Arrays

CSC100 Introduction to programming in C/C++ - Spring 2024

Marcus Birkenkrahe

April 21, 2024

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**: `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 forerunners of a *class*, a concept that becomes central in C++. Classes contain objects and their properties.

- 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}")

key: brand, value: Ford
key: model, value: Mustang
key: year, value: 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 ...
```

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

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

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.

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

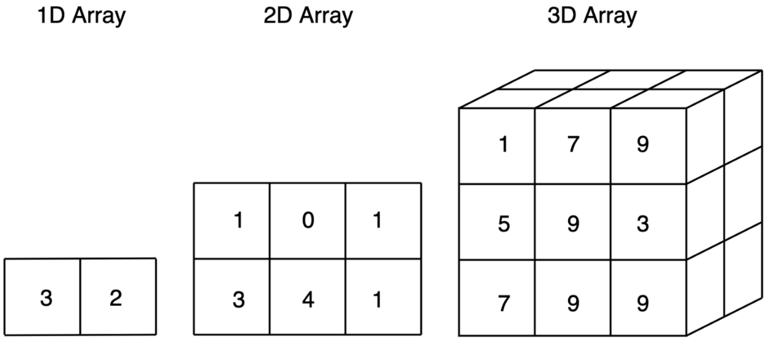
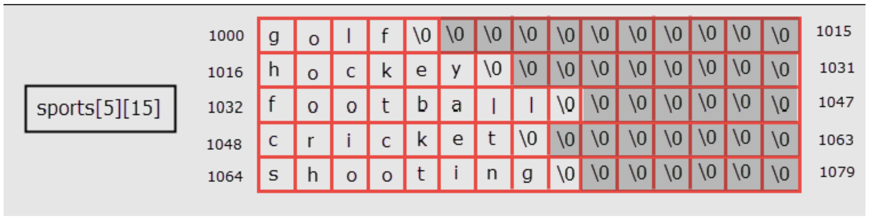


Figure 1: Arrays of different dimensions with values in them



1000	g	o	l	f	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	1015
1016	h	o	c	k	e	y	\0	\0	\0	\0	\0	\0	\0	\0	\0	1031
1032	f	o	o	t	b	a	l	l	\0	\0	\0	\0	\0	\0	\0	1047
1048	c	r	i	c	k	e	t	\0	\0	\0	\0	\0	\0	\0	\0	1063
1064	s	h	o	o	t	i	n	g	\0	\0	\0	\0	\0	\0	\0	1079

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).

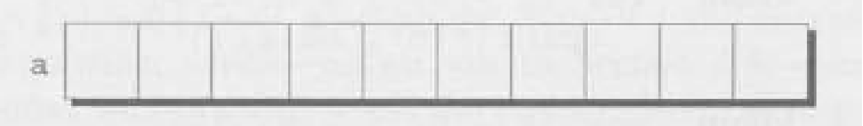


Figure 3: Visualization of a 1-dim array 'a' (Source: King)

- 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] = 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):

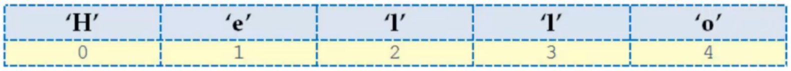
```
int a[10];
for (int i=0;i<10;i++) printf("%d ",a[i]);
```

```
2 0 -1075053569 0 1079232985 32764 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
```

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 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] = 0
1 < N a[2] = -1075053569
2 < N a[3] = 0
3 < N a[4] = -1531307703
4 < N a[5] = 32766
5 < N a[6] = 100
6 < N a[7] = 0
7 < N a[8] = 4096
8 < N a[9] = 0
9 < N a[10] = 91062272
```

- **Explanation 1:**

> In **??**, 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'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]
 }
```

On Windows, you'd get this answer:

```
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!

