

Arrays

CP:AMA Readings: 8.1–8.2, 9.3, 12.1–12.4

CHTP Readings: 6.1–6.7, 6.11

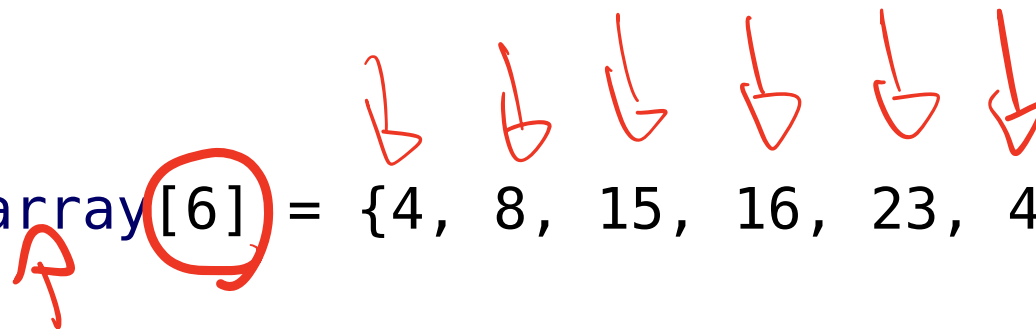
The primary goal of this section is to be able to use arrays and understand the internals of how they work.

Arrays

C only has two *built-in* types of “compound” data storage:

- structures (we’ll see in Section 7)
- *arrays*

int my_array[6] = {4, 8, 15, 16, 23, 42};



An array is a data structure that contains a **fixed number** of elements that all have the **same type**.

my_array[0] = 4;
`int my_array[6] = {4, 8, 15, 16, 23, 42};`

To define an array we must know the **length** of the array **in advance** (we address this limitation in Section 8).

Each individual value in the array is known as an *element*. To access an element, its *index* is required.

The first element of `my_array` is at index 0, and it is written as `my_array[0]`.

The second element is `my_array[1]` and the last is `my_array[5]`.

In computer science we often start counting at 0.

example: accessing array elements

Each individual array element can be used in an expression as if it was a variable.

In addition, the index of the array can be an expression.

```
int a[6] = {4, 8, 16, 16, 23, 42};  
int j = a[0]; // j is 4  
int *p = &a[j - 1]; // p points at a[3]  
a[2] = a[a[0]]; // a[2] is now 23  
++a[1]; // a[1] is now 9
```

example: arrays & iteration

Arrays and iteration are a powerful combination.

```
int a[6] = {4, 8, 15, 16, 23, 42};  
int sum = 0;
```

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

$(0+4) + 8 + 15 + \dots$
~~Float Avg = sum / 6.0;~~

a[0] = 4

a[1] = 8

a[2] = 15

a[3] = 16

a[4] = 23

a[5] = 42

sum = 108

Avg = ...

$(\text{float})\text{sum} / 6;$
 $\text{sum} / 6.0;$

Array initialization

Arrays can only be **initialized** with braces ({}).

```
int a[6] = {4, 8, 15, 16, 23, 42};
```

```
a = {0, 0, 0, 0, 0, 0}; // INVALID
```

```
a = ???; // INVALID
```

Once defined, the entire array cannot be mutated at once, and the length cannot change. Only *individual elements* can be mutated.

If there are not enough elements in the initialization braces, the remaining values are initialized to zero.

```
int b[5] = {1, 2, 3}; // b[3] & b[4] = 0
```

```
int c[5] = {0}; // c[0]...c[4] = 0
```

{1, 2, 3, 0, 0};

{0, 0, 0, 0, 0};

{};

int d[5];

local => default is garbage
global => default is 06

Character arrays can be initialized with double quotes (") for convenience.

The following two definitions are equivalent:

```
char a[3] = {'c', 'a', 't'};
```

```
char b[3] = "cat";
```

Handwritten red note: = {'c', 'a', 't', '\0'};

Handwritten red note: char c[3] = {'\0', '\0', '\0'};
 0, 0, 0;

In this example, **a** and **b** are character arrays and are **not** valid strings. This will be revisited in Section 6.

