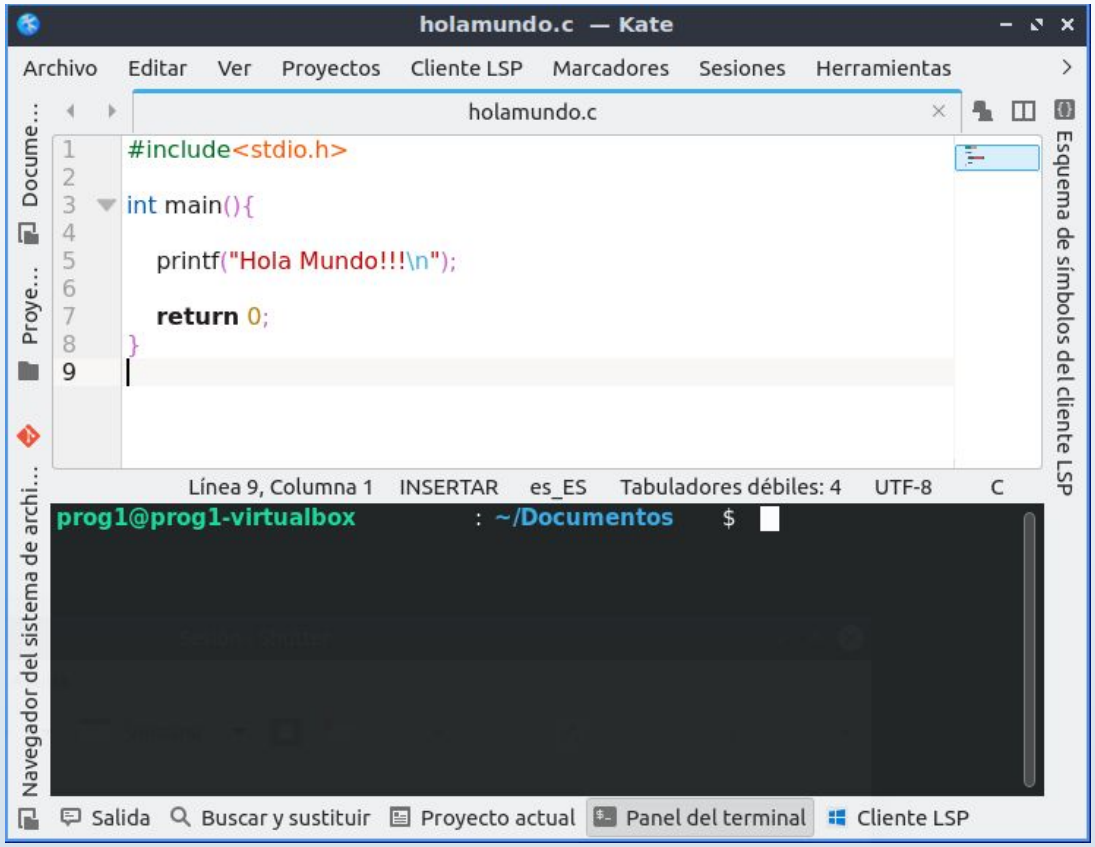



# Programming 1

## Lesson 6. Structured data types: Arrays

**Degree in Computer Engineering**

# Index

# 1. Structured data types



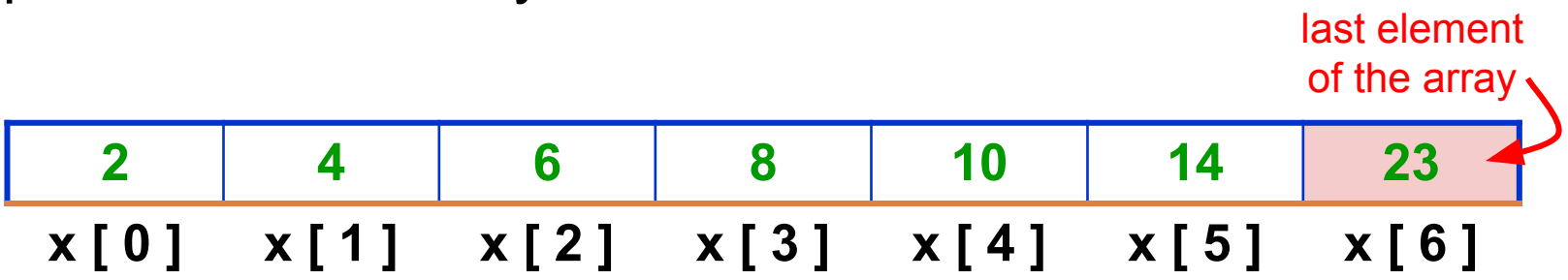
- A structured data type variable, unlike a simple data type, can store more than one value at a time.
- The structured data types in C that will be studied in this subject are the following:
  - **Array** type. All values stored in an array variable must be of the same data type.
  - **Structure** Type. A structure type variable can store values of different data types.
- Example: Let's consider a variable z that will store the winning numbers in the bonoloto. Therefore 6 values will be stored at a time:

```
z = (1, 4, 6, 24, 13, 2);
z = (3, 9, 12, 15, 23, 27);
```

# 2. The *array* type



- Structure in which a finite, homogeneous, ordered collection of data (elements) is stored.
