PROGRAMMAZIONE PROCEDURALE

A.A. 2023/2024

ARRAYS

ARRAYS

- An array contains objects of a given type, stored consecutively in a continuous memory block. The individual objects are called the *elements* of an array. The elements' type can be any object type.
- An array's type is determined by the type and number of elements in the array. If an array's elements have type *T*, then the array is called an "array of *T*."
  - ✓If the elements have type **int**, for example, then the array's type is "**array of int**."
- The type is an incomplete type, however, unless it also specifies the number of elements.
  - ✓If an array of **int** has **16** elements, then it has a complete object type, which is "**array of 16 int elements**".

SUBSCRIPT [], ARRAY ELEMENTS

- The *subscript operator* [] allows you to access individual elements of an array.
- It takes two operands. In the simplest case, one operand is an array name and the other operand designates an integer.
  - ✓The expression myarray[i] then designates element number i in the array, where the first element is element number zero
- The left operand of [] need not be an array name.
- The right operand must have an integer type.

DEFINITION OF ARRAYS

- It is an aggregated type: a sequence of objects with the same type
- The definition of an array determines its **name**, the **type** of its elements, and the **number** of elements in the array. An array definition without any explicit initialization has the following syntax:

  ***type name*[*number_of_elements*];**

- The number of elements, between square brackets ([ ]), must be an **integer expression** whose value is greater than zero. An example:

  ✓char buffer[4*512];

- This line defines an array with the name buffer, which consists of 2,048 elements of type **char**.

  ✓**sizeof(buffer)** yields the value of **2048 * sizeof(char)**.

VARIABLE-LENGTH ARRAYS

- Since C99 it is possible to define an array using a non-constant expression for the number of elements
- Since C11 they are an optional features of compilers: some compilers might not implement VLAs.

```
void func( int n )
{
    int vla1[2*n];
    int vla2[n];
    /* ... */
}
```

ACCESSING ELEMENTS

- The expression myArray[i] designates the array element with the index i.
  - ✓Array elements are indexed beginning with 0.
  - ✓if **len** is the number of elements in an array, the last element of the array has the index len-1.

```
const int a= 4;
long mArray[a];
for ( int i = 0;  i < a;  ++i )
    mArray[i] = 2 * i;
```

mArray[0]	mArray[1]	mArray[2]	mArray[3]
0	2	4	6

INITIALIZING ARRAYS

- To initialize an array explicitly when you define it, you must use an *initialization list*: this is a comma-separated list of *initializers*
  - ✓int a[4] = { 1, 2, 4, 8 };
  - ✓a[0] = 1, a[1] = 2, a[2] = 4, a[3] = 8;

AN EXAMPLE

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

for (int i= 0; i < 20; i++) {
    printf(”Element[%d]= %d\n", i+1, a[i]);
}
```

```
Element [1]= 1
Element [2]= 2
Element [3]= 3
Element [4]= 4
Element [5]= 5
Element [6]= 6
Element [7]= 7
Element [8]= 8
Element [9]= 9
Element [10]= 10
Element [11]= 1339107492
Element [12]= 2097207605
Element [13]= 1477040992
Element [14]= 32767
Element [15]= -1796807251
Element [16]= 32767
Element [17]= 0
Element [18]= 0
Element [19]= 1
Element [20]= 0
```

SOME RULES FOR INITIALIZATION

- When you initialize an array, observe the following rules:
  - ✓You cannot include an initialization in the definition of a variable-length array.
  - ✓You may omit the length of the array in its definition if you supply an initialization list. The array's length is then determined by the index of the last array element for which the list contains an initializer
  - ✓If the definition of an array contains both a length specification and an initialization list, then the length is that specified by the expression between the square brackets. Any elements for which there is no initializer in the list are initialized to zero.
    - If the list contains more initializers than the array has elements, the superfluous initializers are simply ignored.
  - ✓Last comma in the initializers is ignored.

EXAMPLE

- As a result of these rules, all of the following definitions are equivalent:

```
int a[4] =          { 1, 2 };
int a[] =           { 1, 2, 0, 0 };
int a[] =           { 1, 2, 0, 0, };
int a[4] =          { 1, 2, 0, 0, 5 };
```

- In the final definition, the initializer 5 is ignored.
  - ✓Most compilers generate a warning when such a mismatch occurs.
- If array initializers have not the same type as the array elements, implicit type conversion.

INITIALIZING SPECIFIC ELEMENTS

- *Element designators* to allow you to associate initializers with specific elements. To specify a certain element to initialize, place its index in square brackets

  [constant_expression]

```
int a[6] = { 1, 2, [3] = 1, 2 };
```

int a[6]= {1, 2, 0, 1, 2, 0}

- This array definition initializes the elements a[0] and a[3] with the value 1, and the elements a[1] and a[4] with the value 2. All other elements of the array will be given the initial value 0.

EXAMPLE

```
int three= 2;
int a[4] = { 1, [three]=1 };
for (int i= 0; i < 4; i++)
        printf("Element [%d]= %d\n", i+1, a[i]);
```

```
test.c:45:20: error: expression is not an integer constant expression
      int a[4] =      { 1, [three]=1 };
                        ^~~~~
1 error generated.
```

```
const int three= 2;
int a[4] = { 1, [three]=1 };
for (int i= 0; i < 4; i++)
        printf("Element [%d]= %d\n", i+1, a[i]);
```

