Arrays

CSC100 Introduction to programming in C/C++ - Spring 2025

Marcus Birkenkrahe

April 12, 2025

README

- This script introduces C arrays - an important *data structure*.
- Practice workbooks, input files and PDF solution files in GitHub
- This section, including some sample code, is based on chapter 6 in Davenport/Vine (2015) and chapter 8 in King (2008).
- For the 2025 update, some parts were modified with the help of generative AI (Grok 3, ChatGPT-4o).

Overview

- Variables that can hold only a single data item (a number or a character, which is a number, too) are called **scalars**: `1`, `'a'`
- In mathematics, *ordered tuples* of data (x_{1},...x_{n}) are called **vectors**. In the R code below,a vector `v` is defined and printed:

  ```
  c(1,2,3) -> v ## create a vector of three numbers
  v
  ```

  ```
  [1] 1 2 3
  ```

- In C there are two **aggregate** stuctures that can store *collections* of values: **arrays** and **structures**.

- A **structure** is a forerunner of a **class**, a concept that becomes central in **C++**, which is also called "C with classes".
- Classes contain objects and their properties, and they are a core concept for **object-oriented programming** (OOP).

Collections in other languages

- Different programming languages have different data structures. The language Python has **dictionaries**, the language R has **data frames**, and the language Lisp has **lists**:
- Example with **Python:** a *dictionary* of car data.

```
thisDict = {
    "brand": "Ford",      # key: brand attribute, value: Ford
    "model": "Mustang",   # key: model attribute, value: Mustang
    "year": 1964          # key: year attribute, value: 1964
}
for key, value in thisDict.items():
    print(f"key: {key}, value: {value}")
```

- Example with **R**: a *data frame* of tooth growth data, consisting of three different vectors of the same length but different data types.

```
str(ToothGrowth)
```

```
'data.frame':       60 obs. of  3 variables:
 $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
 $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
 $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

- For **Lisp**, *lists* are the fundamental data structure:

```
(setq my-list '(1 2 3 4 5))
(message "List contents: %s" my-list)
```

- Emacs is programming in Lisp, which is also the oldest language for AI applications; Python is an important general purpose language which dominates machine learning; and R is a language for statistics and data visualization. If you study computer or data science at Lyon, you will learn all of these languages.

What is an array?

- An **array** is a *data structure* containing a number of data values, all of which have the same type (like `int`, `char` or `float`).
- You can visualize arrays as sorted box collections.

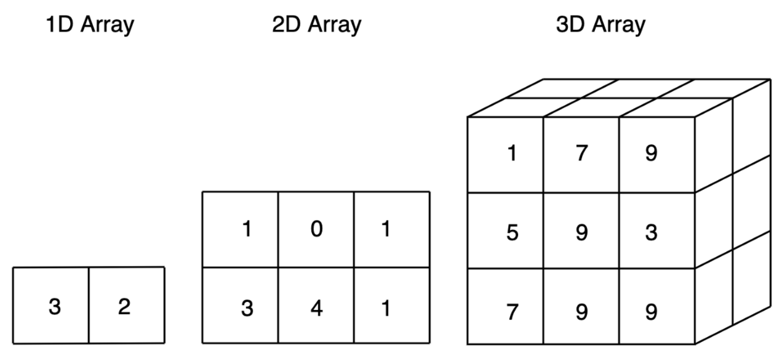

Figure 1: Arrays of different dimensions with values in them

- The computer stores them differently - sequentially as a set of memory addresses.

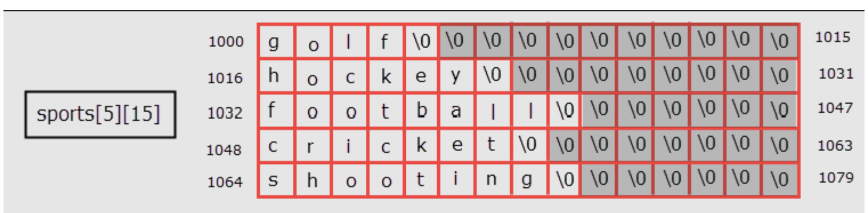

Figure 2: Memory representation of a 2D character array (Source: TheCguru.com)

One-dimensional arrays

- The simplest kind of array has one dimension - conceptually arranged visually in a single row (or column).

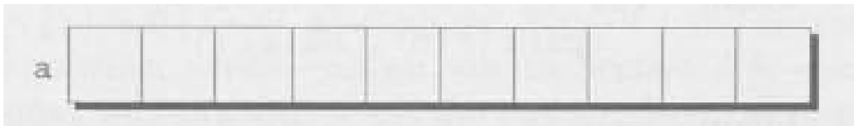

Figure 3: Visualization of a 1-dim array 'a' (Source: King)

- Each element of an array of type T is treated as if it were a variable of type T. Here are three short examples:

