Practical Programming

The C Language : Arrays

David Bouchet

david.bouchet.epita@gmail.com

Arrays

- An array is a collection of values.
- All values have the same type.
- The length cannot be changed.
- Arrays can be multidimensional.
- Values are selected by indexes.
- Indexes are unsigned integers (usually *size_t* type).
- Indexes start at 0.

One-Dimensional Arrays

Declaring One-Dimensional Arrays

General syntax

```
<type> <identifier>[<length>];
```

Examples

```
int a[18];  // 18 elements of int type
char c[42]; // 42 elements of char type
float f[2]; // 2 elements of float type
```

Not Initialized by Default

```c
int a[5];

for (size_t i = 0; i < 5; i++)
    printf("a[%zu] = %i\n", i, a[i]);
```

```
a[0] = 0
a[1] = 0
a[2] = -825155456
a[3] = 22030
a[4] = -210797760
```

Initializing Arrays

```c
int a[5] = { 30, 9, 72, 17, 18 };

for (size_t i = 0; i < 5; i++)
    printf("a[%zu] = %i\n", i, a[i]);
```

```
a[0] = 30
a[1] = 9
a[2] = 72
a[3] = 17
a[4] = 18
```

Initializing Arrays

No length

```
int a[] = { 30, 9, 72, 17, 18 };

for (size_t i = 0; i < 5; i++)
    printf("a[%zu] = %i\n", i, a[i]);
```

```
a[0] = 30
a[1] = 9
a[2] = 72
a[3] = 17
a[4] = 18
```

Initializing Arrays

Unset values are initialized to zero

```
int a[5] = { 30, 42 };

for (size_t i = 0; i < 5; i++)
    printf("a[%zu] = %i\n", i, a[i]);
```

```
a[0] = 30
a[1] = 42
a[2] = 0
a[3] = 0
a[4] = 0
```

Initializing Arrays

Unset values are initialized to zero

```
int a[5] = { [1] = 30, [3] = 42, 9 };

for (size_t i = 0; i < 5; i++)
    printf("a[%zu] = %i\n", i, a[i]);
```

```
a[0] = 0
a[1] = 30
a[2] = 0
a[3] = 42
a[4] = 9
```

Initializing Arrays

Initializing all values to zero

```
int a[5] = {};

for (size_t i = 0; i < 5; i++)
    printf("a[%zu] = %i\n", i, a[i]);
```

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

Out of Bound Access

Out of bound access is undefined behavior

Out of Bound Access

out_of_bound.c

```c
#include <stdio.h>

int main()
{
    int a[5];
    printf("a[20] = %i\n", a[20]);
    return 0;
}
```

Out of Bound Access

```
$ gcc -Wall -Wextra out_of_bound.c
$ ./a.out
a[20] = -692211237
$ ./a.out
a[20] = -2024418853
$ ./a.out
a[20] = 1341063643
```

No compilation errors!

No compilation warnings!

Random values!

Out of Bound Access

out_of_bound.c

```c
#include <stdio.h>

int main()
{
    int a[5];
    printf("a[5000] = %i\n", a[5000]);
    return 0;
}
```

Out of Bound Access

```
$ gcc -Wall -Wextra out_of_bound.c
$ ./a.out
Segmentation fault (core dumped)
```

No compilation errors!

No compilation warnings!

Segmentation fault!

The *sizeof()* Operator

If ***sizeof()*** is called in the same body as an array declaration, it returns the size in bytes of the array.

Otherwise, it returns the size in bytes of an address (always 8 bytes in the LP64 data model).

The *sizeof()* Operator

```c
char a0[] = { 5, 10, 3 };
int a1[] = { 5, 10, 3 };

printf("sizeof(a0) = %zu\n", sizeof(a0));
printf("sizeof(a1) = %zu\n", sizeof(a1));
```

```
sizeof(a0) = 3
sizeof(a1) = 12
```

The *sizeof()* Operator

```
char a0[] = { 5, 10, 3 };
int a1[] = { 5, 10, 3 };

size_t len0 = sizeof(a0) / sizeof(char);
size_t len1 = sizeof(a1) / sizeof(int);

printf("a0: %zu elements\n", len0);
printf("a1: %zu elements\n", len1);
```

```
a0: 3 elements
a1: 3 elements
```

The *sizeof()* Operator

```
size_t array_size(int a[])
{
    return sizeof(a);
}
```

Be careful!

