Arrays

CSC100 / Introduction to programming in C/C++

Table of Contents

README

- This script introduces C arrays.
- Practice workbooks and input files in GDrive
- PDF version of this file and of the completed practice workbooks is available in GitHub.
- This section, including some sample code, is based on chapter 6 in Davenport/Vine (2015) and chapter 8 in King (2008).

Overview

- Variables that can hold only a single data item (a number or a character, which is a number, too) are called **scalars**

- In mathematics, ordered tuples of data are called **vectors**[1]:

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

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

- In C there are two **aggregate** stuctures that can store collections of values: **arrays** and **structures**.
- **structures** are the forerunners of classes, a concept that becomes central in C++ (more on structures).

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 box collections.

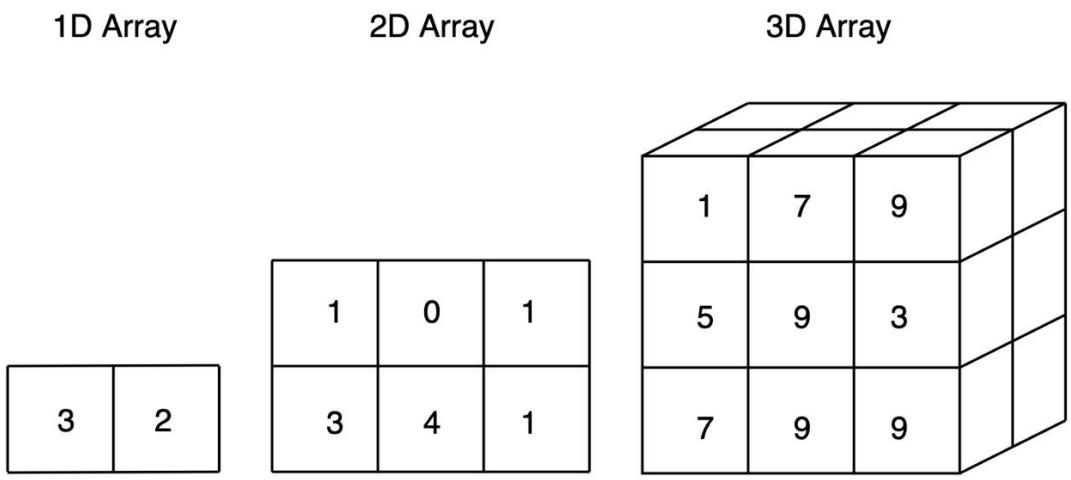

Figure 1: Arrays of different dimensions with values in them

- 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).

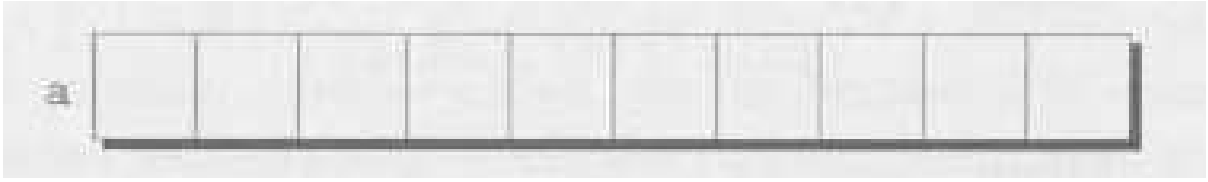

Figure 3: Visualization of a 1-dim array 'a' (Source: King)

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("%d", a[9]);  // print one array element
```

```
208628
```

  - If you run this block repeatedly, you see what the computer stores in the respective memory location - random integers
- The array must be initialized, just like any scalar variable, to be of use to us.
- You can initialize arrays explicitly using `{...}`:

```
int intarray[5] = {1,2,3,4,5};
double doublearray[] = {2.1, 2.3, 2.4, 2.5};
char chararray[] = {'h','e','l','l','o','\0'};
```

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

'H'	'e'	'l'	'l'	'o'
0	1	2	3	4

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
// ...
int a[N];
```

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

Array subscripting

- *Subscripting* or *indexing* means accessing a particular array element.
- Array elements in C are always numbered starting from 0, so the elements of an array of length `n` are *indexed* from `0` to `n-1`.

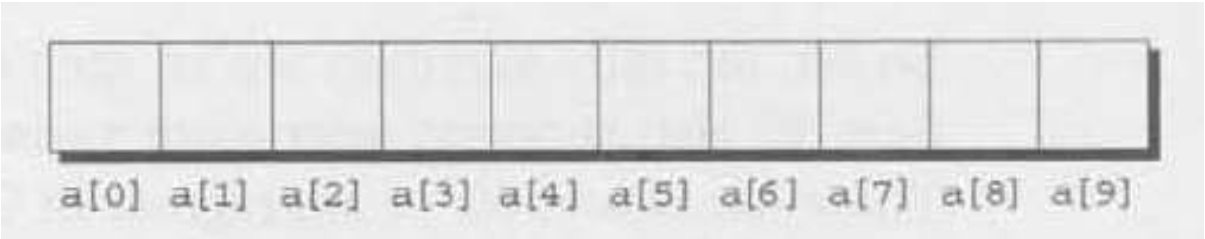

Figure 5: Indexing of an 1-dim array 'a' (Source: King)

