Table of Contents

s#+TITLE:Arrays

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).

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]. In 1, a vector `v` is defined and printed

```
v <- c(1,2,3)  ## 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**.
- **structures** are the forerunners of classes, a concept that becomes central in C++ (more on structures).

- Different programming languages have different data structures. The language Python also knows **dictionaries** (see 1, and R knows **data frames** and **lists** (see 1).

- Example with Python: a *dictionary* of car data.

```
thisDict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
print(thisDict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

- 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 ...
```

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.

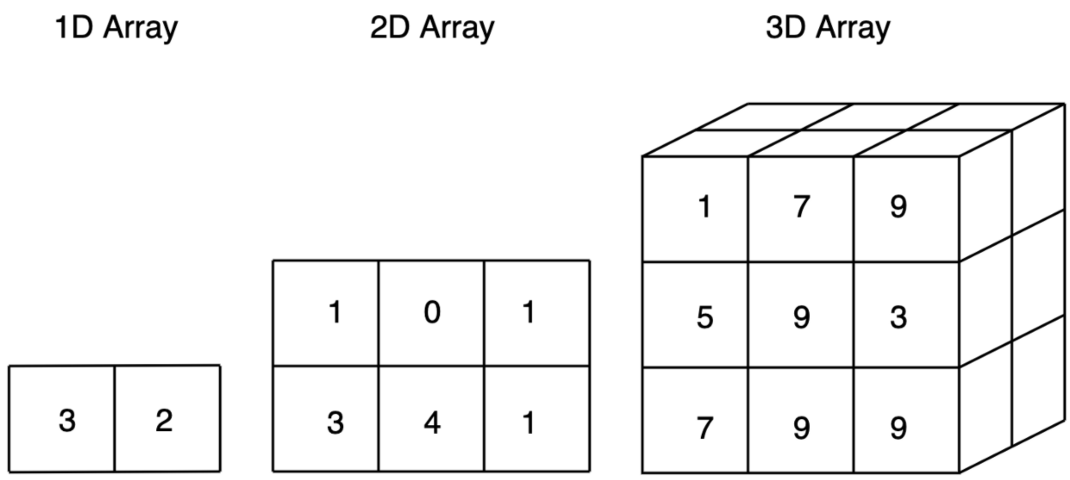


Figure 1: Arrays of different dimensions with values in them

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

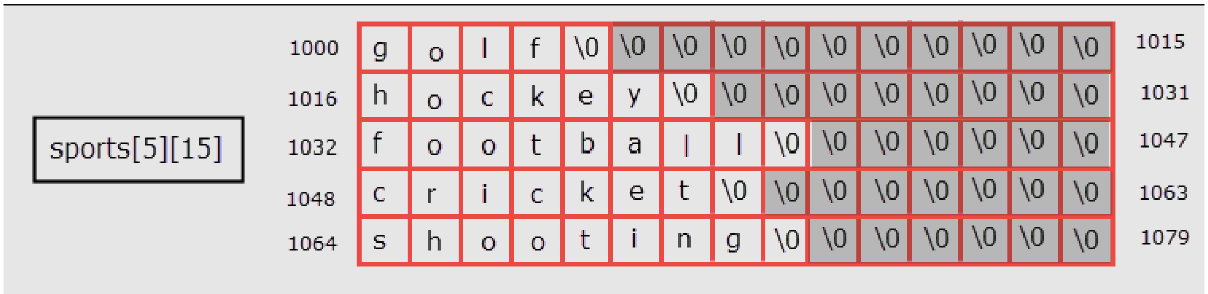


Figure 2: Memory representation of a 2D character array (Source: TheCguru.com)

One-dimensional arrays

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

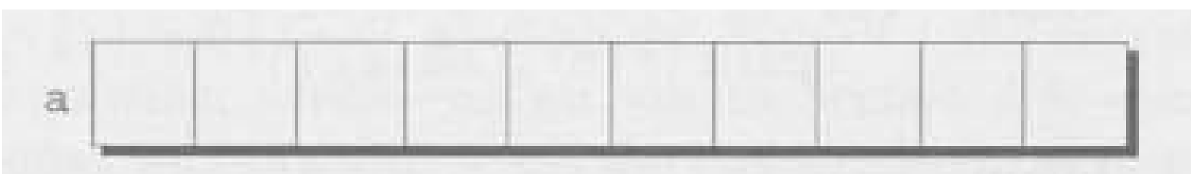


Figure 3: Visualization of a 1-dim array 'a' (Source: King)

- Each element of an array of type T is treated as if it were a variable of type T:

```
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
```

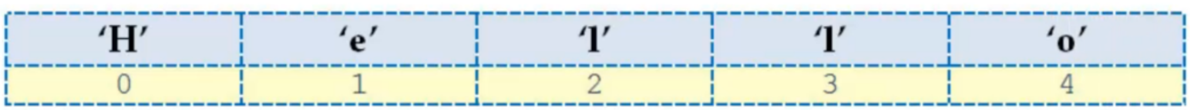
```
a[0] = -1225942824
a[9] = 0
```

- If you run this block repeatedly, you see what the computer stores in the respective memory locations - random integers[2]
- The array must be initialized, just like any scalar variable, to be of use to us.

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

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

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



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];
```

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

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 4: 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); // subtracts 4 from 4
printf("%d\n", ++a[0]); // ++a[0] => a[0] + 1
```

```
4
0
2
```

Array subscripting side effects

C is too 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 < 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]
 }
```

```
0 < N   a[0] = 0
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[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 < N\t", i);  // 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 < N   a[1] = -1225808624
1 < N   a[2] = -1226065704
2 < N   a[3] = -1226062408
3 < N   a[4] = 66748
4 < N   a[5] = 66708
5 < N   a[6] = 0
6 < N   a[7] = 66328
7 < N   a[8] = 0
8 < N   a[9] = 0
9 < N   a[10] = 10
```

- **Explanation 1:** in 1, 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]}`.

