Module 5: Arrays

Readings: CP:AMA 8.1, 9.3, 12.1, 12.2, 12.3

The primary goal of this section is to be able to use arrays.

Arrays

To define a variable that stores an array, we say:

- the **type** of each item in the array, followed by
- the **name** of the variable, then
- its **length** in square brackets.

Then we should initialize the array by writing = and values in curly braces:

```
int a[6] = {1, 1, 2, 5, 14, 42};
char word[5] = {'m', 'a', 't', 'h', '\0'};
```

- Once created, the length of an array cannot change.
- Every item in an array is of the **same type**.
  There is nothing like `(listof (anyof Int Str))`.

We can use arrays to do many of the things we used lists for in Racket, even though they work quite differently. We will create Racket-like lists in Module 10.

Arrays

We can extract a single element from an array using **indexing.**

After the name of an array we write square brackets around an integer called an **index.**

```
int a[6] = {1, 1, 2, 5, 14, 42};
 a[0] ⟹ 1
 a[1] ⟹ 1
 a[2] ⟹ 2
 a[3] ⟹ 5
 a[4] ⟹ 14
 a[5] ⟹ 42
```

```
char word[5] = {'m', 'a', 't', 'h', '\0'};
 word[0] ⟹ 'm'
 word[1] ⟹ 'a'
 word[2] ⟹ 't'
 word[3] ⟹ 'h'
 word[4] ⟹ '\0'
```

! Notice: the first element is element 0, not element 1.
The last element is numbered 1 less than the length.
The index of a element indicates how many elements there are **before** it.

Arrays

We can pass an array to a function.

The array inside a function is an *alias* of the array in the calling function.

In C, an array does not store its length; we need another parameter for that.

A function looks like this:

```
void print_array(int arr[], int len){
  for (int i = 0; i < len; ++i) {
    printf(" %d", arr[i]);
  }
  printf("\n");
}
```

We call it like this:

```
int main(void) {
  int jvj[5] = {2,4,6,0,1};
  print_array(jvj, 5);
}
```

We see:

```
 2 4 6 0 1
```

Exercise

Write a function `int int_sum(int a[], int len)` that returns the sum of the first `len` elements of `a`.
For example, `int_sum(arr, 5)` $\Longrightarrow$ `13`

Arrays

We can change the array, again treating an element like a variable:

```
int a[6] = {1, 1, 2, 5, 14, 42};
int j = a[3];        // j is 5
int *p = &a[j - 1]; // p points at a[4]
a[2] = a[a[3]];      // a[2] is 42
++a[3];              // a[3] is 6
```

Exercise

Write a function `void array_map(int (*f)(int), int a[], int len)` that mutates each value in `a`, transforming it by `f`. For example,

```
int sqr(int x)   { return x * x; }

int main(void) {
   int arr[4] = {2, 5, 14, 42};
   array_map(&sqr, arr, 4);
   assert(arr[0] == 4 && arr[1] == 25 && arr[2] == 196 && arr[3] == 1764);
}
```

Array initialization

We use curly braces `{}` to initialize array variables.

We cannot assign a new value to an array. We can only mutate individual elements.

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

a = {0, 0, 0, 0, 0, 0};  // INVALID
a[3] = 17;               // OK
```

If there are not enough elements in the braces, the rest are initialized to zero.

```
int b[5] = {3, 7, 31}; // equivalent to b[5] = {3, 7, 31, 0, 0};
int c[5] = {0};        // equivalent to c[5] = {0, 0, 0, 0, 0};
int d[5] = {};         // technically forbidden until C23, though gcc allows it....
```

Exercise

Write code to create an array `thousand` containing the numbers $[0, 1, 2, \ldots, 999]$.