```
for ( int i = 0; i < N; i++ )
  a[i] = 0;                    /* clears a */

for ( int i = 0; i < N; i++ )
  scanf("%d", &a[i]);          /* reads data into a */

for ( int i = 0; i < N; i++ )
  sum += a[i];                 /* sums the elements of a */
                               /* sum += a[i] => sum = sum + a[i] */
```

Declaring arrays

- To declare an array, we must specify the *type* and *number* of its elements, e.g. for an array of 10 elements:

```
int a[10];           // declare array a of 10 integers
printf("a[0] = %d\na[9] = %d\n",
       a[1], a[9]);  // print two array elements


a[0] = 0
a[9] = 0
```

- The array must be initialized, just like any scalar variable, to be of use to us (otherwise strange values may appear):

```
int a[10];
for (int i=0;i<10;i++) printf("%d ",a[i]);


2 0 -1075053569 0 21858233 32765 100 0 4096 0
```

- You can initialize arrays explicitly using `{...}`:

  ```
  int int_array[5] = {1,2,3,4,5};  // initialize with integers
  double double_array[] = {2.1, 2.3, 2.4, 2.5}; // initialize with floats
  char char_array[] = {’h’,’e’,’l’,’l’,’o’,’\0’}; // initialize with chars
  ```

  This is how `char_array` looks like (the last character `\0` is only a terminating character):

‘H’	‘e’	‘l’	‘l’	‘o’
0	1	2	3	4

- Control over start/finish of arrays is essential, otherwise you incur a so-called *memory overflow*:

```
char c1[] = {’h’,’e’,’l’,’l’,’o’,’\0’}; // initialize with chars
char c2[] = {’h’,’e’,’l’,’l’,’o’}; // initialize with chars
printf("%s\n%s",c1,c2);
```

```
hello
hellohello
```

Practice Exercises: Declaring Arrays in C

Reading array values

- **Question**: What values do you expect this program to print? Explain the output.

  ```
  int arr[4] = {10, 20};
  printf("arr[0] = %d\n", arr[0]);
  printf("arr[1] = %d\n", arr[1]);
  printf("arr[2] = %d\n", arr[2]);
  printf("arr[3] = %d\n", arr[3]);
  ```

  ```
  arr[0] = 10
  arr[1] = 20
  arr[2] = 0
  arr[3] = 0
  ```

- **Explanation**

  Only the first two values are initialized; the rest default to zero (compiler-dependent - in truth, `arr[3]` and `arr[4]` are undefined).

Default values and garbage data

- **Question**: What happens when you declare an array without initializing it? What values do you see and why?

```
int nums[6]; // define an array with 6 values
for (int i = 0; i < 6; i++) {
  printf("%d ", nums[i]); // print the uninitialized values
 }
```

```
1248297993 32765 100 0 4096 0
```

- **Explanation**:

  Uninitialized local arrays contain garbage values — leftovers in memory.

Fixing initialization

- **Question**: Update the previous program and initialize the array in two different ways.

Initialize array values with a loop

- Initialize the array `nums` to the value 1:

```
int nums[6];
for (int i = 0; i < 6; i++) {
  nums[i] = 1;
  printf("%d ", nums[i]);
 }
```

```
1 1 1 1 1 1
```

Initialize array with an initializer list

- Initialize the array `nums` to the value 1 using an initializer list.

```
int nums[6] = {1,1,1,1,1,1};
for (int i = 0; i < 6; i++) printf("%d ", nums[i]);
printf("\n");
```

```
1 1 1 1 1 1
```

Character array experiments

- **Question**: What will this code print? Why does `word1` behave differently from `word2`?