  - **Finite**: the maximum number of elements that can be stored in the array must be specified.
  - **Homogeneous**: all the elements must be of the same type.
  - **Ordered**: it can be specified which is the n-th element of the array.
- To refer to a particular element in an array, an **index** (enclosed in square brackets [ ]) specifying its relative position in the array must be used.



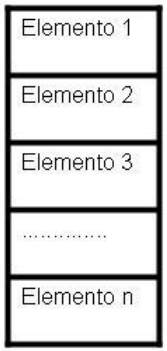

**Note**: In C language, the first element of the array is at **position** (index) **zero**.

# 2. The *array* type

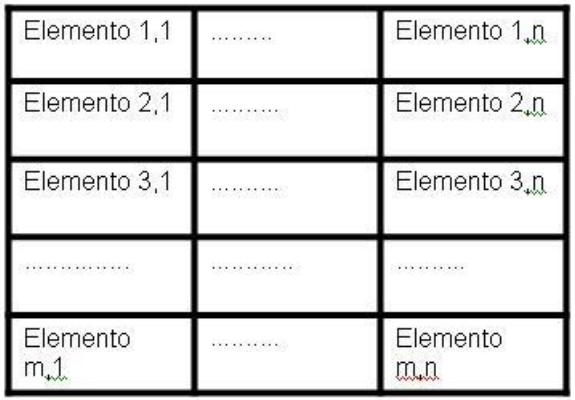


## Classification of arrays

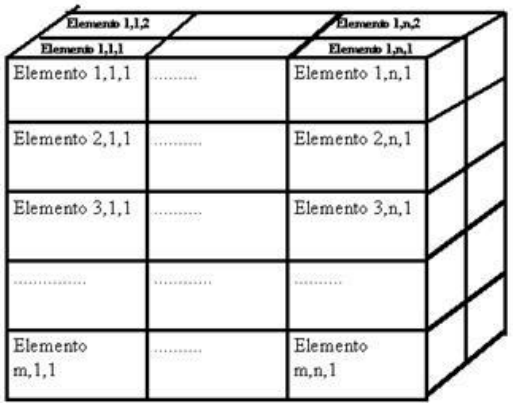
- Arrays are classified according to the number of dimensions:
  - **One-dimensional** array (vector)
  - **Two-dimensional** array (matrix)
  - **Multidimensional**, three or more dimensions
- The dimension of an array is the number of indices used to reference any one of its elements.