```
assert(int_sum(thousand, 1000) == 499500);
```

Array initialization

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

Consider the following definitions:

```
char a[4] = {'c', 'a', 't', '\0'}; // specified all 4 elements
char b[4] = {'c', 'a', 't'};       // equivalent to {'c', 'a', 't', '\0'}
char c[4] = "cat";                 // equivalent to {'c', 'a', 't', '\0'}
char d[3] = "cat";                 // equivalent to {'c', 'a', 't'}
char e[3] = "012";                 // equivalent to {'0', '1', '2'}
                          // also to {48, 49, 50}, since 48 == '0', 49 == '1', 50 == '2'
```

An array of `char` that **contains a zero** is how we store a **string** in C.
Here `a`, `b`, and `c` are strings. But `d` and `e` are not: each is missing a zero.
We will work with strings more in Module 8.

Extension: uninitialized arrays

It can save a *tiny* amount of time to not initialize an array, but if you do it wrong, you can end up working on garbage data.

> ! In this course, you should always initialize your arrays.

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. (But we won't use them in this course.)
- 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.

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

Array length

It might seem better to use a constant to specify the length of an array.

```
const int a_len = 6;
int a[a_len] = {4, 8, 15, 16, 23, 42};  // NOT IN CS136
```

This appears to be “better style”, but it’s not correct C.

The syntax to do this properly is outside of the scope of this course.

! In this course, always define arrays using numbers.
It is okay to have these “magic numbers” appear in your assignments.

```
int a[6] = {4, 8, 15, 16, 23, 42};  // This is OK.
```

Extension: Array Length

A preferred syntax 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.

Extension: Array Length

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.

```c
int some_function(int n) {
  int m = n * 2;
  int a[m];      // length determined at run time
  // ...
}
```

This approach has many disadvantages. **You are not allowed to use VLAs in this course.** In Module 9 we see a better approach.

Extension: Array Length

Theoretically, in some circumstances you could use `sizeof` to determine the length of an array.

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

The CP:AMA textbook uses this on occasion.

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

Array Memory Model: `char` arrays

In the C memory model, each array element sits immediately after the previous element.

Each `char` occupies one cell of memory, so the address of each element is one more than the item before it.

When we run this:

```c
char plato[5] = {4, 6, 8, 12, 20};
printf("%p\n", &plato[0]);
printf("%p\n", &plato[1]);
printf("%p\n", &plato[2]);
printf("%p\n", &plato[3]);
printf("%p\n", &plato[4]);
```

We see this:

```
0x7ffc02397cd0
0x7ffc02397cd1
0x7ffc02397cd2
0x7ffc02397cd3
0x7ffc02397cd4
```

Pointer arithmetic

If we add `i` to a pointer, this moves over by `i` elements in the array.

When we run this:

```
char plato[5] = {4, 6, 8, 12, 20};
char *p = &plato[0];
for(int i = 0; i < 5; ++i) {
  printf("p+%d = %p, *(p+%d) = %d\n",
         i, p + i, i, *(p + i));
}
```

We see this:

```
p+0 = 0x7ffc02397cd0, *(p+0) = 4
p+1 = 0x7ffc02397cd1, *(p+1) = 6
p+2 = 0x7ffc02397cd2, *(p+2) = 8
p+3 = 0x7ffc02397cd3, *(p+3) = 12
p+4 = 0x7ffc02397cd4, *(p+4) = 20
```

Exercise

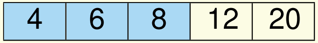
Write a function `int char_sum(char *p, int n)`. It takes a pointer to a `char` inside an array, and returns the sum of the `n` items starting from that location. For example,

```
assert(char_sum(&plato[0], 3) == 4 + 6 + 8);
assert(char_sum(&plato[2], 3) == 8 + 12 + 20);
```

Do not use array notation, `p[i]`. Instead use pointer arithmetic.

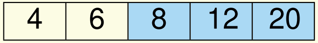
Pointer arithmetic

When we call `char_sum(&plato[0], 3)`, it adds up 3 values:



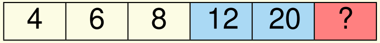
$\Rightarrow 18$

And when we call `char_sum(&plato[2], 3)`, it adds up 3 different values:



$\Rightarrow 40$

What happens if we call `char_sum(&plato[3], 3)`?



$\Rightarrow$?

!

According to the C standard, pointer arithmetic is only valid **within an array**.

If you go off the end of an array, what happens is **undefined**:

- You might find garbage.
- You might get a crash.
- You might get something else.

Array Pointer Notation

We have a choice of how to write code using arrays: either using array notation, using `[]`, or pointer arithmetic.

Here is equivalent code written in both:

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;
}
```

Pointer 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 Pointer 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.