```
char word1[] = {'h','e','l','l','o'};
char word2[] = {'h','e','l','l','o','\0'};
printf("word1: %s\n", word1);
printf("word2: %s\n", word2);
```

```
word1: hellohello
word2: hello
```

- **Explanation**

  `word1` lacks the `null` character \0, so `printf("%s", ...)` runs past its end and prints whatever happens to be there in the memory.

- Strings like "hello" are stored as arrays. This is how you will do it later:

```
char *word = "hello"; // `word` is a `char` pointer to `h`
printf("word: %s\n", word); // prints the string

for (int p=0; p < 5 ; p++) { // pointer arithmetic
  printf("word: %c\n", word[p]);
 }
```

```
word: hello
word: h
word: e
word: l
word: l
word: o
```

Practice writing declarations

Task: Write C declarations for the following array scenarios, then print them.

1. An array `a` of 10 integers.
2. An array `b` of 5 floats initialized to 1.1, 2.2,...,5.5
3. A character array `c` initialized to the word "Hi"
4. An array `d` of 100 doubles initialized to 0 (print first and last five elements only)

Solution:

```
// 1. An array of 10 integers
int a[10];
for (int i=0;i<10;i++) printf("%d ", a[i]);
// 2. An array of 5 floats initialized to 1.1, 2.2, ..., 5.5
float b[] = {1.1, 2.2, 3.3, 4.4, 5.5}; puts("");
for (int i=0;i<5;i++) printf("%.1f ", b[i]); puts("");
// 3. A character array initialized to the word "Hi"
char c[] = {'H', 'i', '\0'};
for (int i=0;i<2;i++) printf("%c ", c[i]); puts("");
// 4. An array of 100 doubles initialized to 0
// PRINT the first and the last five elements only
double d[100] = {0};
for (int i=0; i < 100; i++) {
  if (i < 5 || i > 94)
    printf("%g ", d[i]);
 }
```

```
1294760200 30910 1294870112 30910 10022912 0 1415751728 32767 1 0
1.1 2.2 3.3 4.4 5.5
H i
0 0 0 0 0 0 0 0 0 0
```

Array length

- An array can have any length. Since the length may have to be adjusted, it can be useful to define it as a macro with `#define`.

```
#define N 10  // directive to define N = 10 everywhere
int a[N]; // declare array of length N
```

- Remember that now `N` will **blindly** be replaced by `10` **everywhere** in the program by the pre-processor.

Array subscripting side effects

C is very permissive

- C does not require that the subscript bounds be checked.
- If a subscript goes out of bounds, the program's behavior is undefined.
- An array subscript may be an integer expression, therefore it's easy to miss subscript violations.

```
foo[i+j*10] = 0; // e.g. i=-10, j=1 => foo[0]
bar[i++];        // e.g. i = -1 => bar[0]
```

Weird `while` loop

- As an example for the weird effects, trace this code:

```
i = 0;
while ( i < N )
   a[i++] = 0;
```

- After `i` is set to `0`, the `while` statement checks whether `i` is less than `N`: to test this, we need to introduce a support variable.

```
#define N 10
int i = 0, a[N]; int j;
while ( i < N ) {
  printf("%d < %d\t", i, N);  // print condition
  j = i;  // support variable
```

```
  a[i++] = 0; // store 0 in a[i] then i = i + 1
  printf("a[%d] = %d\n", j, a[j]); // print i then a[i]
 }
```

```
0 < 10 a[0] = 0
1 < 10 a[1] = 0
2 < 10 a[2] = 0
3 < 10 a[3] = 0
4 < 10 a[4] = 0
5 < 10 a[5] = 0
6 < 10 a[6] = 0
7 < 10 a[7] = 0
8 < 10 a[8] = 0
9 < 10 a[9] = 0
```

- Without the support variable, we would get weird printing results: can you explain them?

```
#define N 10
int i = 0, a[N];
while ( i < N ) {
  printf("%d < %d\t", i, N);  // print condition
  a[i++] = 0; // store 0 in a[i] then i = i + 1
  printf("a[%d] = %d\n", i, a[i]); // print i then a[i]
 }
```

```
0 < 10 a[1] = 0
1 < 10 a[2] = -1075053569
2 < 10 a[3] = 0
3 < 10 a[4] = 1904143161
4 < 10 a[5] = 32767
5 < 10 a[6] = 100
6 < 10 a[7] = 0
7 < 10 a[8] = 4096
8 < 10 a[9] = 0
9 < 10 a[10] = -1387497472
```

- **Explanation 1:**

In the second program the condition test is printed alright, because `i` has not been incremented. But after the assignment, `a[i]` is the next index that has not been assigned a 0 yet, so all values are random. When we print `a[1]` for example, it has not been assigned to `0` yet. `a[10]` is not declared or assigned a value at all, because `a[N]` has the elements `{a[0] ... a[N-1]}`.

It.	i (before)	a[i++] = 0 sets	i (after)	a[i] in printf
1	0	a[0] = 0	1	a[1] uninitialized
2	1	a[1] = 0	2	a[2] uninitialized

- What'd happen if the assignment were with `a[++i]` instead of `a[++i]`?

