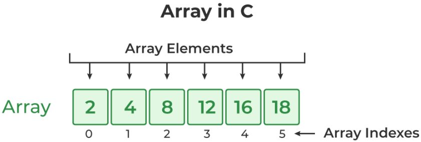
Arrays

Pedro Ribeiro

DCC/FCUP

2024/2025



(based and/or partially inspired by Pedro Vasconcelos's slides for Imperative Programming)

Arrays in C

- An **array** (also known as *indexed variable* or *vector*) is a fixed-size collection of data items of the same type stored in contiguous memory locations.
- It's *values* (*elements*) can be accessed using **indices**.
- Let's see an example of a declaration of an array with 1 dimension:

```
int a[4];
```

a[0]	a[1]	a[2]	a[3]

  - All the elements are of the same type (in this case `int`)
  - The index of the first element is `0`
  - The index of the last element is the *size(array) - 1*

Declaring an array

- We have to **declare** the array like any other variable before using it
- We can declare an array by specifying its **name**, the **type** of its elements, and the **size** of its dimensions:

  `type array_name[size];` for 1 dimension

  or

  `type array_name[size1][size2]...[sizeN];` for N dimensions

- The size of an array should be a positive integer constant:

```c
int a[4], b[5];
```

- We can use macros to improve the readability and facilitate program modifications:

```c
#define SIZE 10
...
int a[SIZE], b[SIZE+1];
```

Indices

- We access an element with `a[i]`
  - `i` is the (integer) *index*

- `a[i]` can be used just like a simple variable:
  - in an expression
  - on the left-hand side of an assignment
  - with increment operators; etc

```
a[i] = 1;
printf("%d", a[i]);
++a[i];
```

- We can use *expressions as indices* (as long as their value is integer):

```
a[i+j*10] = 0;
b[i++] = 42;
```

For loops for arrays

- **For loops** are very convenient for processing elements of arrays
- Some examples with an array of size **N**:

```c
for(i = 0; i < N; i++) // Put zeros on all positions
    a[i] = 0;
```

```c
for(i = 0; i < N; i++) // Read values
   scanf("%d", &a[i]);
```

```c
int soma = 0;
for(i = 0; i < N; i++) // sum elements
    soma += a[i];
```

Valid indices

- The C language **does not** check the indices of arrays
- This allows the implementation to be as efficient as possible
- But if the program accesses invalid indices, the *behaviour is undefined*:
  - it may terminate abruptly
  - it may not terminate
  - it may give wrong results
- Common error: accessing element `a[N]` of a variable with size `N`

Valid indices

- Can you spot the error on this code?

```c
int main(void) {
   int a[10];

   for(int i = 0; i <= 10; i++)
      a[i] = 0;

   ...
}
```

  - This program modifies `a[0], a[1], ..., a[10]`
  - But the valid elements are `a[0], a[1], ..., a[9]`
  - This can cause the loop not to finish or create errors on the remaining part of the program!

Array initialization

- Like simple variables, we can **initialize** arrays in the declaration.
- We can use a *list of values* in curly braces, separated by commas:

```
int a[5] = {0,2,4,6,8};
```

- If the list of values is shorter than the variable size, the remaining elements are filled with zeros:

```
int a[6] = {1,2,3}; // the initial values are {1,2,3,0,0,0}
```

- You can *omit the size* of the variable, in which case the compiler infers the size from the list of values:

```
int a[] = {0,2,4,6,8,10,12}; // Equivalent to declaring a[7]
```

- We can also use for loops to initialize:

```
int a[7];
for (int i=0; i<7; i++)
  a[i] = i*2;
```

Example

- Here is a full example of a program using arrays that you can download and test: `arrays.c` (source code)

```c
#include <stdio.h>

int main(void) {
  // Initialization without explicitly declaring size
  int a[] = {1,2,3,4,5};

  // changing value of a position
  a[2] = 42;

  // showing values of the array
  for (int i=0; i<5; i++)
    printf("a[%d] = %d\n", i, a[i]);

  return 0;
}
```

```
a[0] = 1
a[1] = 2
a[2] = 42
a[3] = 4
```

Functions and arrays

- We can pass arrays as *arguments* to a function
  - We just use its name, without the brackets

- A function **cannot** return an array
  - Later we will see that it can return a *pointer*
  - However, it can *modify* the content of an array passed as an argument

- When a function has an array as an argument, we don't need to explicitly specify its size:

```
int fun(int a[]) {
  // size of a[] not specified
  ...
}
```

  - However, it has no way of knowing the size with which the array was declared
  - If it needs to know the size, we have to **pass it explicitly**

Example

- `array_functions.c` (source code) - example functions working with arrays

```c
#include <stdio.h>

// Prints the values of the array a of size n
void show_array(int a[], int n) {
  printf("{");
  for (int i=0; i<n; i++) {
    if (i>0) printf(",");
    printf("%d", a[i]);
  }
  printf("}\n");
}

// Returns the sum of all the values of the array a with size n
int sum_array(int a[], int n) {
  int sum = 0;
  for (int i=0; i<n; i++)
    sum += a[i];
  return sum;
}

// (continues on next slide)
```

Example

- `array_functions.c` (source code) - example functions working with arrays

```c
// (continuation from previous slide)

int main(void) {

  // Creates an array if size 5 and initializes it
  int a[] = {1,2,3,4,5};

  // Calls the previously defined functions to showcase them
  show_array(a, 5);
  int sum = sum_array(a, 5);
  printf("sum = %d\n", sum);

  return 0;
}
```

```
{1,2,3,4,5}
sum = 15
```