one-dimensional array




two-mensional array




multidimensional array

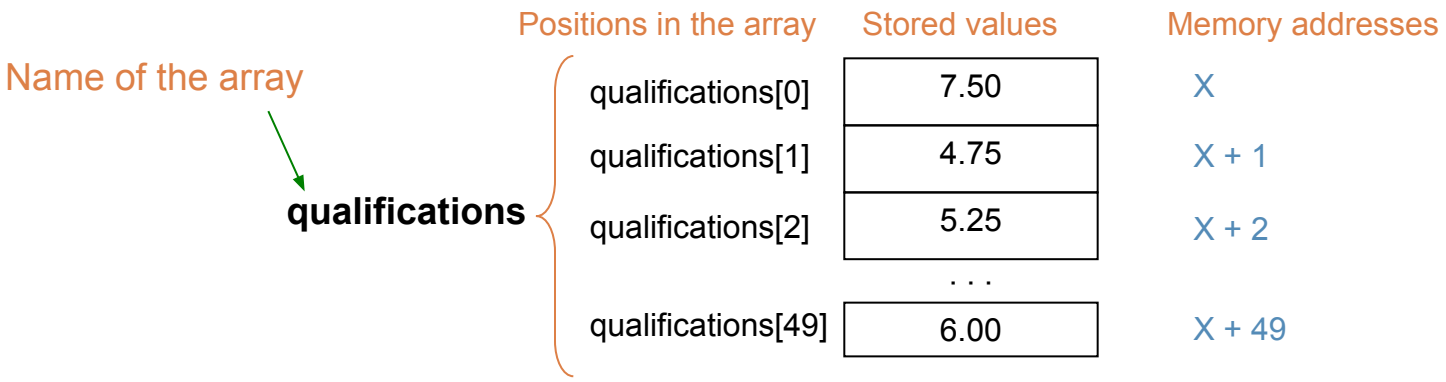
# 3. One-dimensional arrays



A one-dimensional array, also called a **vector**, is a series of data of the same type that is stored in contiguous memory locations, and can be accessed directly by a single index.

**Example**:
Suppose we want to store the mark of the Programming 1 exam of 50 students. Therefore we will need:

1. to reserve 50 memory locations ⇒ the easiest way to do this is to declare an array
2. to give the array a name
3. to associate a position in the array with each of the 50 students
4. to assign the marks to each of these positions



# 3. One-dimensional arrays



## Declaration

- In order to use an *array* variable (one-dimensional) we first have to declare it.
- Syntax:

```
type_of_data name_of_array [nr_elem] ;
```

  - `type_of_data`: indicates the the type of data of the elements in the array. All elements are of the same data type.
  - `name_of_array`: indicates the name of the array. It can be any valid identifier.
  - `num_elem`: indicates the **maximum number of elements** the array will be able to store. It must be an integer numeric constant value.
- Example: **float qualifications[**50**]**;

# 3. One-dimensional arrays



## Initialization and access to elements of an *array*

- As any other type of variable, before using an array we must **initialize** its contents.
- One possible way to initialize an array is by accessing each of its components using a loop and assigning them a value.
- To access an array position we use the following syntax:

```
name_of_array [index] ;
```

**Example**: Accessing the qualification of the student which is at the fifth position in the array: `qualifications[4];`

**Important**: Using index values **outside the range** of the array size may cause errors in the execution of our program.

# 3. One-dimensional arrays



**Example**: Initialization of an array

- If the values to store in the array are known when the array is declared, they can be assigned to the corresponding positions at the same time:

```
// example of array initialization
#include <stdio.h>

int main() {
    int vectorA [4]  = {1, 5, 3, 9};
    int vectorB []   = {1, 5, 3, 9};
    int vectorC [10] = {1, 5, 3, 9};

    return 0;
}
```

**vectorB**
It takes the number of values as the size of the vector

**vectorC**
It is possible to partially initialise the array. In this case, the remaining positions of the array, from 4 to 9, are initialized to 0.

# 3. One-dimensional arrays



## **Example (II)**: Initialization of an array

- Initialization of an array with the data the user enters by keyboard:

```
// example array initialisation
#include <stdio.h>

void inicialize_Array(float qual[]);

int main () {
  float qualifications[50];
  inicialize_Array(qualifications);
  return 0;
}

// procedure to initialize the array
void inicialize_Array(float qual[]) {
   int i;

   for ( i=0 ; i < 50 ; i++ ) {
      printf("Enter the qualification %d: ", i);
      scanf("%d", &(qual[i]) );
   }
}
```

**Note**: In C language, arrays are always passed **by reference** to modules and there is no need to be indicated in the module declaration.