```
#define N 10
int i = 0, a[N]; int j;
while ( i < N ) {
  printf("%d < N\t", i);  // print condition
  j = i;  // support variable
  a[++i] = 0; // store 0 in a[i] then i = i + 1
  printf("a[%d] = %d\n", j, a[j]); // print i then a[i]
 }
```

Result:

"stack smashing detected" = attempt to write out of bounds.

It.	i (before)	j = i	++i	a[i] = 0 sets	a[j] printed
1	0	0	1	a[1] = 0	a[0] undefined
2	1	1	2	a[2] = 0	a[1] undefined
...	...	...	...	...	...
10	9	9	10	a[10] = 0	a[10] out of bounds

On Windows, you'd get this answer (I have no idea why):

```
0 < N a[0] = 66110
1 < N a[1] = 0
2 < N a[2] = 0
3 < N a[3] = 0
4 < N a[4] = 0
```

```
5 < N a[5] = 0
6 < N a[6] = 0
7 < N a[7] = 0
8 < N a[8] = 0
9 < N a[0] = 66110
```

- **Explanation 2:**

  `a[++i]` would not be right, because `0` would be assigned to `a[0] during the first loop iteration - remember that ~++i` increments `i` first and then stores the result in `i`. The last iteration tries to assign 0 to `a[11]` which is undeclared. You can test that by initializing `int i = -1` at the start. Same problem at the end, for `i=9`, the computer tries to initialize `a[10]`, which is not declared - "stack smashing" means that the computer tries to write beyond its defined boundaries.

Copying arrays into one another

- Be careful when an array subscript has a side effect. Example: the following loop to copy all elements of `foo` into `bar` may not work properly:

```
i = 0;
while (i < N)
  a[i] = b[i++];
```

- The statement in the loop accesses the value of `i` and modifies `i`. This causes undefined behavior. To do it right, use this code:

```
for (i = 0; i < N; i++)
    a[i] = b[i];
```

- This is one example where the `while` loop is not the same as the `for` loop.

Weird `for` loop

- This innocent-looking `for` statement can cause an infinite loop:

```
int a[10], i;

for ( i = 1; i <= 10; i++)
  a[i] = 0;
```

- Explanation:* when `i` reaches `10`, the program stores `0` in `a[10]`. But `a[10]` does not exist (the array ends with `a[9]`), so `0` goes into memory immediately after `a[9]`. If the variable `i` happens to follow `a[9]` in memory, then `i` will be reset to `0`, causing the loop to start over!
- "Stack smashing" because we're writing out of bounds (`a[10]`):

```
int a[10], i;

for ( i = 1; i <= 10; i++)
  a[i] = 0;
```

- Why "stack smashing"?

  > You have corrupted part of the **stack frame** which is where local variables are stored. A **stack canary** guards the stack (like a canary in a coal mine, who died in the presence of toxic gases).

- Illustration of a "stack frame" for the `main` function followed by stack frames for other functions, forming the full **call stack** of routines (and their variables) that can be called.

Iterating over arrays

- `for` loops are made for arrays. Here are a few examples. Can you see what each of them does?

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

> **Answer 1:** `0` is assigned to `a[0]` through `a[9]`.

```
for (i = 0; i < 10 ; i++ )
  scanf("%d", &a[i]);
```

Answer 2: external integer input is assigned to `a[0]` through `a[9]`.

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

Answer 3: The values `a[0]` through `a[9]` are summed up: `sum = sum + a[i=1] = sum + a[i=1] + a[i=0] ...`

Iteration examples

- These short problems build on the three examples you've just seen.
- Open a new file at OneCompiler.Com and put all of these into it.

Initialization with Pattern

- Initialize the array `a` with the values 1,2, ..., 10 using a `for` loop that starts at i = 0;

```
// Initialize the array a with the values 1, 2, ..., 10
// using a for loop
/****************************************************/
// SET array a of 10 elements
int a[10];
// FOR i from 0 to 10: DO
for (int i = 0; i < 10; i++) {
  // initialize array elements
  a[i] = i + 1;
  // PRINT array element
  printf("%d ", a[i]);
 } // END FOR