Like variables, the value of an uninitialized array depends on the scope of the array:

```
int a[5]; // uninitialized
```

- uninitialized *global* arrays are zero-filled.
- uninitialized *local* arrays are filled with arbitrary (“garbage”) values from the stack.

Array length

C does not explicitly keep track of the array **length** as part of the array data structure.

You must keep track of the array length separately.

To improve readability, the array length is often stored in a separate variable.

```
#define a_len 6
```

```
int a[6] = {4, 8, 15, 16, 23, 42};  
const int a_len = 6;
```

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

Another common way to specify the length of an array is to define a *macro*.

```
#define A_LEN 6
```

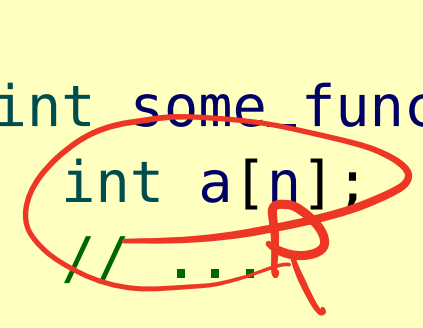
```
int main(void) {  
    int a[A_LEN] = {4, 8, 15, 16, 23, 42};  
    // ...  
}
```

In this example, `A_LEN` is not a constant or even a variable.

`A_LEN` is a *preprocessor macro*. Every occurrence of `A_LEN` in the code is replaced with 6 before the program is run.

C99 supports *Variable Length Arrays* (VLAs), where the length of an **uninitialized** local array can be specified by a variable (or a function parameter) not known in advance. The size of the stack frame is increased accordingly.

```
int some_function(int n) {  
    int a[n];           // length determined at run time  
    // ...  
}
```



In more recent versions of C this feature was made optional and it is not supported by all compilers, including Microsoft's C compiler. In Section 8 we see a better approach.

Theoretically, in some circumstances `sizeof` can be used to determine the length of an array.

```
int len = sizeof(a) / sizeof(a[0]);
```

The CP:AMA text uses this on occasion.

However, in practice this should be avoided, as the `sizeof` operator only properly reports the array size in very specific circumstances.

`[1, 2, 3, 4]`

`sizeof(int [4])`

`4 * 4 = 16`

`sizeof(int)`
4

`sizeof(int)`

4 or 8

1 or 2?

`f(a, len)`

Array size



The **length** of an array is the number of elements in the array.

The **size** of an array is the number of bytes it occupies in memory.

Size of

An array of k elements, each of size s , requires exactly $k \times s$ bytes.

In the C memory model, array elements are adjacent to each other.

Each element of an array is placed in memory immediately after the previous element.

If a is an integer array with six elements (`int a[6]`) the size of a is:

$$(6 \times \text{sizeof}(\text{int})) = 6 \times 4 = 24.$$

$\text{len} \times \text{sizeof}(\text{type}) = 9$

Not everyone uses the same terminology for length and size.

example: array in memory

```
int a[6] = {4, 8, 15, 16, 23, 42};  
printf("&a[0] = %p ... &a[5] = %p\n", &a[0], &a[5]);  
&a[0] = 0x5000 ... &a[5] = 0x5014
```

addresses	contents (4 bytes)
0x5000 ... 0x5003	4
0x5004 ... 0x5007	8
0x5008 ... 0x500B	15
0x500C ... 0x500F	16
0x5010 ... 0x5013	23
0x5014 ... 0x5017	42

The array identifier

An array does not have a “value” in C. When an array is used by itself in an expression, it evaluates (“decays”) to the **address** of the array (&a), which is also the address of the first element (&a[0]).

```
int a[6] = {4, 8, 15, 16, 23, 42};
```

Handwritten annotations:

- A blue circle around the `a` in the array declaration, with an arrow pointing to the first line of the summary.
- Summary of values and types:
 - `a` => 0x5000 (with a handwritten blue bracket and `int` next to it)
 - `&a` => 0x5000
 - `&a[0]` => 0x5000 (with a handwritten blue arrow pointing to `int []` below it)

Even though `a` and `&a` have the same *value*, they have different *types*, and cannot always be used interchangeably.