# 3. One-dimensional arrays



## Linear search for an element in an *array*

We go through the array from the first position accessing consecutive positions until we find the searched element.

```
// Linear search for an element. Function to search for an "elem" element in an array with
// MAX_SIZE elements. Returns the position of "elem" in the array if found, or -1 if not
// found.
int lineal_Search(int array_name[], int elem) {
   int  pos;
   bool found;

   pos = 0;
   found = false;

   while ( pos < MAX_SIZE  &&  !found) { // search is finished if the end of the array is
     if (array_name[pos] == elem)        // reached or if the element is found
       found = true;
     else
       pos = pos + 1;
   }
   if (!found)
     pos = -1;

   return(pos);
}
```

## Binary search for an element in an *array*

If the array elements are SORTED we can use the **binary search (dichotomous)**: the search is reduced by dividing it in halves, so that the search interval is narrowed down depending on the value to be searched.

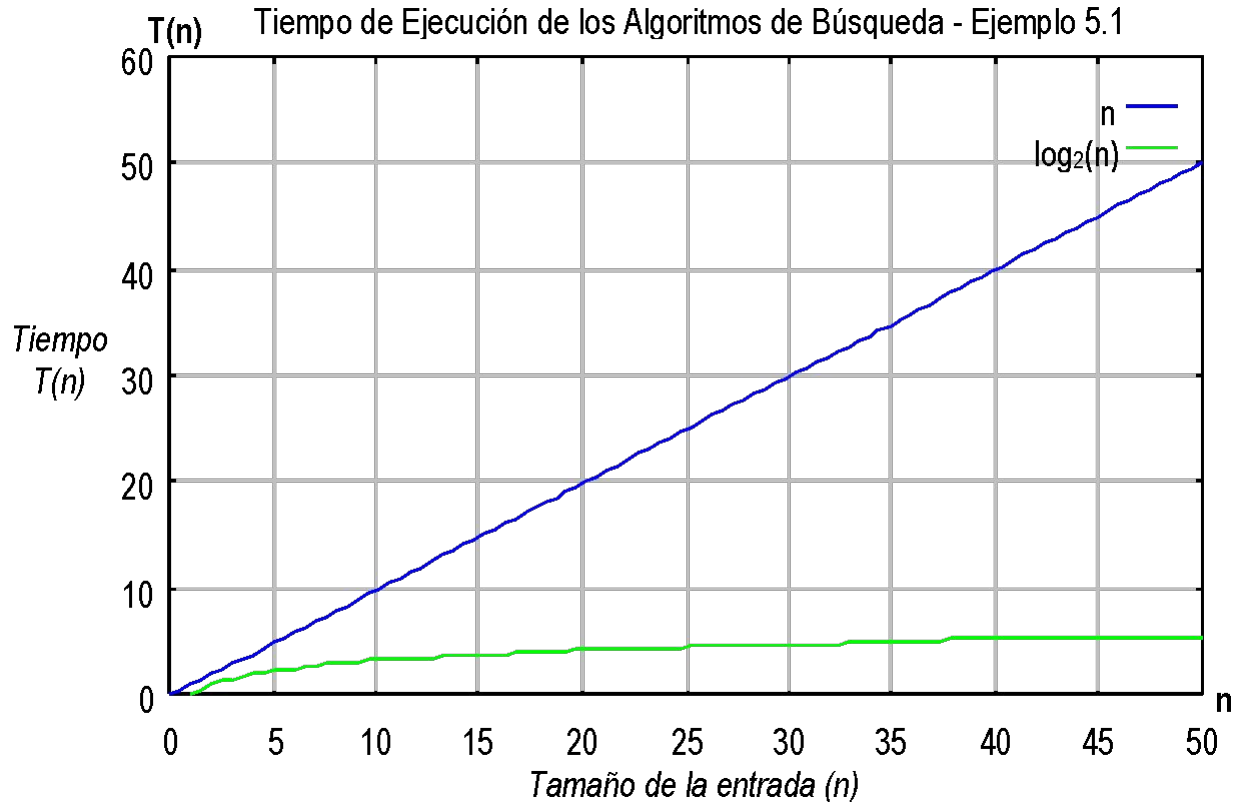
```
// Binary search in an array with MAX_SIZE elements sorted in ascending order
int binary_Search(int array_name[], int elem)  {
  int ini_pos, fin_pos, middle_pos; // [ini_pos, fin_pos] = current search interval
  bool found = false;

  ini_pos = 0;   // first position of the array
  fin_pos = MAX_SIZE - 1;   // last position of the array

  while ( (ini_pos <= fin_pos)  &&   !found) {
    middle_pos = (ini_pos + fin_pos) / 2;   // middle position of the array
    if (elem == array_name[middle_pos])  // element found at the middle_pos position
      found = true;
    else if (elem > array_name[middle_pos] )
      ini_pos = middle_pos + 1;  // the element is to be searched in the upper half
    else
      fin_pos = middle_pos - 1;  // the element is to be searched in the lower half
  }
  if (!found)
    middle_pos = -1;

  return middle_pos;
}
```