1 2 3 4 5 6 7 8 9 10
```

Input and Count

- Read 5 whole (non-negative integer) numbers into an array `b` and count how many of them are even:

```
// Read 5 integers into an array and count
// how many of them are even numbers.
/****************************************************/
// SET array b of 5 elements
int b[5];
// SET count to 0
int count = 0;
// PRINT "Enter 5 whole numbers:"
printf("Enter 5 whole numbers: ");
// FOR i from 0 to 10: DO
for (int i = 0; i < 5; i++) {
  // READ array element
  scanf("%d",&b[i]);
  // PRINT array element
  printf("%d ",b[i]);
  // IF array element even
  if (b[i] % 2 == 0) {
    // ADD 1 to count
    count++;
  } // END IF
 } // END FOR
// PRINT "Number of even values = " + count
printf("\nNumber of even values = %d\n", count);
```

```
Enter 5 whole numbers: 10 21 33 4 5
Number of even values = 2
```

- How could this be generalized?
    1. Accepting arrays of any length.
    2. Aborting gracefully when entry is not a whole number.
- Input:

```
echo 10 21 33 4 5 > input
cat input
```

```
10 21 33 4 5
```

Conditional Summation

- Initialize an array `c` of 10 elements, and only sum up the positive values in the array.
- Sample input: 3, -1, 7, 0, -5, 2, 8, -3, 6, -2.
- Sample output: 26.
- Solution:

```
// Sum up only the *positive* values in the array.
/*****************************************************/
// SET array c of 10 elements
int c[10] = { 3, -1, 7, 0, -5, 2, 8, -3, 6, -2 };
// SET sum of positive values to 0
int sum = 0;
// FOR i from 0 to 10; DO
for (int i = 0; i < 10; i++) {
  // IF element of c greater than 0
  if (c[i] > 0) {
    // ADD element to sum
    sum += c[i];
  } // END IF
 } // END FOR
// PRINT "Sum of positive values
printf("Sum of positive values: %d\n", sum);
```

```
Sum of positive values: 26
```

- How could this be generalized?
    1. Accepting arrays of any length.
    2. Aborting gracefully when entry is not a whole number.

Initalizing arrays with *designated initializers (C99)*

- You can give default values to arrays if you want to change only few elements, e.g. here:

```
int a[15] = {0,0,29,0,0,0,0,0,0,0,7,0,0,0,48};
```

- When you initialize explicitly, you don't have to specify the number of elements on the left hand side:

```
int b[] = {0,0,29,0,0,0,0,0,0,0,7,0,0,0,0,48};
```

- You can only initialize non-zero elements:

```
int c[] = { [2] = 29, [10] = 7, [14] = 48};

for (int i=0;i<15;i++) printf("%d ",c[i]);


0 0 29 0 0 0 0 0 0 0 7 0 0 0 48
```

- Iterate over `c` and print only the non-zero elements:

```
int c[] = { [2] = 29, [10] = 7, [14] = 48};

for (int i=0; i<15; i++)
  if (c[i]!=0) printf("%d ",c[i]);


29 7 48
```

Multi-dimensional arrays

- An array may have any number of dimensions.
- Example: the following array declares a 5 x 9 matrix of 5 rows and 9 columns.

```
int m[5][9]; // This goes from m[0][0] to m[4][8]
```

- **Declare** a 2 x 2 matrix named `foo` of floating point values:

```
float foo[2][2]; // declare 2x2 floating point matrix
```

- **Initialize** the matrix with zero values as you would initialize an one-dimensional array.

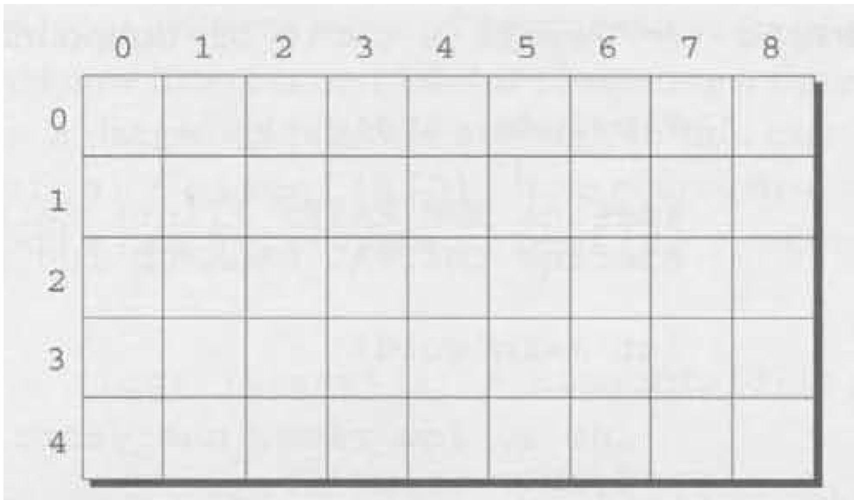

Figure 4: Matrix indexes in a 2-dim C array (Source: King)