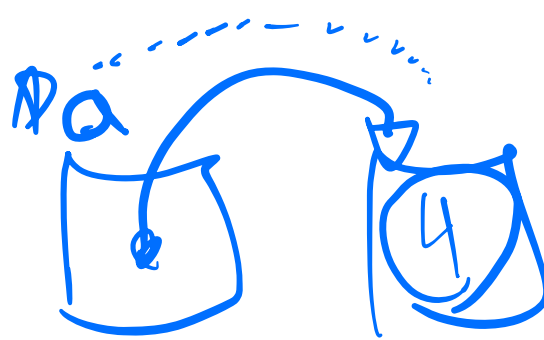
Dereferencing the array (`*a`) is equivalent to referencing the first element (`a[0]`).

```
int a[6] = {4, 8, 15, 16, 23, 42};
```

`a[0]` \Rightarrow 4

`*a` \Rightarrow 4

0x5000



`a[0]` \Rightarrow 0x5000 \Rightarrow 4

`*a`

`a[1]` \Rightarrow 0x5004 \Rightarrow 8

`*(a+1)` \rightarrow 0x5004 \rightarrow 8
`*(a+2)` \rightarrow 0x5008 \rightarrow ...

Passing arrays to functions

When an array is passed to a function, only the **address** of the array is copied into the stack frame.

This is more efficient than copying the entire array to the stack.

Typically, the length of the array is unknown to the function, and is a separate parameter.

There is no method of “enforcing” that the length passed to a function is valid.

Functions should **require** that the length is valid, but there is no way for a function to **assert** that requirement.

example: array parameters

// sum_array(a, len) returns the sum of elements in a
// requires: the array a is of length len

```
int sum_array(int a[], int len) {  
    int sum = 0;  
    for (int i = 0; i < len; ++i) {  
        sum += a[i];  
    }  
    return sum;  
}
```

~~size~~

```
int main(void) {  
    int my_array[6] = {4, 8, 15, 16, 23, 42};  
    assert(sum_array(my_array, 6) == 108);  
}
```

Note the parameter syntax: `int a[]`

and the calling syntax: `sum_array(my_array, 6)`.

example: “pretty” print an array

```
// print_array(a, len) pretty prints an array a with commas,  
// ending with a period  
// requires: the array length len > 0
```

```
void print_array(int a[], int len) {
```

```
    assert(len > 0);
```

```
    for (int i = 0; i < len; ++i) {
```

```
        if (i != 0) {
```

```
            printf(", ");
```

```
        }
```

```
        printf("%d", a[i]);
```

```
    }
```

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

```
}
```

```
int main(void) {
```

```
    int a[6] = {4, 8, 15, 16, 23, 42};
```

```
    print_array(a, 6);
```

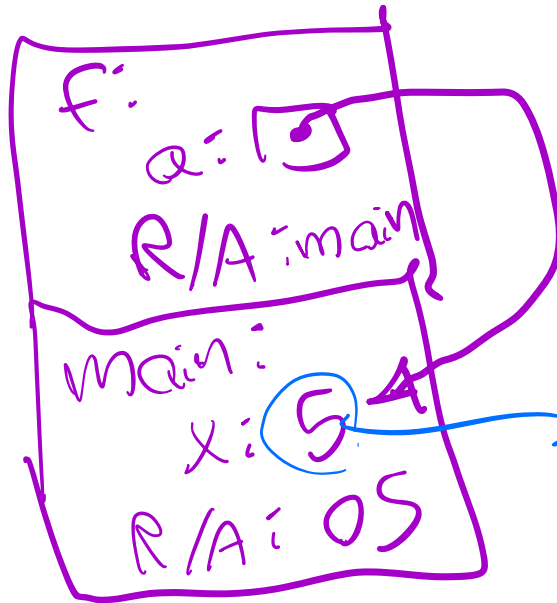
```
}
```

```
4, 8, 15, 16, 23, 42.
```

✓ X print
X error
(assert)
X memory
changes
✓ variable
changes

definition $\rightarrow f(\text{int } a)$ \rightarrow add 1 to a

main $\rightarrow f(\&x);$



$\rightarrow 6$
 $\rightarrow 5$
 $\rightarrow 7$

C allows you to specify the intended length of the array in the parameter, but it is **ignored**.

```
void calendar(int days_per_month[12]) {  
    // ...  
}
```

f(int a[], int len)

len

In this example, the 12 is ignored. The function may be passed an array of arbitrary length.