# 3. One-dimensional arrays



## Search (time cost)

- **Lineal** search: linear execution time
- **Binary** search: logarithmic execution time



# 3. One-dimensional arrays



## Character strings in C

- A character string, also called a ***string***, is a finite sequence of consecutive characters.
- To store character strings in C, ***arrays* of characters** are used:

```
char name_of_array[];
```

- Any alphanumeric text can be stored in a character array: words, phrases, names of persons, names of cities, alphanumeric codes, etc.

# 3. One-dimensional arrays

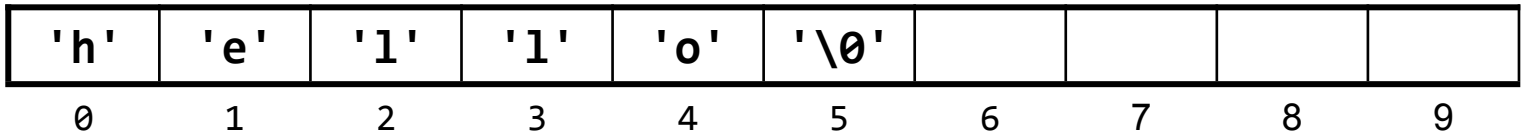


## Character strings in C

- In C language, a character string is written between double quotes:

```
"hello"
```

- In C language, all character strings must end with the null character **`'\0'`**, which must be stored in the array after the last character of the string:




- The string "hello":
  - has been stored in a character array of size 10.
  - is made up of 5 characters (length 5) but occupies the space of 6 characters in memory, because the character `'\0'` is also stored.

```
char cad[10]="hello";
```

# 3. One-dimensional arrays



## C functions for handling char *arrays*

| Function | Description | Use |
|---|---|---|
| **scanf(" %[^\n]s", &*string*)** | It reads a string of characters entered by keyboard until finding the newline character (\n). The sequence of characters read is stored in the variable *string* (array of characters) | As procedure |
| **Librería <string.h>** | | |
| **Function** | **Description** | **Use** |
| **strcpy(*target_string*, *source_string*)** | String copy. It copies the contents of *source_string* to *target_string* | As procedure |
| **strcat(*string1*, *string2*)** | String concatenation. It concatenates the content of *string2* to *string1* | As procedure |
| **strcmp(*string1*, *string2*)** | Alphabetical comparison of strings<br>if *string1* < *string2*<br>then it returns a number < 0<br>if *string1* == *string2*<br>then it returns 0<br>if *string1* > *string2*<br>then it returns a number > 0 | As function |
| **strlen(*string*)** | returns an *int* type indicating the length of the string passed in, i.e. the number of valid characters in the array (up to the special end-of-string character '\0', not included). | As function |

# 3. One-dimensional arrays



**Examples**:

- Procedure that prints the contents of an array of elements of type double on the screen.

```
// prints the elements of an array of type double on the screen
void print_Array(double a[], int len) {
  int i;
  for (i=0; i < len; i++)
    printf("[%d] = %f\n", i, a[i]);
}
```

- Function that calculates the average of the students' marks.

```
// we have "len" marks of type float
float calculate_Average(float a[], int len) {
  int   i;
  float sum;

  sum = 0.0;
  for (i=0; i < len; i++)
    sum = sum + a[i];

  return(sum / len); // supposing len > 0
}
```

# 3. One-dimensional arrays



**Examples (II)**:

Given an array of integers, shift all its elements one position to the right. The shifting will be circular, i.e. the last element will become the first element.

```
void circular_shifting(int v[]){
  int i, last;

  // store the value of the last position in the array
  last = v[MAX_LENGTH - 1];

  // shift all elements one position to the right,
  // except the last one
  for(i = MAX_LENGTH - 1; i > 0; i--)
    v[i] = v[i - 1];

  // store in the first position the value that was in the last position
  v[0] = last;
}
```