An array is a lot like a pointer

We have now written examples of functions that work with arrays in two different ways:

1. `int_sum(int a[], int len)`, which we called like `int_sum(jvj, 5)`.
2. `char_sum(char *p, int n)`, which we called like `char_sum(&plato[0], 3)`.

In the first case, “`a` is an array”. In the second case, `p` is a pointer.

We understand the second case. What is happens in the first?

An array is a lot like a pointer

The **value** of an array (a) is the same as the **address** of the array (&a), which is also the address of the first element (&a[0]).

In a function definition, `char a[]` is treated identically to `char *a`.

When we run:

```
int a[6] = {101, 109, 149, 181, 269, 389};
printf("%p\n", a);
printf("%p\n", &a[0]);
printf("%p\n", &a);
```

We see:

```
0x7ffce1837e20
0x7ffce1837e20
0x7ffce1837e20
```

a, &a[0], and &a have the same *value*, but different types.
They cannot be used interchangeably.

Exercise

Write functions f and g, that can be called like f(a) and g(&a[0]).
Challenge: try to write a function h that can be called like h(&a). (It is possible.)

An array is a lot like a pointer

Since the value of an array is the address of the first element, dereferencing the array (`*a`) is equivalent to referencing the first element (`a[0]`).

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

trace_int(a[0]);  → 4
trace_int(*a);    → 4
```

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

! Since the value of an array is its address, **do** assert that every array is not `NULL`.

Functions should `require` that the length is valid, but there is no way to `assert` it.
An array does not store its own length.

More Pointer Arithmetic

Earlier we saw that when we add `i` to a `char *` inside an array, it moves over by `i` elements:

```
p+0 = 0x7ffc02397cd0, *(p+0) = 4
p+1 = 0x7ffc02397cd1, *(p+1) = 6
p+2 = 0x7ffc02397cd2, *(p+2) = 8
```

What happens if we add `i` to an `int *` inside an array of `int`?

We run similar code:

```
int vals[5] = {239, 241, 251, 257, 263};
int *p = &vals[0];
for(int i = 0; i < 5; ++i) {
  printf("p+%d = %p, *(p+%d) = %d\n",
         i, p + i, i, *(p + i));
}
```

And get *similar* output:

```
p+0 = 0x7fff3a39cc70, *(p+0) = 239
p+1 = 0x7fff3a39cc74, *(p+1) = 241
p+2 = 0x7fff3a39cc78, *(p+2) = 251
p+3 = 0x7fff3a39cc7c, *(p+3) = 257
p+4 = 0x7fff3a39cc80, *(p+4) = 263
```

Adding `i` still moves over by `i` elements.
But the pointer changes by **4**: the **size** of an `int` on our architecture.

The `sizeof` operator

The `sizeof` operator produces the number of `char` values that would be required to store a type or variable. `sizeof` looks like a function, but it is an operator.

By definition, `sizeof(char)` ⇒ `1`.

```
int n = 0;
sizeof(int);  ⇒ 4
sizeof(n);    ⇒ 4
sizeof(char); ⇒ 1
sizeof(bool); ⇒ 1
```

```
// Some other types not used in CS136:
sizeof(short int);      ⇒ 2
sizeof(long long int); ⇒ 8
```

The size of `int` depends on the architecture.

Items in an array will always be spaced by the `sizeof` a single item.

When we run:

```
short int s[3] = {1024, 234, 2048};
printf("%p\n", &s[0]);
printf("%p\n", &s[1]);
printf("%p\n", &s[2]);
```

We see:

