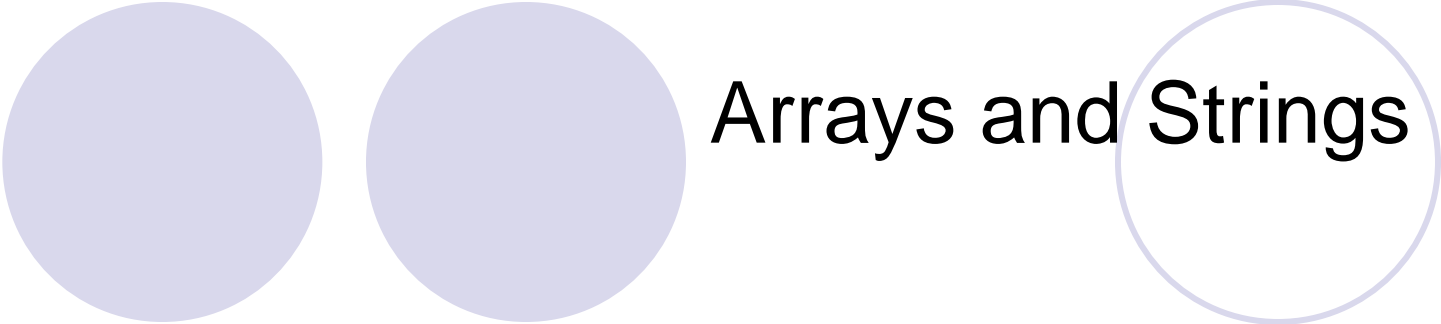




Informatics

Ingeniería en Electrónica y Automática Industrial



Arrays and Strings

Arrays and strings in C language

- Arrays
- Arrays declaration
 - Unidimensional arrays
 - Multidimensional arrays
- Arrays inicialization
- Strings
 - Functions to operate with strings
- Arrays of strings

Arrays



- An **array** is a special kind of variable that can store a set of data of the same type
- Each element can be referenced independently using the array identifier and an index between brackets « [] »
 - The first element has index 0.
 - The index can be represented by an integer expression
- Arrays can be
 - Unidimensionals
 - Multidimensionals
- Elements are stored in consecutive memory positions (first one in the lower address)
- The programmer must check the limits in size and dimensions of each array

Arrays declaration (I)

- Unidimensional array declaration:

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

- `datatype` for data type of the elements (any except `void`)
- `arrayname` is the array identifier
- `size` is an integer representing the number of elements, `n`
 - First element is `arrayname[0]` and last `arrayname[n-1]`
- The array declaration reserves memory for its elements
Reserved bytes = `size * sizeof(datatype)`

Arrays declaration (II)

- Multidimensional array declaration:

```
datatype arrayname[size1][size2]...[sizeN];
```

- `sizeX` are integer constant expressions

- `N+1` is the number of dimensions, so it is necessary to provide `N+1` indexes to determine one element of the array
- The value of each `size` fixes the size of each dimension

- Elements are stored in consecutive memory positions, and the total amount of memory the array occupies is:

Reserved bytes = `size1*size1*sizeN* sizeof(datatype)`

Arrays declaration (III)

- **Bidimensional arrays** (tables, 2D-matrices) are common and their dimensions are called **rows** and **columns**

```
datatype arrayname[numrows][numcolumns];
```

- `numrows` indicates the number of rows
- `numcolumns` indicates the number of columns

- **Examples:**

```
int list[10];           /* Array of 10 integers */
char vowels[5];        /* Array of 5 characters */
float matrix[6][4];    /* Matrix of 6 rows and 5
                        columns of real numbers*/
```

Arrays initialization (I)

- When an array is declared memory positions are assigned to it but their contents are not deleted, so it contains trash.
 - Except for external and static arrays that are automatically initialized to zero.

- To avoid this, arrays can be initialized in their declaration:

```
datatype arrayname[size1]...[sizeN]={valuelist};
```

- `valuelist` is a list of constants that initializes all elements

- In this case, the first dimension size can be omitted:

```
datatype arrayname[] = {valuelist};
```

```
datatype arrayname[][size2]..[sizeN]={valuelist};
```

Arrays initialization (II)

- When initializing an array, it is important to notice that:
 - The index that changes faster is the right one
 - Values must be given for all elements (completely initialized)

```
int digits[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int odds[] = {1, 3, 5, 7, 9};
int matrix[3][4] = {00, 01, 02, 03,
                   10, 11, 12, 13,
                   20, 21, 22, 23};
char letters[][5] = {'a', 'b', 'c', 'd', 'e',
                   'f', 'g', 'h', 'i', 'j',
                   'k', 'l', 'm', 'n', 'o',
                   'p', 'q', 'r', 's', 't',
                   'u', 'v', 'x', 'y', 'z'};
```


Arrays initialization (III)

- To initialize an array after its declaration, a loop must be programmed for each dimension.
- Example: to initialize a 2D-matrix:

```
int matrix[ROW][COL], f, c;
for(f=0 ; f<ROW ; f++)
    for(c=0 ; c<COL ; c++)
    {
        printf("Type element [%d][%d]", f, c);
        scanf("%d", &matrix[f][c]);
    }
```

Strings (I)



- A **character string** is a 1D-array with `char` elements

```
char stringname[stringlength];
```

- `stringname` is an identifier for the whole string
- `stringlength` indicates the length including « '\0' »
- « '\0' » (null) is always included as last character to indicate the end of the string
- Each character can be accessed individually as in any array

Strings (II)

- Initialization in declaration

```
char stringname[stringlength]= "mystring";  
char stringname[stringlength]= {charlist};
```

- `charlist` are individual characters between «' '», separated by commas, and including '`\0`' at the end.

```
char name[6]= "Peter";  
char name[]={ 'P', 'e', 't', 'e', 'r', '\0' };
```

- Strings are not a data type, so as with any array, initialization after declaration requires a loop.

Functions to operate with strings (I)

- There are many functions that facilitates operations with strings in the the libraries `stdio.h` `stdlib.h` `string.h`
- Some are:
- **scanf ()** To read characters from standard input (keyboard)
 - `scanf ("%[^\\n]s", string);` reads a string until return ('\\n')
 - `fflush(stdin)` cleans input buffer (recommended before `scanf`)
- **printf ()** with `%s` sends a string to the standard output (screen)
- **fgets ()** (`gets()` not recommended): reads a whole string substituting return by «\\0»
- **puts ()** writes a whole string (substituting «\\0» by return)

Functions to operate with strings (II)

- **strcat(string1, string2)** concatenates `string1` and `string2`
- **strcpy(string1, string2)** ; copies `string2` in `string1`
- **strcmp(string1, string2)** ; compares `string1` and `string2`
- **strlen(string)** ; returns `string` length
- **strlwr(string)** ; converts `string` characters to lower case
- **strupr(string)** ; converts `string` characters to upper case

- **atof(string)** ; converts a `string` to a double equivalent to the one represented by `string` (ex. '983' is converted to 983.0...)
- **atoi(string)** ; converts a `string` to an integer
- **atol(string)** ; converts a `string` to a long integer
- **fcvt()** ; converts a floating point number into a string of digits.

Arrays of strings

- An **array of strings** is a 2D-array in which the right index indicates the string number and the left one is the maximum length of the strings.

```
char arrayname[numstring][stringlength];
```

- **Example:**

```
char phrases[3][80]= {"Reading error",  
                      "Writing error",  
                      "Access error" };
```

```
// puts(phrases[0]); sends "Reading error" to screen  
/* stringlength=80 assures all sentences fit  
   (although probably wasting memory) */
```