```
Element [1]= 1
Element [2]= 0
Element [3]= 1
Element [4]= 0
```

EXAMPLE

```
#include <stdio.h>

void fun(int a[3]) {
    printf("%lu\n", sizeof(a));
    for ( int i = 0; i < 3; ++i )
        a[i] = i;
}

int main(){
    int a[3];
    printf("%lu\n", sizeof(a));
    fun(a);
}
```

```
MacBook-Francesco:ProgrammI francescosantini$ gcc -o main main.c
main.c:4:29: warning: sizeof on array function parameter will
return size of
      'int *' instead of 'int [3]' [-Wsizeof-array-argument]
    printf("2 %lu\n", sizeof(a));
                            ^
main.c:3:14: note: declared here
void fun(int a[3]) {
             ^
1 warning generated.
MacBook-Francesco:ProgrammI francescosantini$ ./main
12
8
```

STRINGS

STRINGS

- A *string* is a continuous sequence of characters terminated by '\0', the null character. The length of a string is considered to be the number of characters excluding the terminating null character.
- Strings are stored in arrays whose elements have the type char
- You can initialize arrays of char using string literals. For example, the following two array definitions are equivalent:

```
char str1[30] = "Let's go";       // String length: 8; array length: 30.
char str1[30] = { 'L', 'e', 't', '\'', 's',' ', 'g', 'o', '\0' };
```

An array holding a string must always be at least one element longer than the string length to accommodate the terminating null character.

EXAMPLE

```
#include <stdio.h>
int main(){
    char string[5]= "Jim Morrison";
    printf("%s\n", string);
}
```

```
MacBook-Francesco:ProgrammI francescosantini$ gcc -o main
main.c
main.c:4:19: warning: initializer-string for char array is too
long
  char string[5]= "Jim Morrison";
                  ^~~~~~~~~~~~~~
1 warning generated.
MacBook-Francesco:ProgrammI francescosantini$ ./main
Jim M?0]?
```

EXAMPLE

```
#include <stdio.h>
int main(){
    char string[14]= {'J', 'i', 'm', ' ', 'M', '\0', 'o', 'r', 'r', 'i', 's', 'o', 'n', '\0'};
    printf("%s\n", string);
}
```

```
MacBook-Francesco:ProgrammI francescosantini$ gcc -o main main.c
MacBook-Francesco:ProgrammI francescosantini$ ./main
Jim M
```

MORE ON STRINGS AND LENGTH

- If you define a character array without an explicit length and initialize it with a string literal, the array created is one element longer than the string length.

```
char str2[] = " to London!";      // String length: 11
                                  // Array length: 12.
```

MULTIDIMENSIONAL ARRAYS

MULTIDIMENSIONAL ARRAYS

- A *multidimensional* array in C is merely an array whose elements are themselves arrays. The elements of an *n*-dimensional array are (*n*–1)-dimensional arrays.
- For example, each element of a two-dimensional array is a one-dimensional array (an array).
- A multidimensional array declaration has a pair of brackets for each dimension:
  - ✓char screen[10][40][80]; // A three-dimensional array.
  - ✓10 elements screen[0] to screen[9]. Each of these elements is a two-dimensional array, consisting in turn of 40 one-dimensional arrays of 80 characters each.
  - ✓ 32,000 elements.
- The standard recommends the implementations to accept at least 256 (ISO 14882, B.2), but they may support less or more

MATRICES

- Two-dimensional arrays are also called *matrices*.
  - ✓Frequently used, they merit a closer look.
- The matrix mat in the following definition has three rows and five columns (helpful to think in rows and cols):
  - ✓float mat[3][5];
- mat[0], mat[1], and mat[2] are the rows of the matrix mat. Each of these rows is an array of five float elements.
  - ✓15 float elements

	0	1	2	3	4
mat[0]	0.0	0.1	0.2	0.3	0.4
mat[1]	1.0	1.1	1.2	1.3	1.4
mat[2]	2.0	2.1	2.2	2.3	2.4

```
for ( int row = 0; row < 3; ++row )
    for ( int col = 0; col < 5; ++col )
        mat[row][col] = row + (float) col/10;
```

INITIALIZING MULTI-DIM ARRAYS

- You can initialize multidimensional arrays using an initialization list according to the rules described before

```
int a3d[2][2][3] = { { { 1, 0, 0 }, { 4, 0, 0 } }, { { 7, 8, 0 }, { 0, 0, 0 } } };
```

```
int a3d[][2][3] = { { { 1 }, { 4 } }, { { 7, 8 } } };
```

You do not need to specify that the first dimension has the size 2, as the outermost initialization list contains two initializers.

	0	1	2
a3d[0][0]	1	0	0
a3d[0][1]	4	0	0

	0	1	2
a3d[1][0]	7	8	0
a3d[1][1]	0	0	0

MORE ON INITIALIZATION

- You can also omit some of the braces. If a given pair of braces contains more initializers than the number of elements in the corresponding array dimension, then the excess initializers are associated with the next array element in the storage sequence:

```
int a3d[2][2][3] = { { 1, 0, 0, 4 }, { 7, 8 } };
int a3d[2][2][3] = { 1, 0, 0, 4, 0, 0, 7, 8 };
```

- You can also use element designators

```
int a3d[2][2][3] = { 1, [0][1][0]=4, [1][0][0]=7, 8 };
```

SU LIBRO

- Sezioni 6.1-6.5, 6.11
- Stringhe: Capitolo 8, Capitolo 9