```
0x7ffdecfc4e00
0x7ffdecfc4e02
0x7ffdecfc4e04
```

The `sizeof` operator

In C, the size of an `int` depends on the machine (processor) and/or the operating system that it is running on.
Every processor has a natural “word size” (e.g. 32-bit, 64-bit). Historically, the size of an `int` was the word size, but most modern systems use a 32-bit `int` to improve compatibility.
In C99, the `inttypes` module defines many types (e.g. `int32_t`, `int16_t`) that specify *exactly* how many bits (bytes) to use.
In this course, you should only use `int`, and there are always 32 bits in an `int`.

The sizeof operator

We can make an array that contains pretty much anything.

An array of structs:

```
struct quaternion { int a; int b; int c; int d; };
int main(void) {
  struct quaternion qarr[2] = {{1,2,3,4}, {5,6,7,8}};
  printf("%d\n", qarr[1].c); // → 7
}
```

An array of `int *`:

```
int main(void) {
  int a = 5; int b = 8; int c = 13;
  int *iarr[3] = {&a, &b, &c};
  printf("%d\n", *iarr[0]); // → 5
  printf("%d\n", *iarr[1]); // → 8
  printf("%d\n", *iarr[2]); // → 13
}
```

The `sizeof` operator

On any particular architecture, a pointer is the same size, regardless of what it points to:

`sizeof(char)` ⟹ 1	`sizeof(char *)` ⟹ 8
`sizeof(int)` ⟹ 4	`sizeof(int *)` ⟹ 8
`sizeof(quaternion)` ⟹ 16	`sizeof(quaternion *)` ⟹ 8
	`sizeof(&main)` ⟹ 8

The `sizeof` operator

Using `sizeof` on an array can be misleading. **Don't do it.**

In the scope where the array is defined, the compiler knows how big it is. But elsewhere, all that is available is a pointer.

```
void print_size(int func_arr[]) {
  trace_int(sizeof(func_arr)); // just a pointer → 8 bytes
}

int main(void) {
  int main_arr[7] = {1,2,3,4,5,6,7};
  trace_int(sizeof(main_arr)); // 4 bytes per element ∗ 7 elements → 28 bytes
  print_size(main_arr);
}
```

It's possible to say the size of an array in the function definition, like: `void print_size(int func_arr[7])`. But this does not actually do anything. Even if you do, `sizeof(func_arr)` will still be just the size of a pointer.

Pointer Arithmetic

Look at this code:

```
int main(void){
  int arr[5] = {2, 4, 6, 0, 1};
  int *p = &arr[4];
  int *q = &arr[2];
  int d = p - q;
  printf("%d\n", d);
}
```

Exercise

Discuss with your neighbours: what do you think “should” be printed when we run it?

By standard arithmetic, if $d = p - q$, then $p = q + d$.

And we know that `q + d` means “move over this many slots in the array”.

Since `p` is 2 slots over from `q`, it should be that `d` is 2. And it is so.

Subtraction of pointers is valid; the result is an integer “distance”.

Sorting Algorithms

There are many interesting sorting algorithms.

By tradition, we look here only at some extremely old algorithms.

Here we are going to take a close look at **selection sort**, the origin of which is lost in time.

The notes also include **insertion sort**, also very old, and **quicksort**, developed much more recently, in 1959.

There are plenty of newer algorithms: a newer one is Timsort, released in 2002, and used in Python.

In the real world, **you should almost certainly not write your own sorting algorithm**; use a library instead. But they can be fun to study.

Selection sort

In *selection sort*, we find the smallest item in the list, and swap it into the first position; then find the next-smallest item in the list, and swap it into the next position; and so on.

Sorting Out Sorting is an ancient movie that demonstrates "straight" selection sort, along with several other sorting algorithms. You might want to watch it later.

Watch *Sorting Out Sorting* about Selection Sort

Selection sort

Exercise

Using the `swap` function below as a helper, write a function `void selection_sort(int a[], int len)` that mutates `a` to be in increasing order.

It might help to use `print_array`, each time through the outer loop. Then from here:

```
int arr[] = {15, 6, 7, 4, -1, -2, 12};
selection_sort(arr, 7);
```

you should see:

```
15 6 7 4 -1 -2 12
-2 6 7 4 -1 15 12
-2 -1 7 4 6 15 12
-2 -1 4 7 6 15 12
-2 -1 4 6 7 15 12
-2 -1 4 6 7 15 12
-2 -1 4 6 7 12 15
```

```
// swap(a, b) Swap contents of a and b.
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Insertion sort

In *Insertion sort*, we consider the first element to be a sorted sequence (of length one).