The '**a**' array is not declared in the body of ***array_size()***.

→ ***sizeof()*** returns 8.

The *sizeof()* Operator

```
int a0[] = { 5 };
int a1[] = { 5, 10, 3 };

printf("array_size(a0) = %zu\n", array_size(a0));
printf("array_size(a1) = %zu\n", array_size(a1));
```

```
array_size(a0) = 8
array_size(a1) = 8
```

The *sizeof()* Operator

Therefore, in most cases, when an array is passed to a function, the length must be passed too.

```
void my_function(int a[], size_t len)
{
    // some code...
}
```

Arrays as Parameters

When an array, is passed to a function, it can be modified by the function (no copy is made).

Arrays as Parameters

```c
void set_to_ten(int a[], size_t len)
{
    for (size_t i = 0; i < len; i++)
        a[i] = 10;
}
```

```c
int x[] = { 1, 2, 3, 4 };

set_to_ten(x, 4);

for (size_t i = 0; i < 4; i++)
    printf("x[%zu] = %i\n", i, x[i]);
```

```
x[0] = 10
x[1] = 10
x[2] = 10
x[3] = 10
```

Arrays as Parameters – *const*

To prevent any modifications, the ***const*** keyword must be used.

```
void my_function(const int a[], size_t len)
{
    // The ‘a’ array cannot be modified.
    // It can be read only.
}
```

Arrays as Parameters – *const*

```c
void set_to_ten(int a[], size_t len)
{
    for (size_t i = 0; i < len; i++)
        a[i] = 10;
}
```

```c
void print(const int a[], size_t len)
{
    for (size_t i = 0; i < len; i++)
        printf("x[%zu] = %i\n", i, a[i]);
}
```

```c
int x[] = { 1, 2, 3, 4 };

set_to_ten(x, 4);
print(x, 4);
```

```
x[0] = 10
x[1] = 10
x[2] = 10
x[3] = 10
```

Local-Dynamic Allocation

```
void local_dyn_alloc(size_t len)
{
    int a[len];

    for (size_t i = 0; i < len; i++)
        a[i] = 2 * i;

    printf("{ ");
    for (size_t i = 0; i < len; i++)
        printf("%i ", a[i]);
    printf("}\n");
}
```

The size can be specified at runtime.

```
local_dyn_alloc(2);
local_dyn_alloc(6);
```

```
{ 0 2 }
{ 0 2 4 6 8 10 }
```

Two-Dimensional Arrays

Declaring Two-Dimensional Arrays

General syntax

```
<type> <identifier>[<length0>][<length1>];
```

Examples

```
int a[18][5];  // 18×5 elements of int type
char c[42][8]; // 42×8 elements of char type
float f[2][9]; // 2×9 elements of float type
```

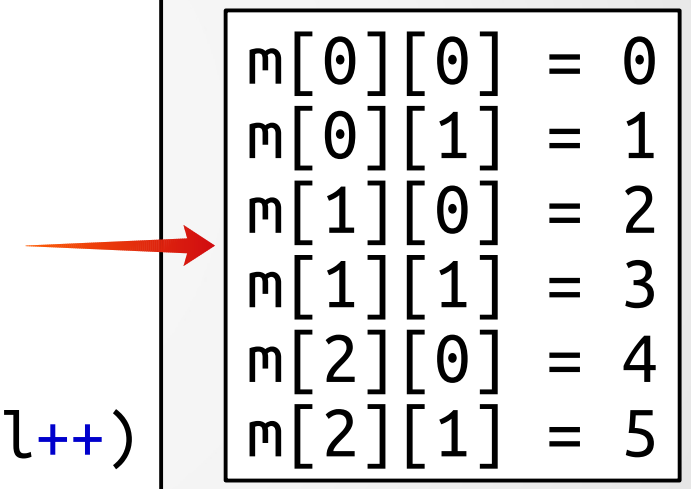
Not Initialized by Default

```c
int m[3][2];    // 3 rows, 2 columns

for (size_t row = 0; row < 3; row++)
    for (size_t col = 0; col < 2; col++)
        printf("m[%zu][%zu] = %i\n",
               row, col, m[row][col]);
```

```
m[0][0] = 0
m[0][1] = 0
m[1][0] = 1788432512
m[1][1] = 22089
m[2][0] = -1796671360
m[2][1] = 32765
```