```
float foo[2][2] = {0.f};  // declare and init 2x2 float matrix
```

- Print the matrix (using nested `for` loops):

```
float foo[2][2] = {0.f};  // Declare a 2x2 matrix

for (int i=0;i<2;i++) {
  for (int j=0;j<2;j++) {
    printf("%.0f ",foo[i][j]);
  }
  printf("\n");
 }


0 0
0 0
```

- You can also initialize a matrix using designated initializers:

```
double foo[2][2] = {[0][0] = 1.0, [1][1] = 1.0}; // identity matrix
for (int i=0;i<2;i++) {
  for (int j=0;j<2;j++) {
    printf("%.0f ",foo[i][j]);
```

```
  }
  printf("\n");
 }
```

```
1 0
0 1
```

- Or you can initialize every single element (wasteful for 0s):

```
double foo[2][2] = {1.0, 0., 0.,1.0};
for (int i=0;i<2;i++) {
  for (int j=0;j<2;j++) {
    printf("%.0f ",foo[i][j]);
  }
  printf("\n");
 }
```

```
1 0
0 1
```

- Arrays in C cannot be assigned to after their declaration.

Practice declaring and initializing matrices

- Declare a 3 x 3 character matrix `hw`.
- Initialize the matrix using designated initializers, with the letters of "hello world".
- Print the first and the last matrix element ('h','d').
- Solution:

```
// SET row index M
#define M 3
// SET column index N
#define N 3
// SET 3 by 3 character matrix hw to "hello world"
char hw[3][3] = {
```

```
  {'h','e','l'},
  {'o','w','o'},
  {'r','l','d'}
};

// PRINT "First letter = " + "Last letter = "
printf("First letter = %c. Last letter = %c\n",
       hw[0][0], hw[M-1][N-1]);


First letter = h. Last letter = d
```

Accessing arrays with `[]` (index operator)

- To access the element in row `i` and column `j`, we must write `m[i][j]`.
- To access row `i` of `m`, we write `m[i]`
- The expression `m[i,j]` is the same as `m[j]` (don't use it)
- C stores arrays not in 2 dim but in row-major order: In row-major order, the entire row is stored in sequence before moving to the next row.

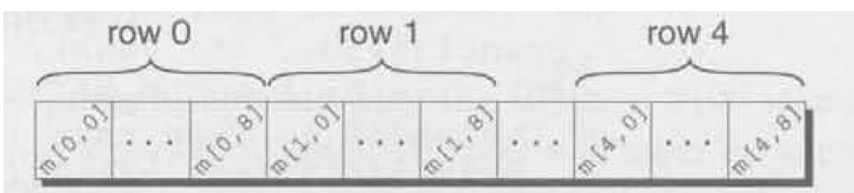

Figure 5: Row-major memory storage in C (Source: King)

- Multi-dimensional arrays play a lesser role in C than in many other programming languages because C has a more flexible way to store multi-dimensional data, namely *arrays of pointers*.

Examples: Accessing arrays

- In the 4x4 matrix below, what are the values of:

```
int foo[4][4] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
for (int i=0;i<4;i++) {
  for (int j=0;j<4;j++) {
    printf("%3i ",foo[i][j]);
  }
  printf("\n");
 }
```

```
  0   1   2   3
  4   5   6   7
  8   9  10  11
 12  13  14  15
```

1. `foo[0][0]` - Answer:

   0
2. `foo[1][3]` - Answer:

   7
3. `foo[2][1]` - Answer:

   9
4. `foo[4][4]`

   Out of bounds!

- Let's check:

```
int foo[4][4] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
for (int i=0;i<4;i++) {
  for (int j=0;j<4;j++) {
    printf("[%d][%d]:%3i ",i,j,foo[i][j]);
  }
  printf("\n");
 }
```

```
[0][0]:  0 [0][1]:  1 [0][2]:  2 [0][3]:  3
[1][0]:  4 [1][1]:  5 [1][2]:  6 [1][3]:  7
[2][0]:  8 [2][1]:  9 [2][2]: 10 [2][3]: 11
[3][0]: 12 [3][1]: 13 [3][2]: 14 [3][3]: 15
```

- How would you declare a `matrix` of characters a,b,c,d?

