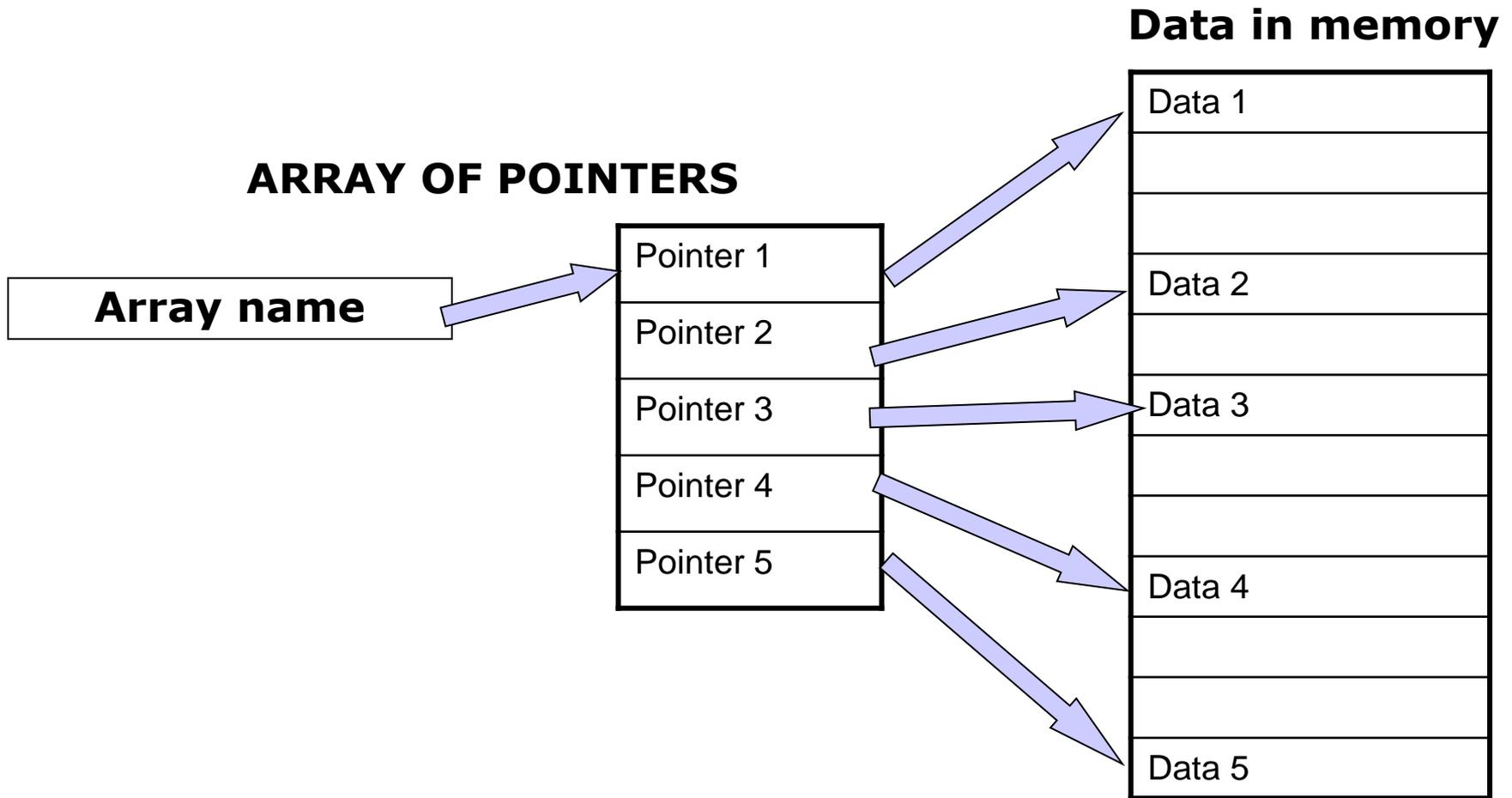
Arrays and pointers (III)

- An **array of pointers** is declared as

```
datatype *arrayname[size];
```

- Its elements are addresses where `datatype` elements are contained
 - All elements must be initialized pointing them to a valid data
- An array of pointers to character is similar to a array of strings

Arrays and pointers (IV)



Arrays and pointers (V)

- Examples:

- 2D array of characters

```
char mssg[3][80] ={"Initial", "Central", "Last"};
puts(mssg[1]);    /* "Initial" to screen */
```

- Array of pointers to character

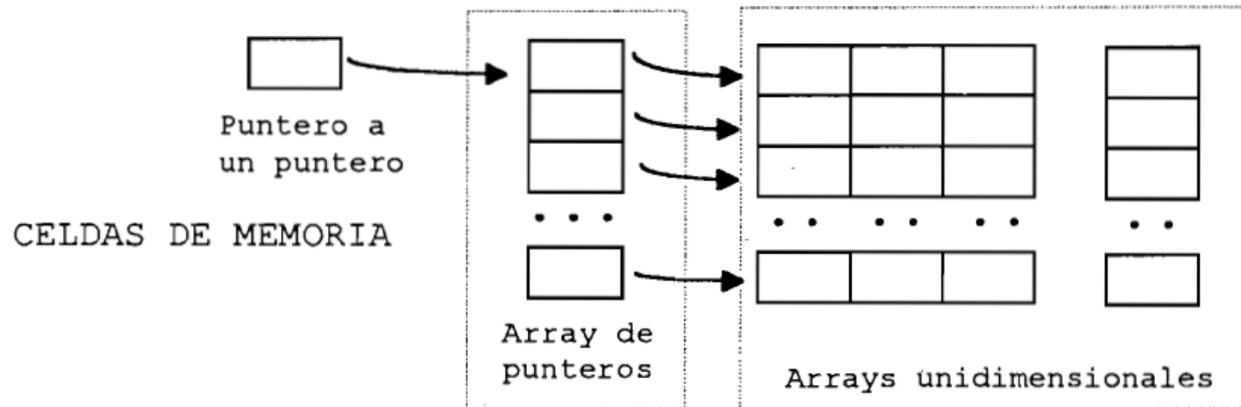
```
char *mssg [3]; /* Array of 3 pointers to char */
mssg[0]= "Initial"; /* Initialization*/
mssg[1]= "Central";
mssg[2]= "Last";
puts(mssg[1]);
```

Pointer to pointer

- A **pointer to pointer** is a double indirection:

```
datatype **pointername;
```

- `pointername` contains the address of `*pointername` whose contains the address of `**pointername`
- Particularly important in **dynamical memory allocation of multidimensional arrays** (unit 4.10)
- Element `matrix[i][j]` of 2D-array can be accessed `*(*(matriz+i)+j)`



**ARRAY BIDIMENSIONAL CREADO MEDIANTE
ASIGNACIÓN DINÁMICA DE MEMORIA**

Pointers to structures and unions

- Pointer to structure/union declaration (unit 4.9)

```
struct structtypename *pointername;
```

```
union uniontypename *pointername;
```

- The types must be previously defined
- To access one members using pointers
 - Usual notation: `*pointername.membername`
 - With «->» operator: `pointername -> membername`