Similarly, some prefer to pass the length of the array first:

```
void f(int len, int a[len]) {  
    // ...  
}
```

*f(int a_len, int b_len,
int a[a_len],
int b[b_len])*

But since the [len] is ignored (and not enforced) it is more common to pass the array first.

Passing an address to a function allows the function to change (mutate) the contents at that address.

pass by ref.

```
// array_negate(a, len) negates all entries of a
void array_negate(int a[], int len) {
    for (int i = 0; i < len; ++i) {
        a[i] = -a[i];
    }
}
```

int a

{ 4, 5, -10 }
→ { -4, -5, 10 }

It's good style to use the `const` keyword to both prevent mutation and communicate that no mutation occurs.

```
int sum_array(const int a[], int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```

Pointer arithmetic

We have not yet discussed any *pointer arithmetic*.

C allows an integer to be added to a pointer, but the result may not be what you expect.

If p is a pointer, the value of $(p+1)$ depends on the type of the pointer p .

$0x5000 + 4$ for ints
 $(p+1)$ adds the `sizeof` whatever p points at.
`sizeof(type pp)`
int $\rightarrow 4$
char $\rightarrow 1$

int ϕ
char ϕ

According to the C standard, pointer arithmetic is only valid **within an array** (or a structure—to be covered in Section 7).

Pointer arithmetic rules

(array)

- When adding an integer i to a pointer p , the address computed by $(p + i)$ in C is given in “normal” arithmetic by:

$$p + i \Rightarrow p + (i \times \text{sizeof}(*p)).$$

- Subtracting an integer from a pointer ($p - i$) works in the same way.

- Mutable pointers can be incremented (or decremented).

$++p$ is equivalent to $p = p + 1$.

```
int a[4];  
int *p = &a;  
++p;  
...
```

char
 $\text{int } p[4] = \{ \};$

$$(p + 1) = p + 1 \times \text{sizeof}(\text{int})$$

char
 $= 0x5000 + 1 \times 4 = 0x5004$

- You cannot add two pointers. ~~X~~
- A pointer q can be subtracted from another pointer p if the pointers are the same type (point to the same type). The value of $(p - q)$ in C is given in “normal” arithmetic by:

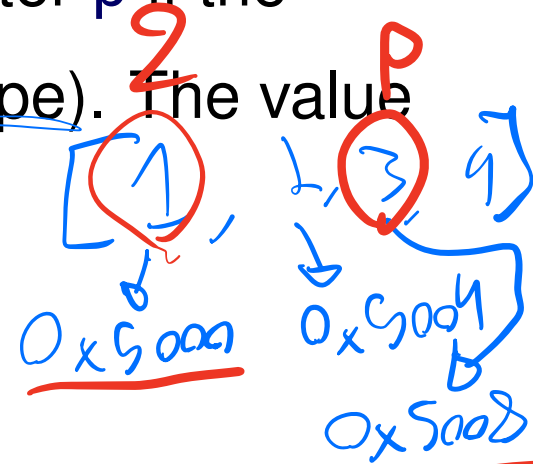
$$(p - q) / \text{sizeof}(*p).$$

In other words, if $p = q + i$ then $i = p - q$.

- Pointers (of the same type) can be compared with the comparison operators: $<$, $<=$, $==$, $!=$, $>=$, $>$ (e.g., `if (p < q) ...`).

`while (0 < (a + len) - p)`

`p + 4;`



`int q = &a[0];`
`int p = &a[2];`
 $(p - q)$
 $(0x5008 - 0x5000)$

$8 / \text{sizeof}(int)$

$\rightarrow 8 / 4 = 2$

Pointer arithmetic and arrays

Pointer arithmetic is useful when working with **arrays**.

Recall that for an array a , the value of a is the address of the first element ($\&a[0]$). $\text{a} = \&\text{a}$

Using pointer arithmetic, the address of the second element $\&a[1]$ is $(a + 1)$, and it can be referenced as $*(a + 1)$.

The array indexing syntax ($[]$) is an **operator** that performs *pointer arithmetic*.

$a[i]$ is equivalent to $*(a + i)$.

C does not perform any array “bounds checking”.

For a given array a of length l , C does not verify that $a[j]$ is valid ($0 \leq j < l$).

C simply “translates” $a[j]$ to $*(a + j)$, which may be outside the *bounds* of the array (e.g., $a[10000000]$ or $a[-1]$).

This is a common source of errors and bugs and a common criticism of C. Most recent languages have fixed this shortcoming and have “bounds checking” on arrays.

~~$a[-1] \rightarrow a[\text{len}-1]$~~

In array pointer notation, square brackets ([]) are not used, and all array elements are accessed through pointer arithmetic.

Point notation