```
char matrix[2][2]={
  {'a','b'},
  {'c','d'}
};

for (int i=0;i<2;i++) {
  for (int j=0;j<2;j++) {
    printf("%c ",matrix[i][j]);
  }
  printf("\n");
 }


a b
c d
```

Practice: Accessing arrays with nested `for` loops

- Nested `for` loops are ideal for processing multi-dimensional arrays.

2x2 matrix of floating point values

- Declare and print a 2 x 2 array `M` of floating-point values.
- Sample output:

```
0.0  3.14
2.71 0.0
```

- Open a new file in `OneCompiler.com`.
- Let's write the pseudocode first:

```
// Init and print 2x2 floating point matrix values 0,3.14,2.71,0
/**************************************************************/
// SET M to 2x2 matrix with M[0][1]=3.14, M[1][0]=2.71

// PRINT M
// FOR row in 0 to 2; DO
```

```
    // FOR col in 0 to 2; DO
       // PRINT m[row][col]
    // END FOR
    // PRINT new line
// END FOR
```

- Code:

```
// Init and print 2x2 floating point matrix values 0,3.14,2.71,0
/************************************************************/
// SET M to 2x2 matrix with M[0][1]=3.14, M[1][0]=2.71
float foo[2][2]={ [0][1] = 3.14, [1][0] = 2.71 };
// PRINT M
// FOR row in 0 to 2; DO
for ( int row=0; row < 2; row++) {
  // FOR col in 0 to 2; DO
  for ( int col=0; col < 2; col++) {
    // PRINT m[row][col]
    printf("%3.2f ", foo[row][col]);
  } // END FOR
  printf("\n"); // PRINT new line
 } // END FOR
```

```
0.00 3.14
2.71 0.00
```

5x5 identity matrix

- Open a new file in `OneCompiler.com`.
- The following code code initializes a 5x5 *identity* matrix.

  1. Set the dimension of the matrix to N = 5
  2. Declare a `double` matrix named `ident`
  3. Loop over rows with loopindex `row`
  4. For each row, loop over columns with column index `col`
  5. Set each diagonal element `ident[row][col]` to 1, all others to 0
  6. Print the resulting matrix

```
// PRINT N x N identity matrix
/*****************************/
// DEFINE N as 5
#define N 5
// SET N x N integer matrix ‘ident‘
int ident[N][N];
// SET row, col indices
int row, col;
// FOR row from 0 to N; DO
for (row = 0; row < N; row++) {
  // FOR col from 0 to N; DO
  for (col = 0; col < N; col++) {
    // IF row index equal to col index
    if (row == col) {
      // SET ident[row][col] to 1
      ident[row][col] = 1;
    } else { // OTHERWISE
      // SET ident[row][col] to 0
      ident[row][col] = 0;
    } // END IF
    printf("%d ", ident[row][col]);
  } // END FOR
  printf("\n"); // PRINT new line
 } // END FOR
```

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

- By comparison, this is how easy it is to declare, create and print an identity matrix in a language that is built for math manipulation, R:

```
diag(5) #    diag
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
```

```
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

- To initialize an array, you can use brackets as in the 1-dim case, but for each dimension, you need a new set of `[]`.
- What happens in the next code block? What do you think the output looks like?

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

for (int i=0;i<3;i++) {
  for(int j=0;j<3;j++) {
    printf("%d ", m[i][j]);
  }
  printf("\n");
 }
```

```
1 2 3
4 5 6
7 8 9
```

- By comparison, in R this looks like:

```
(matrix(data = 1:9,
        nrow = 3,
        byrow= TRUE)) # populate
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

- How could you populate the matrix column-wise instead of row-wise?

  By swapping the indices in the print statement.