Initializing Two-Dimensional Arrays

```
int m[3][2] =
{
    { 0, 1 },
    { 2, 3 },
    { 4, 5 },
};

for (size_t row = 0; row < 3; row++)
    for (size_t col = 0; col < 2; col++)
        printf("m[%zu][%zu] = %i\n",
               row, col, m[row][col]);
```



```
m[0][0] = 0
m[0][1] = 1
m[1][0] = 2
m[1][1] = 3
m[2][0] = 4
m[2][1] = 5
```

Initializing Two-Dimensional Arrays

Unset values are initialized to zero

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

=

```
int m[4][5] =
{
    { 1, 2, 0, 0, 0 },
    { 3, 4, 5, 6, 7 },
    { 6, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0 },
};
```

Initializing Two-Dimensional Arrays

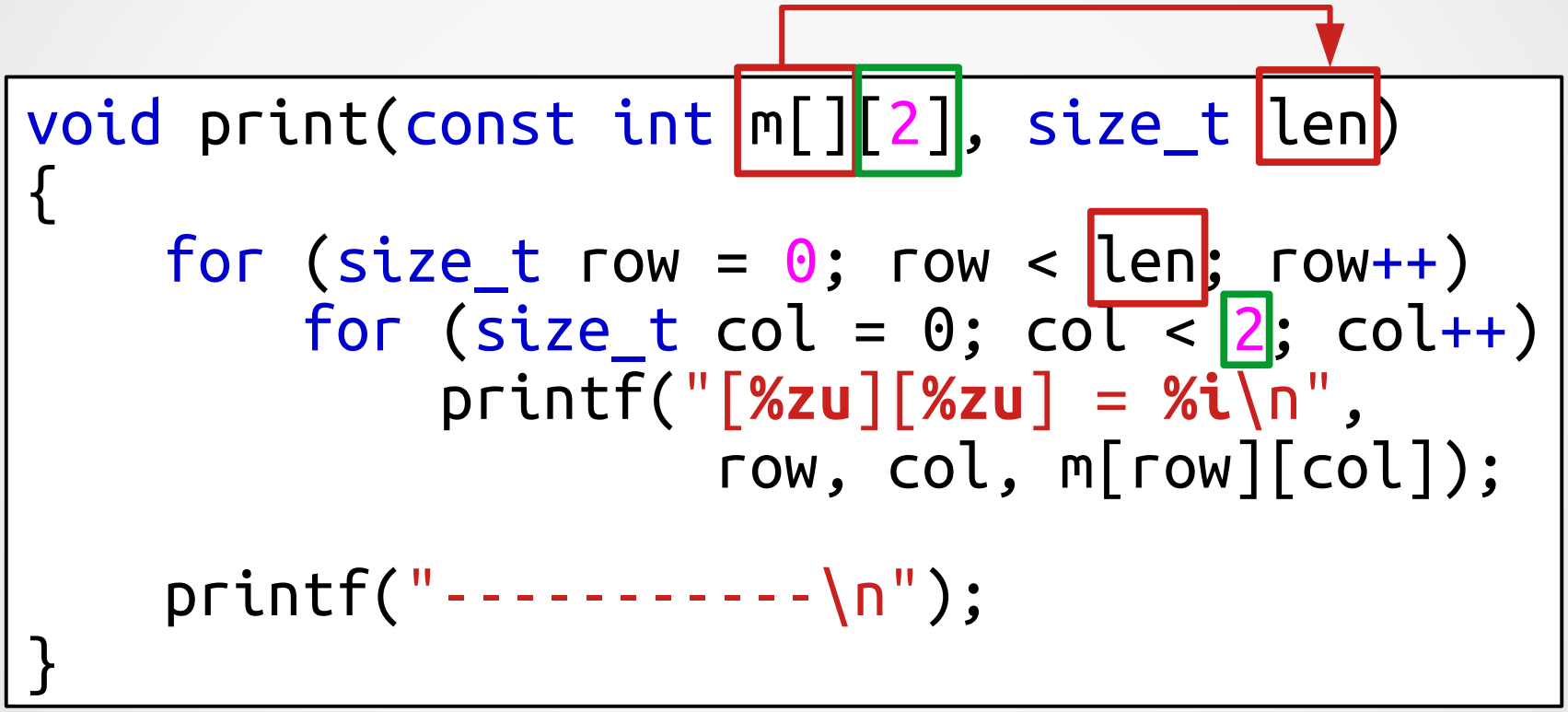
Only the first dimension can be removed.

```
int m[][2] =
{
    { 0, 1 },
    { 2, 3 },
    { 4, 5 },
};

for (size_t row = 0; row < 3; row++)
    for (size_t col = 0; col < 2; col++)
        printf("m[%zu][%zu] = %i\n",
               row, col, m[row][col]);
```

```
m[0][0] = 0
m[0][1] = 1
m[1][0] = 2
m[1][1] = 3
m[2][0] = 4
m[2][1] = 5
```