```
int sum_array(const int * a, int len) {
    int sum = 0;
    for (const int * p = a; p < a + len; ++p) {
        sum += *p;
    }
    return sum;
}
```

array

Array Notation

for (int i = 0; i < len; i++) sum += a[i];

a = [0, 1, 2, 3]

p

a + len

a[0]

a[1]

Note that the above code behaves **identically** to the previously defined sum_array:

array notation

```
int sum_array(const int a[], int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```

current index

a[i]

a + len

a[1] < a[len]

*(a + i) = &a + i * sizeof(int)*

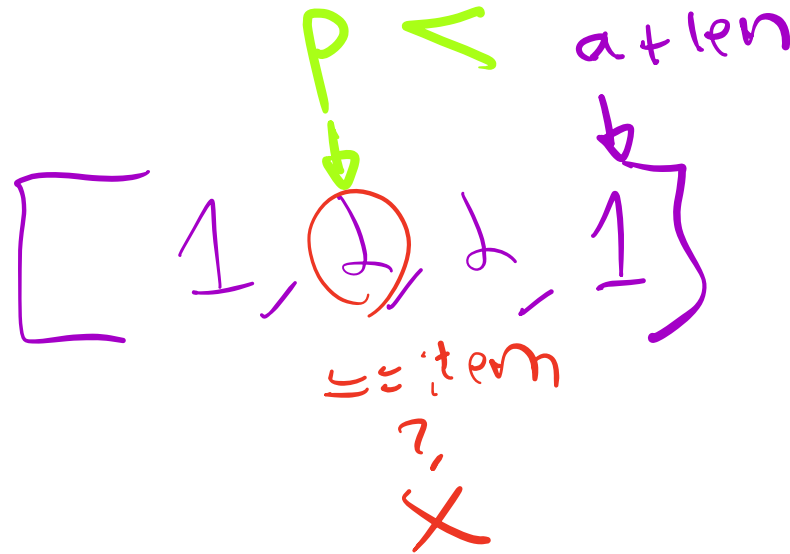
*= 0x5000 + len * 4*

another example: pointer notation

```
// count_match(item, a, len) counts the number of
// occurrences of item in the array a
int count_match(int item, const int * a, int len) {
    int count = 0;
    const int * p = a;
    while (p < a + len) {
        if (*p == item) {
            ++count;
        }
        ++p;
    }
    return count;
}
```

Handwritten annotations on the code:

- A blue arrow points to the `item` parameter in the function signature.
- A purple bracket highlights the initialization `const int * p = a;`.
- A green oval highlights the loop condition `p < a + len`.
- A red circle highlights the dereferenced pointer `*p` in the `if` statement.
- A blue circle highlights the increment `++p;` at the end of the loop.
- A red checkmark is next to the `++count;` line.
- A red `X` is next to the `++count;` line with the handwritten text "X SKIP" next to it.