- Index expressions `a[i]` can be used like other variables:

```
int a[10];  // declare array

a[0] = 1;   // assign value to array element
a[5] = 2 * 2; // assign operation result to array element

printf("%d\n", a[5]); // print array element
printf("%d\n", a[5] - 4);
printf("%d\n", ++a[0]);
```

```
4
0
2
```

Side effects

- 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:

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

- 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`.
  - If it is, `0` is assigned to `a[i]`, `i` is incremented, and the loop repeats.
  - Note that `a[++i]` would not be right, because `0` would be assigned to `a[0]` during the first loop iteration.

- 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];
```

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;  // initialize a with 0s
```

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

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

Let's practice!

- `[ ]` Head over to GDrive for the first workbook `array1.org`.

Multi-dimensional arrays

Setup

- 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]
```

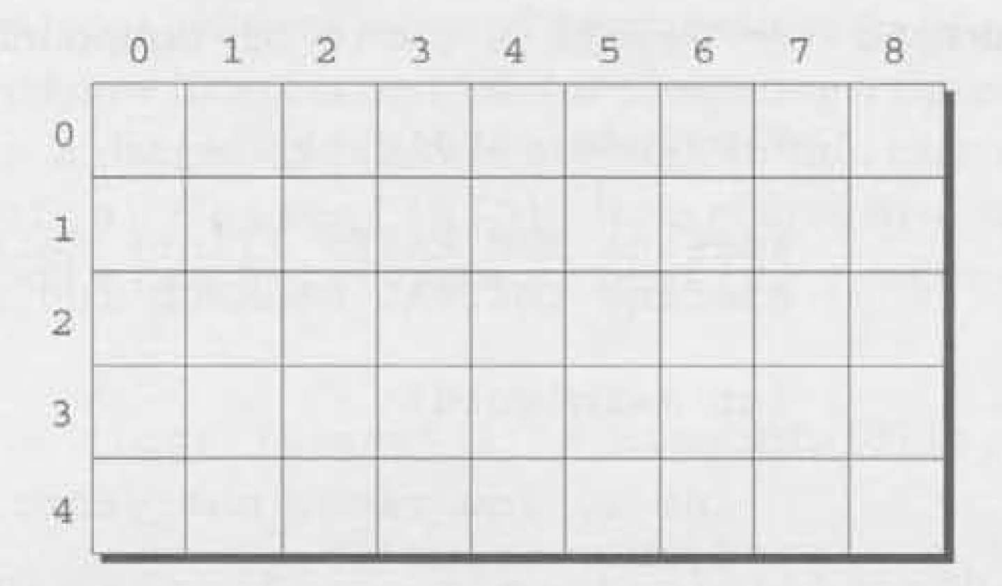

Figure 6: Matrix indexes in a 2-dim C array (Source: King)

Accessing arrays

- 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:

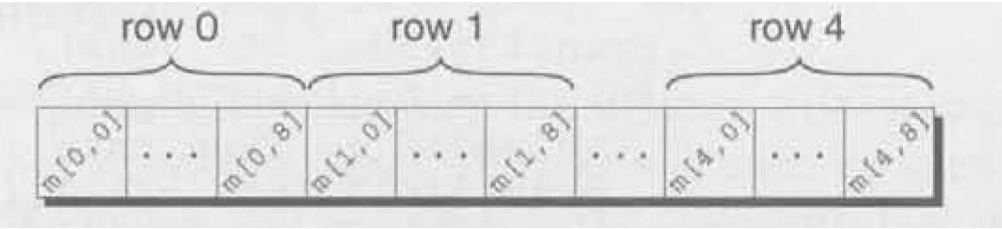

Figure 7: Row-major memory storage in C (Source: King)

- 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*.

Accessing arrays with nested `for` loops

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

- Here is the code to initialize a 10x10 *identity* matrix.

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
  {
    for (col = 0; col < N; col++)
      {
        if (row == col) {
          ident[row][col] = 1.0;
        } else {
          ident[row][col] = 0.0;
        }
      }
  }
```

- To initialize an array, you can use brackets as in the 1-dim case.

```
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");
 }
```

```
123
456
789
```

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[2].

```
int a[10] = {0};
printf("%d", sizeof(a));
```

```
40
```

- You can use the operator also to measure the size of an array element: 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.

Let's practice!

- `[X]` Head over to GDrive for the second workbook `array2.org`.

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

Footnotes:

[1] The code block is an example of the statistical programming language R, which is especially strong when it comes to vector manipulation. `c()` is R's concatenation function that chains elements together to form a vector.

[2] On a 32-bit computer, an `int` ranges from -32,768 to 32,767 and only requires 2 bytes of storage.

Author: Marcus Birkenkrahe
Created: 2022-04-25 Mon 09:09