Two-Dimensional Arrays as Parameters

The first dimension is usually passed to the function.



```
void print(const int m[][2], size_t len)
{
    for (size_t row = 0; row < len; row++)
        for (size_t col = 0; col < 2; col++)
            printf("[%zu][%zu] = %i\n",
                   row, col, m[row][col]);

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

The other dimensions must be known at compile time.

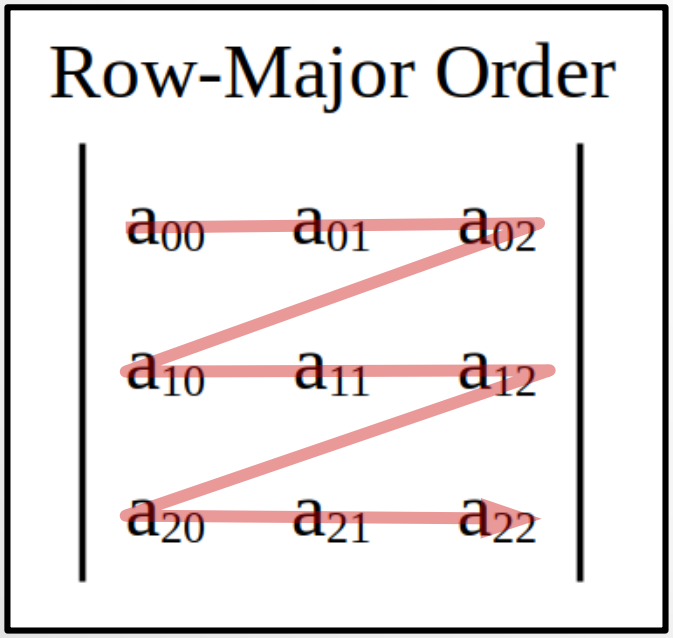
Two-Dimensional Arrays as Parameters

```
int m1[][2] =
{
    { 0, 1 },
    { 2, 3 },
};

int m2[][2] =
{
    { 0, 1 },
    { 2, 3 },
    { 4, 5 },
    { 6, 7 },
};