Remember, for the variable:

`const int * p`

you can mutate `p` but you cannot mutate `*p`.

```
int count = 0;  
int i = 0;  
while (i < len) {  
    if (a[i] == item)  
        ++count;  
    i++;  
}
```

```
return count;
```

Array notation

The choice of notation (pointers or []) is a matter of style and context. You are expected to be comfortable with both.

C makes no distinction between the following two function declarations:

```
int array_function(int a[], int len) {...}    // a[]  
int array_function(int * a, int len) {...}    // *a
```

In *most* contexts, there is no practical difference between an array identifier and an immutable pointer.

The subtle differences between an array and a pointer are discussed at the end of Section 6.

Multi-dimensional data

$\text{int } a[5][2]$
 $\rightarrow \text{int } a[10];$

All of the arrays seen so far have been one-dimensional (1D) arrays.

We can represent multi-dimensional data by “mapping” the higher dimensions down to one.

For example, consider a 2D array with 2 rows and 3 columns.

$\begin{matrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{matrix}$ $\text{int } a[2][3] = \{ \{1, 2, 3\}, \{7, 8, 9\} \};$
 $r=1, c=1$

We can represent the data in a simple one-dimensional array.

$\text{int data}[6] = \{1, 2, 3, 7, 8, 9\};$

To access the entry in row r and column c , we simply access the element at $\text{data}[r*3 + c]$.

$a[r][c]$

$1*3 + 1 = 4$

In general, it would be $\text{data}[\text{row} * \text{NUMCOLS} + \text{col}]$.

Multi-dimensional arrays

C also supports multiple-dimensional arrays natively:

```
int two_d_array[2][3];  
int three_d_array[10][10][10];
```

When multi-dimensional arrays passed as parameters, the second (and higher) dimensions must be fixed.

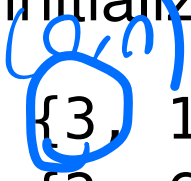
(e.g., `int function_2d(int a[][10], int numrows)`).

Internally, C represents a multi-dimensional array as a 1D array and performs “mapping” similar to the method described in the previous slide.

Initialization of 2D arrays

Multi-dimensional arrays can be initialized by using nested braces:

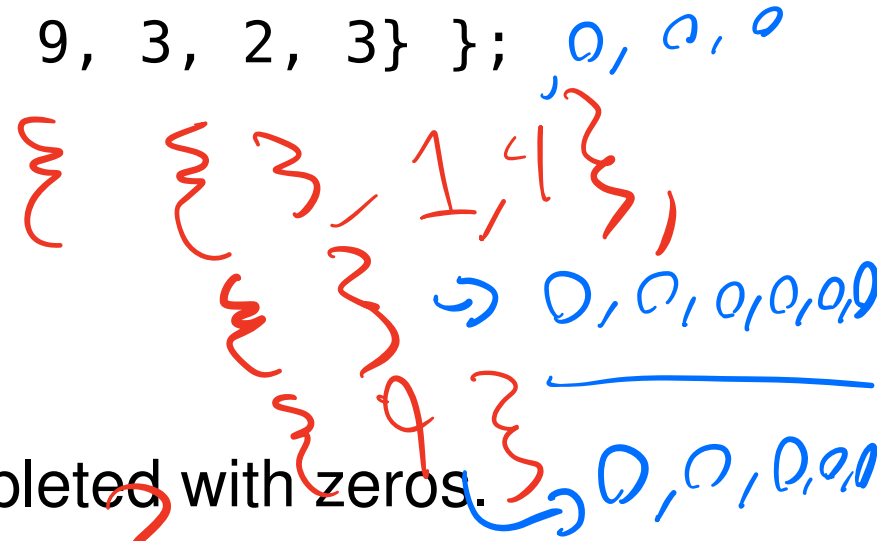
```
int two_d_array[3][6] = { {3, 1, 4, 1, 5, 9},  
                          {2, 6, 5, 3, 5, 8},  
                          {9, 7, 9, 3, 2, 3} };
```