- Test it:

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

for (int i=0;i<3;i++) {
  for(int j=0;j<3;j++) {
    printf("%d ", m[j][i]);  // prints matrix column-wise
  }
  printf("\n");
 }
```

```
1 4 7
2 5 8
3 6 9
```

- In R, that's the default, so the command is even shorter:

```
(matrix(1:9,3))
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

The size of arrays

- The `sizeof` operator can determine the size of arrays (in bytes).
- If `a` is an array of `10` integers, then `sizeof(a)` is 40 provided each integer requires 4 bytes of storage.
- Write this in your practice file: The block below declares and initializes an array of 10 elements and prints its size in bytes.

```
int a[100000] = {0};  // initialize all array elements with 0
printf("%ld", sizeof(a));
```

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

- You can use the operator also to measure the size of an array: dividing the array size by the element size gives you the length of the array:

  ```
  int a[10] = {0};
  printf("%d", sizeof(a)/sizeof(a[0])); // prints length of array a
  ```

  ```
  10
  ```

- You can use this last fact to write a `for` loop that goes over the whole *length* of an array - then the array does not have to be modified if its length changes (see practice file).

Use `sizeof` in a `for` loop

- The code block below defines an array `a` of length `5` initialized with 0. We then overwrite the array elements with 1.
- Source code:

  ```
  // DEFINE length N of array
  #define N 5
  // SET array to 0
  int a[N] = {0};
  // FOR i from 0 to length of a; DO:
  for (int i = 0; i < sizeof(a)/sizeof(a[0]); i++) {
    // SET element i of a to 1
    a[i] = 1; // re-initialize array
    // PRINT "a[i] = "
    printf("a[%d] = %d\n", i, a[i]);
   } // END FOR
  ```

Use sizeof to print a matrix

- Example:

```
int B[3][3] = {0};     // 3 * 3 = 9 array elements
printf("%ld", sizeof(B));  // 9 * 4 = 36 bytes
```

```
36
```

- If an array of N elements has length N * 4 (one for every byte of length 4), what is the length of a matrix of size M x N?

  > It is the number of matrix elements (stored linearly) times the byte length. In the case of N = 4, M = 3 that is 4 * 3 * 4 = 48.

- Storing a matrix:

```
#define M 4
#define N 3
int C[M][N] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

- Can we use sizeof when looping over rows and columns?

```
#define M 4
#define N 3
int C[M][N] = {1,2,3,4,5,6,7,8,9,10,11,12};
for (int i = 0; i < M ; i++) { // iterate over M rows
  for(int j = 0; j < N; j++) { // iterate over N columns
    printf("%3d", C[i][j]);
  }
  printf("\n"); // next row
 }
```

```
  1  2  3
  4  5  6
  7  8  9
 10 11 12
```

- The length of the row vectors:

  ```
  #define M 4
  #define N 3
  int C[M][N] = {1,2,3,4,5,6,7,8,9,10,11,12};
  printf("%ld\n", sizeof(C)); // size of matrix C = M * N * 4
  printf("%ld\n", sizeof(C)/sizeof(C[0][0])); // size of row = 48 / 4
  printf("%ld\n", sizeof(C)/sizeof(C[0][0])*M/N); // size of column = 48 / 3
  ```

  ```
  48
  12
  16
  ```

Noweb chunks

```
for (int i=0;i<2;i++) {
  for (int j=0;j<2;j++) {
    printf("%.0f ",foo[i][j]);
  }
  printf("\n");
 }
```

```
for (int i=0;i<4;i++) {
  for (int j=0;j<4;j++) {
    printf("%3i ",foo[i][j]);
  }
  printf("\n");
 }
```

```
for (int i=0;i<4;i++) {
  for (int j=0;j<4;j++) {
    printf("[%d][%d]:%3i ",i,j,foo[i][j]);
  }
  printf("\n");
 }
```

```
for (int i=0;i<2;i++) {
  for (int j=0;j<2;j++) {
    printf("%c ",matrix[i][j]);
```

```
  }
  printf("\n");
 }

for (int i=0;i<2;i++) {
  for (int j=0;j<2;j++) {
    printf("%s ",matrix[i][j]);
  }
  printf("\n");
 }
```

References

- Davenport/Vine (2015) C Programming for the Absolute Beginner (3ed). Cengage Learning.
- Kernighan/Ritchie (1978). The C Programming Language (1st). Prentice Hall.
- King (2008). C Programming - A modern approach (2e). W A Norton.
- Orgmode.org (n.d.). 16 Working with Source Code [website]. URL: orgmode.org
- Image 2 from: TheCguru.com