# 3. One-dimensional arrays



## Examples (III):

Given an array of integers, return the greatest value, the number of occurrences of that value, and the position of the first and the last occurrence.

```
void Ocurrences(int v[], int *greatest, int *num_ocur, int *pos_first, int *pos_last){
  int i;
  *greatest = v[0]; // initially, the largest number will be the number in the first position
  *num_ocur = 1;
  *pos_first = 0;
  *pos_last = 0;
  // go through the array: from the second position to the final position (constant MAX_LENGTH)
  for (i = 1; i < MAX_LENGTH; i++) {
    if (v[i] > *greatest) {  // a greatest number is found
      *greatest = v[i];
      *num_ocur = 1;
      *pos_first = i;
      *pos_last = i;
    }
    else if (v[i] == *greatest) { // a new occurrence of the greatest number is found
      *num_ocur = *num_ocur + 1;
      *pos_last = i;
    }
  }
}
```

## Array sorting algorithms

- The sort operation on an array is interesting and common.
  - Example: keep our vector of marks in order so that we can quickly search for the five best marks. To do this, we would have to sort our vector from highest to lowest (in decreasing order) and access the first five positions of the vector.
- There are many algorithms for sorting the elements of an array.

## Array sorting algorithms

- **Exchange** sorting

```
// v is the vector of elements and n is the
// number of elements in the vector
void bubble(int v[], int n) {
  int aux, i, j;

  for( i = 1; i < n; i++)
    for( j = n-1; j >= i; j--) {
      if(v[j-1] > v[j]) {
        aux = v[j-1];
        v[j-1] = v[j];
        v[j] = aux;
      }
    }
}
```

## Array sorting algorithms

- **Direct insertion** sorting

  This type of sorting can be compared to the sorting of a hand of cards. Each time we pick up a card we insert it into its correct position among the cards already in our hand.

  The insertion splits the array into two parts:

  - The sorted part. It represents the cards in our hand. It is sorted and increases in size as the sorting goes on.
  - The unsorted part. It represents the cards of the deck that we are adding. It is unsorted, and contains the elements that we are going to insert in the sorted part. This second part decreases in size as the sorting goes on.

## Array sorting algorithms

- **Direct insertion** sorting

```
// v is the vector of elements and n is the
// number of elements in the vector
void directInsertion(int v[], int n) {
  int aux, i, j;

  for( i = 1; i < n; i++) {
    aux = v[i];
    j = i - 1;
    while( j >= 0 && v[j] > aux) {
      v[j+1] = v[j];
      j--;
    }
    v[j+1] = aux;
  }
}
```

## Array sorting algorithms

- **Selection** sorting
  - **Step 1**: search and select from all the elements that are not yet sorted the lowest of them (if it is an increasing order).
  - **Step 2**: swap the positions of that element with the one on the leftmost position of the unordered part.