two_d_array[0][0] => 3

two_d_array[1][0] => 2

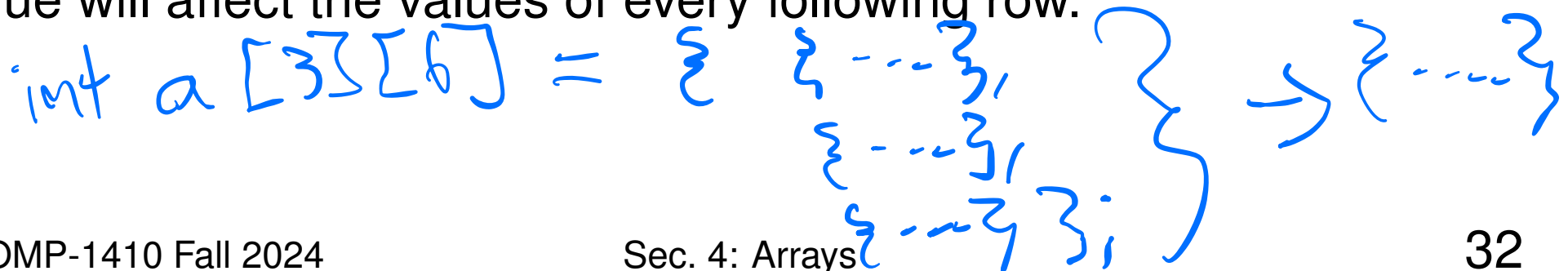
two_d_array[2][1] => 7



Any missing entries in a row will be completed with zeros.

You can omit the inner braces, but this is risky since then an extra value will affect the values of every following row.

```
int a[3][6] = { {---},  
                {---},  
                {---} };
```



$$\text{int } a[2][3] = \{ \begin{array}{l} \{0, 1, 2\}, \\ \{3, 4, 5\} \end{array} \};$$

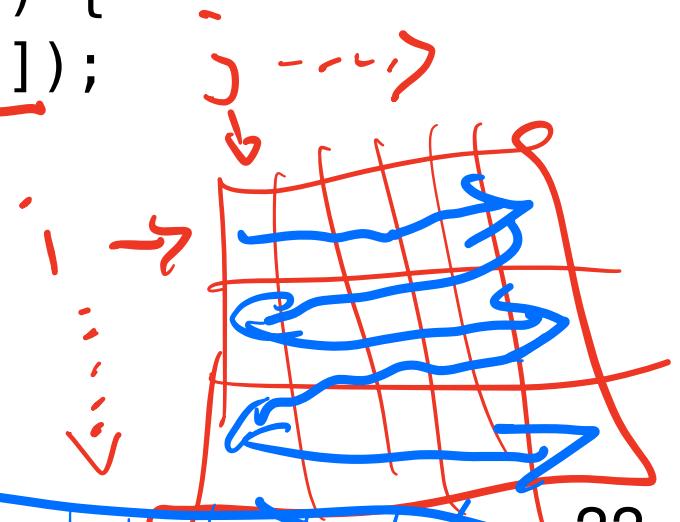
$$= \{0, 1, 2, 3, 4, 5\};$$

Printing the values in a multi-dimensional array

Write a program to print the values of the 2D array from the previous slide with the contents of each row printed on a separate line.

```
int main(void) {  
    int two_d_array[3][6] = { /* contents */ };  
    const int num_rows = 3;  
    const int num_cols = 6;  
  
    for(int i = 0; i < num_rows; ++i) {  
        for(int j = 0; j < num_cols; ++j) {  
            printf("%d ", two_d_array[i][j]);  
        }  
        printf("\n");  
    }  
}
```

(0,0)
↓
1





Goals of this Section

At the end of this section, you should be able to:

- define and initialize arrays, and use iteration to loop through arrays
- use pointer arithmetic
- explain how arrays are represented in the memory model, and how the array index operator (`[]`) uses pointer arithmetic to access array elements
- use both array index notation (`[]`) and array pointer notation and convert between the two
- represent multi-dimensional data in a single-dimensional array

~~$\&(p+i)$~~

1D array

~~$\&(\&(p+i)+j)$~~

row pos

col pos