Modifying arrays

- If a function modifies an array, this is reflected in the passed argument
  `array_modify.c` (source code) - example of function that modifies array

```c
...
// Changes all values between indices inf and sup to v
void modify_array(int a[], int inf, int sup, int v) {
  for (int i=inf; i<=sup; i++)
    a[i] = v;
}

int main(void) {
  int a[] = {1,2,3,4,5};
  show_array(a, 5);
  modify_array(a, 1, 3, 999); // changes values from indices 1 to 3
  show_array(a, 5);

  return 0;
}
```

```
{1,2,3,4,5}
{1,999,999,999,5}
```

Note: This seems contradictory with what we saw with simple variables that we passed by value (when we talk about pointers we will understand why this happens)

Multidimensional arrays

- We can declare variables with **more than one dimension**

- For example, with two dimensions:

```
int a[3][4]; // a 3x4 matrix
```

  - A table with 3 rows and 4 columns
  - The row and column indices start at zero

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

- To access the element in row *i* and column *j* we write `a[i][j]`

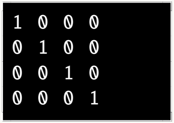
- Although we visualize the matrix in two dimensions, the organization in memory is sequential (in contiguous positions):

$$\overbrace{\boxed{a[0][0] \ \cdots \ a[0][3]}}^{\text{row } 0}\overbrace{\boxed{a[1][0] \ \cdots \ a[1][3]}}^{\text{row } 1}\overbrace{\boxed{a[2][0] \ \cdots \ a[2][3]}}^{\text{row } 2}$$

Example

- matrix.c (source code) - **nested for loops** are useful for multidimensional arrays

```c
#include <stdio.h>

#define N 4

int main(void) {
  int a[N][N];

  // Initialize the matriz
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
      if (i==j) a[i][j] = 1; // main diagonal has 1
      else a[i][j] = 0;      // all other values are zero

  // print the resulting matrix
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++)
      printf("%d ", a[i][j]);
    printf("\n");
  }

  return 0;
}
```



```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

Initialization of multidimensional arrays

- Like with normal arrays, we can initialize when declaring:

```
int a[3][3] = {
  {1,2,3}, // row 0
  {4,5,6}, // row 1
  {7,8,9}, // row 2
};
```

- We need however to declare all dimensions (except eventually the first one, so that the compiler knows how to access the values:)

```
int b[][]   = { {1,2}, {3,4} }; // gives an error
int c[][2]  = { {1,2}, {3,4} }; // no error, ok
int d[2][2] = { {1,2}, {3,4} }; // no error, ok
```

```
error: declaration of 'b' as multidimensional array must have
       bounds for all dimensions except the first
```

$$\overbrace{\underbrace{\boxed{a[0][0]\cdots a[0][M-1]}}_{M\text{ values}}}^{\text{row }0}\overbrace{\underbrace{\boxed{a[1][0]\cdots a[1][M-1]}}_{M\text{ values}}}^{\text{row }1}\cdots\overbrace{\underbrace{\boxed{a[i][0]\cdots a[i][M-1]}}_{M\text{ values}}}^{\text{row }i}$$

- The position of `a[i][j]` is `i*M + j`

Functions and multidimensional arrays

- We can pass multidimensional array as arguments:
  - Like in initialization we must specify all dimensions except the 1st one

```c
int fun1(int m[][], int rows, int cols) {    // error
  ...
}

int fun2(int m[][N], int rows, int cols) {   // ok
  ...
}

int fun3(int m[N][N], int rows, int cols) {  // ok
  ...
}
```

- This need to fix the dimensions of multidimensional arrays is inconvenient *(e.g. it makes it harder to define generic functions for matrices)*
- C programmers can overcome this in several ways
  (no time for that today...)

Example

- matrix_functions.c (source code) - summing and printing matrices

```c
#include <stdio.h>

#define MAX_ROWS 100
#define MAX_COLS 100

// Prints the values of a rows x cols matrix
void show_matrix(int m[MAX_ROWS][MAX_COLS], int rows, int cols) {
  for (int i=0; i<rows; i++) {
    for (int j=0; j<cols; j++)
      printf("%d ", m[i][j]);
    printf("\n");
  }
}

// sums two matrices m1 and m2 of dimensions rows x cols matrix
// stores the result on matrix m3
void sum_matrix(int m1[MAX_ROWS][MAX_COLS],
                int m2[MAX_ROWS][MAX_COLS],
                int m3[MAX_ROWS][MAX_COLS],
                int rows, int cols) {
  for (int i=0; i<rows; i++)
    for (int j=0; j<cols; j++)
      m3[i][j] = m1[i][j] + m2[i][j];
}

// (continues on next slide)
```

Example

- matrix_functions.c (source code) - summing and printing matrices

```
// (continuation from previous slide)

int main(void) {

  // Creates three 3x3 matrices
  int a[MAX_ROWS][MAX_COLS] = {{1,2,3}, {4,5,6}, {7,8,9}};
  int b[MAX_ROWS][MAX_COLS] = {{11,12,13}, {14,15,16}, {17,18,19}};
  int c[MAX_ROWS][MAX_COLS];

  // Sums and prints the resulting matrix
  sum_matrix(a, b, c, 3, 3);
  show_matrix(c, 3, 3);

  return 0;
}
```

```
12 14 16
18 20 22
24 26 28
```