## Array sorting algorithms

- **Selection** sorting

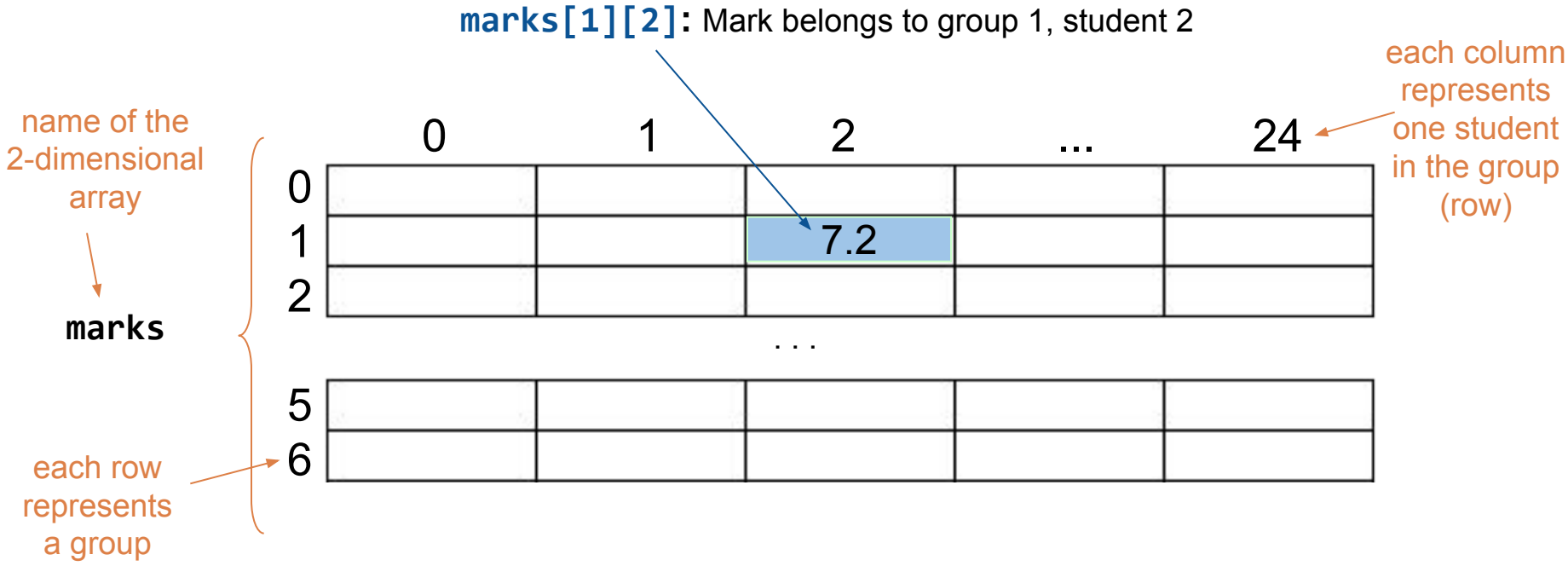
```
// v is the vector of elements and n is the
// number of elements in the vector
void selection(int v[], int n) {
  int aux, i, j, k;

  for( k = 0; k < n - 1; k++) {
    i = k;
    j = k + 1;
    while( j < n) {
      if(v[j] < v[i];
        i = j;
      j = j + 1;
    }
    aux = v[k];
    v[k] = v[i];
    v[i] = aux;
  }
}
```

# 4. Two-dimensional arrays



- They are also called **matrices**.
- To access any of its elements, **2 indexes** are needed.

**Example**: Suppose we want to store the exam marks of 7 groups of Programming 1, each of which has 25 students.



# 4. Two-dimensional arrays



## Declaration

- In order to use a two-dimensional *array* variable, we must first declare it.
- Syntax:

```
type name_of_array [n_rows] [n_columns] ;
```

  - `type`: data type of each element in the array; all elements in the array are of the same type.
  - `name_of_array`: name of the array
  - `n_rows`: number of rows in the array (first dimension)
  - `n_columns`: number of columns in the array (second dimension)

# 4. Two-dimensional arrays



## Initialization and access to a 2-dimensional array

- One possible way to initialize a two-dimensional array is to access each of its components using two loops (one for each dimension) and assign them a value.
- To access a position in a two-dimensional array we use the following syntax:

```
name [indexR] [indexC] ;
```

  - name: name of the array
  - indexR: position of the first dimension (**row**) of the array to be accessed; must be a value between 0 and number of rows - 1.
  - indexC: position of the second dimension (**column**) of the array to be accessed; it must be a value between 0 and number of columns - 1.

Examples: (From the previous example)

```
marks[6][24]; // access to the mark of student 24 of group 6
marks[6];     // access to all the marks in group 6
              // (one-dimensional array associated with row 6)
```

# 4. Two-dimensional arrays



## Initialization and access to a 2-dimensional array

- If the values are known, the array can be initialized as follows:

```
// example two-dimensional array initialization
#include <stdio.h>

#define N_ROWS 4;
#define N_COLS 2;

int main() {
  float matDD[N_ROWS][N_COLS] = { {3.6, 6.7},
                                  {2.9, 7.6},
                                  {8.9, 9.3},
                                  {1.9, 0.2}
                                };

  int mat[][N_COLS] = { {3, 6},
                        {9, 7},
                        {8, 3},
                        {1, 0}
                      };
  return 0;
}
```

**N_ROWS** y **N_COLS** are defined with the **#define** directive in order to be able to use them in the declaration of arrays. The C language does not allow constants (**const**) in the declaration of arrays.