- What would happen if the assignment were with `a[++i]`? Let's see:

```
#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]
 }
```

```
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.

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];
  ```

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!

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++ ) // execute the loop for i = 0, 1....,9
    a[i] = 0;   // assign the value 0 to a[i=0], a[i=1],....,a[i=9]
  ```

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

```
for (i = 0; i < 10 ; i++ )  // execute the loop for i = 0, 1....,9
  scanf("%d", &a[i]);     // scan input values and assign them to a[0]...a[9]
```

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

```
for (i = 0; i < 10 ; i++ )   // execute the loop for i = 0, 1....,9
  sum += a[i];  // add a[0] through a[9] and store result in sum
                // sum = sum + a[i=0]
                // sum = sum + a[i=1] = sum + a[i=1] + a[i=0] ...
```

Answer 3: The values `a[0]` through `a[9]` are summed up.

Let's practice!

- [Download `array1.org` from GitHub]: **tinyurl.com/27uv358b**

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]
```

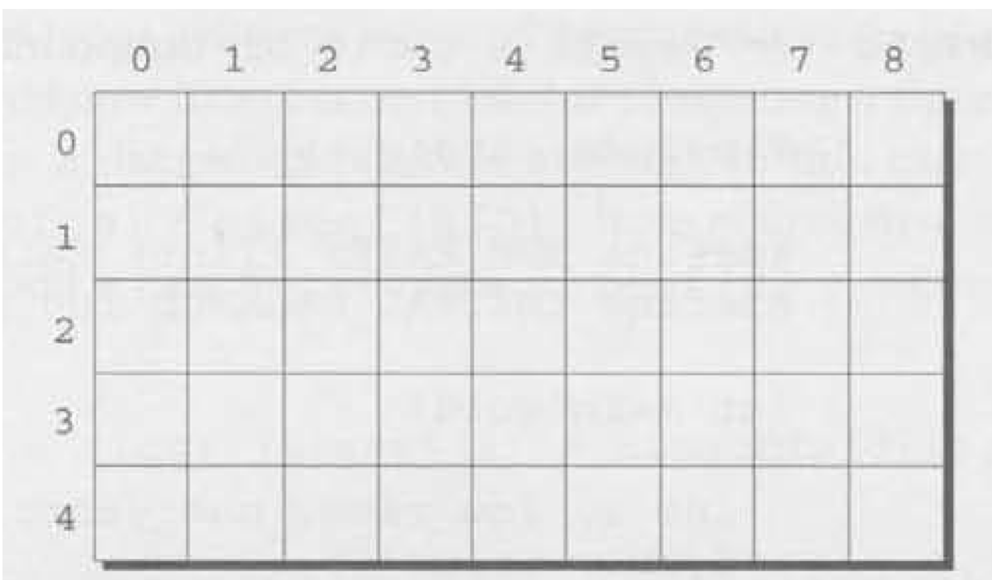


Figure 5: Matrix indexes in a 2-dim C array (Source: King)

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:

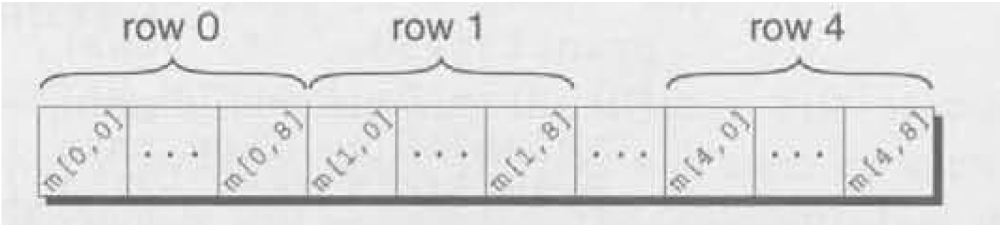


Figure 6: Row-major memory storage in C (Source: King)

- 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.

- The code in 1 initializes a 10x10 *identity* matrix.

```
#define N 5

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;
        }
        printf("%g ", ident[row][col]);
      }
    printf("\n");
  }
```

```
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
```

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

- [ ]

  What happens in 1? 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
  ```

- [ ]

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

  > By swapping the indices in the `printf` statement.

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[3].

- The block 1 declares and initializes an array of 10 elements and prints its size in bytes.

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

  ```
  40
  ```

- 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:

  ```C
  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.

Use `sizeof` to print a matrix

- 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`?

```
int B[3][3] = {0};
printf("%d", sizeof(B));
```

```
36
```

It is the number of matrix elements (stored linearly) times the byte length.

- 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? 1 executes such a loop.

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

```
 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("%d\n", sizeof(C)); // size of matrix C
printf("%d\n", sizeof(C)/sizeof(C[0][0])); // size of row
printf("%d\n", sizeof(C)/sizeof(C[0][0])*M/N); // size of column
```

```
48
12
16
```

Let's practice!

Download `array2.org` from GitHub: **tinyurl.com/3hazjds8**

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] What exactly is displayed depends on the computer you use. On Windows, the array is not automatically initialized, but on the Pi, some elements seem to be initialized with `0`.

[3] 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: 2022-06-20 Mon 21:34