- Let's smash the stack!

```
int a[10], i;

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

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] ...`

Let's practice!

- Let's practice: download tinyurl.com/cpp-array-practice as array.org
- The first two problems can be solved with what you've already heard (one-dimensional arrays).

Initalizing arrays with *designated initializers*

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

- You can only initialize non-zero elements:

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

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

  0 0 29 0 0 0 0 0 0 0 7 0 0 0 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]
  ```

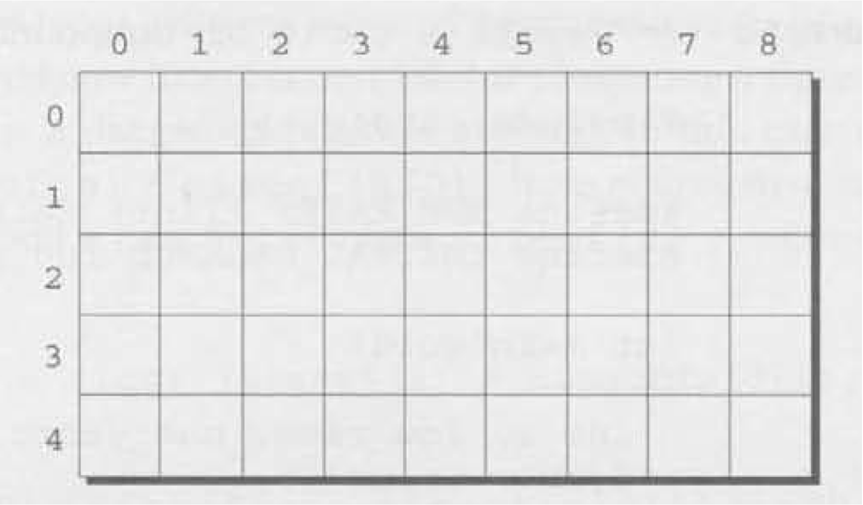


Figure 4: Matrix indexes in a 2-dim C array (Source: King)

- In a practice file, **declare** a 2 x 2 matrix named `foo` of floating point values.
- **Initialize** the matrix with zero values as you would initialize an one-dimensional array.
- Solution:

  ```
  // Declare a 4 x 4 matrix
  float foo[2][2] = {0.f};
  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};
  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
  ```

  ```
  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
  ```

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:
- 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.*

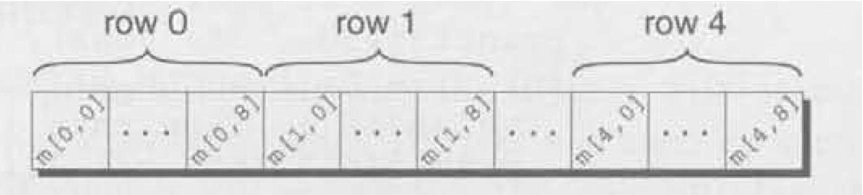


Figure 5: Row-major memory storage in C (Source: King)

- Practice! 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]`

   0

2. `foo[1][3]`

   7

3. `foo[2][1]`

   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
```

- Challenge: How would you declare a matrix of characters a,b,c,d?
- In your practice file, start with a **vector** of characters a,b,c,d.
- Then try a matrix.

Accessing arrays with nested `for` loops

- Nested `for` loops are ideal for processing multi-dimensional arrays.
- Practice! Declare and print a 2 x 2 array of floating-point values.

```
0    3.14
2.71 0
```

- Write the pseudocode first:
- Code:
- The following code code initializes a 10x10 *identity* matrix.
  1. Set the dimension of the matrix to N = 10
  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

```
#define N 5

double ident[N][N];   // matrix dimension is N * N
int row, col;         // loop indices for row and column

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
```

- 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
  ```

- 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(1:9,3,byrow=TRUE))
  ```

- 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]);
    }
    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))
  ```

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));
  ```

  ```
  400000
  ```

- 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` to print a matrix

- Example:

  ```
  int B[3][3] = {0};      // 3 * 3 = 9 array elements
  printf("%d", 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
```

Let's practice!

The last two problems in tinyurl.com/cpp-array-practice can be solved with what you've just heard (multi-dimensional arrays).

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");
 }
```

```
puts("");
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