In the C language, when declaring a multidimensional array, it is not mandatory to specify the size of the first dimension if it is initialized in the same declaration. The rest of the dimensions must be specified.

# 4. Two-dimensional arrays



## Initialization and access to a 2-dimensional array

- The two-dimensional array can also be initialized by using the data entered by keyboard:

```c
// example two-dimensional array initialisation
#include<stdio.h>

#define N_ROWS 2
#define N_COLUMNS 3

void inicialize(float matrix[][N_COLUMNS]);
void print(float matrix[][N_COLUMNS]);

int main () {
  float mat[N_ROWS][N_COLUMNS];

  inicialize(mat);
  print(mat);

  return 0;
}
```

In C language, it is not mandatory to specify the size of the first dimension of an array in the module declaration.

```c
// procedure for initializing the matrix
void inicialize(float matrix[][N_COLUMNS]){
  int i, j;

  for ( i = 0 ; i < N_ROWS ; i++ ) {
    printf("Row %d:\n", i);
    for ( j = 0; j < N_COLUMNS; j++) {
      printf("\tcolumn %d:", j);
      scanf("%f", &(matrix[i][j]));
    }
  }
}
// procedure for printing the matrix
void print(float matrix[][N_COLUMNS]) {
  int i, j;

  for ( i = 0 ; i < N_ROWS ; i++ ) {
    for ( j = 0; j < N_COLUMNS; j++) {
      printf("%5.2f ", matrix[i][j]);
    }
    printf("\n");
  }
}
```

# 4. Two-dimensional arrays



**Example I**:

Given 25 students, for whom the marks of 7 subjects are known, calculate the average mark of the subjects for each of the students and print them on the screen.

```
#include<stdio.h>
#define N_STUDENTS 25
#define N_SUBJECTS 7
void print_avg_mark_students(float marks[][N_SUBJECTS]);
int main() {
  float marks[N_STUDENTS][N_SUBJECTS];
  ...
  print_avg_mark_students(marks);
  ...
  return 0;
}
// calculates the average marks for each student and prints them out on the screen
void print_avg_mark_students(float marks[][N_SUBJECTS]) {
  int i;
  for (i = 0; i < N_STUDENTS; i++){
    printf("Average mark for student %d: %4.2f\n", i, calculate_Avg(marks[i], N_SUBJECTS));
  }
}
```

we use the function of one of the previous examples

# 4. Two-dimensional arrays



**Example II**:

Given a square matrix of integers print, in the following order, the elements of the diagonal, the elements of the upper triangle (above the diagonal) and the elements of the lower triangle (below the diagonal). Do all of this by going through rows and columns.

```
// Version that goes throughout the matrix three times
void print_Matrix_3(int matrix[][MAX_LEN]){
  int i, j;
  // Print diagonal
  for(i = 0; i < MAX_LEN; i++)  // throughout rows
    for(j = 0; j < MAX_LEN; j++) // throughout columns
      if (i == j)
        printf("%d", matrix[i][j]);
  // Print upper triangle
  for(i = 0; i < MAX_LEN; i++)
      for(j = 0; j < MAX_LEN; j++)
         if (j > i)
           printf("%d", matrix[i][j]);
  // Print lower triangle
  for(i = 0; i < MAX_LEN; i++)
    for(j = 0; j < MAX_LEN; j++)
      if (j < i)
        printf("%d", matrix[i][j]);
}
```

```
// Version that goes throughout the matrix
// only once
void print_Matrix_1(int matrix[][MAX_LEN]){
  int i, j;
  // Print diagonal
  for(i = 0; i < MAX_LEN; i++) // rows
    printf("%d", matriz[i][j]);
  // Print upper triangle
  for(i = 0; i < MAX_LEN - 1; i++)
    for(j = i + 1; j < MAX_LEN; j++)
      printf("%d", matrix[i][j]);
  // Print lower triangle
  for (i = 1; i < MAX_LEN; i++)
    for (j = 0; j < i; j++)
      printf("%d", matrix[i][j]);
}
```