print(m1, 2);
print(m2, 4);
```

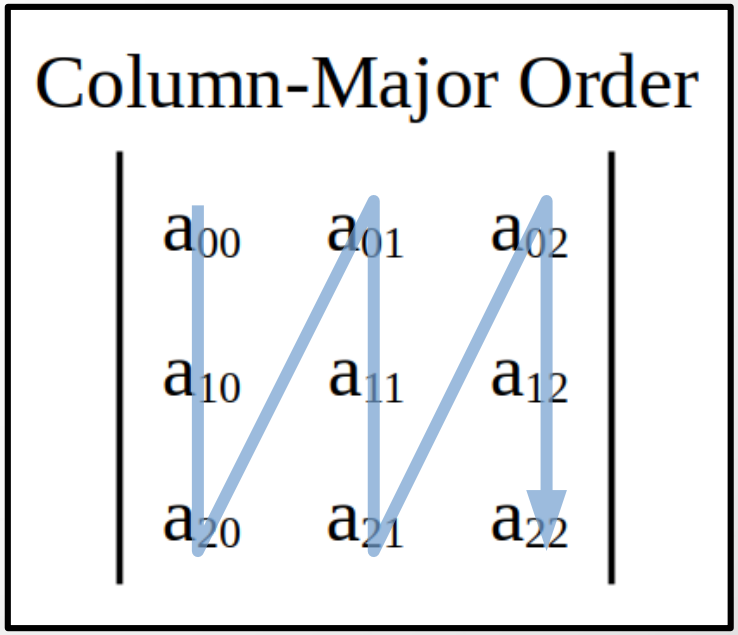
```
[0][0] = 0
[0][1] = 1
[1][0] = 2
[1][1] = 3
-----------
[0][0] = 0
[0][1] = 1
[1][0] = 2
[1][1] = 3
[2][0] = 4
[2][1] = 5
[3][0] = 6
[3][1] = 7
-----------
```

Row- and Column-Major Order

One-Dimensional Arrays Can Be Seen as Multidimensional Arrays

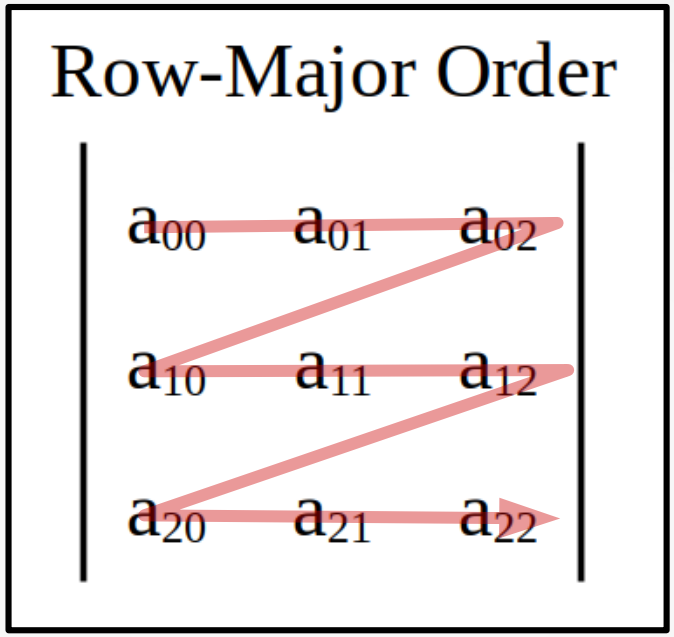


and



See: https://en.wikipedia.org/wiki/Row-_and_column-major_order

Example: Row-Major Order




```
a[i][j] = a[i * COLS + j]
```

COLS = Total number of columns

Example: Row-Major Order

```
void print(const int m[], size_t len)
{
    for (size_t row = 0; row < len; row++)
    {
        for (size_t col = 0; col < 2; col++)
        {
            size_t i = row * 2 + col;
            printf("[%zu][%zu] = %i\n", row, col, m[i]);
        }
    }

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

Example: Row-Major Order

```
int m1[] =
{
    0, 1,
    2, 3,
};

int m2[] =
{
    0, 1,
    2, 3,
    4, 5,
    6, 7,
};

print(m1, 2);
print(m2, 4);
```

```
[0][0] = 0
[0][1] = 1
[1][0] = 2
[1][1] = 3
-----------
[0][0] = 0
[0][1] = 1
[1][0] = 2
[1][1] = 3
[2][0] = 4
[2][1] = 5
[3][0] = 6
[3][1] = 7
-----------
```