We then “insert” the second element into the existing sequence into the correct position, and then the third element, and so on.

For each iteration of *Insertion sort*, the first `i` elements are sorted. We then “insert” the element `a[i]` into the correct position, moving all of the elements greater than `a[i]` one to the right to “make room” for `a[i]`.

Insertion sort

Consider an iteration of insertion sort (`i = 3`), where the first `i` (`3`) elements have been sorted. We want to *insert* the element at `a[i]` into the correct position.

3	7	8	5	4	9	0

We continue to *swap* the element with the previous element until it reaches the correct position.

3	7	5	8	4	9	0

3	5	7	8	4	9	0

Once it is in the correct position, we start on the next element.

3	5	7	8	4	9	0

Insertion sort

```c
void insertion_sort(int a[], int len) {
  for (int i = 1; i < len; ++i) {
    for (int j = i; j > 0 && a[j - 1] > a[j]; --j) {
      swap(&a[j], &a[j - 1]);
    }
  }
}

// Notes:
//   i:   loops from 1 ... len-1 and represents the
//        "next" element to be replaced
//   j:   loops from i ... 1 and is "inserting"
//        the element that was at a[i] until it
//        reaches the correct position
```

Quicksort

Quicksort is an example of a “divide & conquer“ algorithm.

First, an element is selected as a “pivot” element.

The list is then **partitioned** (*divided*) into two sub-groups: elements *less than* (or equal to) the pivot and those *greater than* the pivot.

Finally, each sub-group is then sorted (*conquered*).

Quicksort

We have already seen the implementation of quicksort in Racket.

```
(define (quick-sort lon)
  (cond [(empty? lon) empty]
    [else (define pivot (first lon))
          (define less (filter (lambda (x)
                           (<= x pivot)) (rest lon)))
          (define greater (filter (lambda (x)
                              (> x pivot)) (rest lon)))
          (append (quick-sort less)
                  (list pivot)
                  (quick-sort greater))]))
```

For simplicity, we select the first element as the “pivot”. A more in-depth discussion of pivot selection occurs in CS 240.

Quicksort

In our C implementation of quicksort, we:

- select the first element of the array as our "pivot"
- move all elements that are larger than the pivot to the back of the array
- move ("swap") the pivot into the correct position
- recursively sort the "smaller than" sub-array and the "larger than" sub-array

The core quicksort function `quick_sort_range` has parameters for the range of elements (`first` and `last`) to be sorted, so a wrapper function is required.

Quicksort

Watch *Sorting Out Sorting* about Selection Sort

Quicksort

```
void quick_sort_range(int a[], int first, int last) {
  if (last <= first) return;  // length is <= 1

  int pivot = a[first];       // first element is the pivot
  int pos = last;             // where to put next larger

  for (int i = last; i > first; --i) {
    if (a[i] > pivot) {
      swap(&a[pos], &a[i]);
      --pos;
    }
  }
  swap(&a[first], &a[pos]);   // put pivot in correct place
  quick_sort_range(a, first, pos - 1);
  quick_sort_range(a, pos + 1, last);
}

void quick_sort(int a[], int len) {
  quick_sort_range(a, 0, len - 1);
}
```

Binary search

Quick! Is 7256 in this list?

$$[5421, 4448, 8635, 2444, 3711, 3477, 4367, 1793, 5484, 2508, 9668, 3643, 4257, 9226,$$
$$6525, 2511, 6087, 6259, 3256, 1205, 7471, 4749, 7247, 7699, 5423, 4845, 4860, 6055]$$

Now I have similar data, but sorted in increasing order. Is 7256 in this list?

$$[1041, 1952, 2385, 4743, 4896, 5008, 5081, 5417, 5555, 5612, 5896, 5960, 6278, 6294,$$
$$6391, 6864, 7196, 7339, 7428, 7451, 7624, 7741, 8240, 8461, 9098, 9164, 9408, 9607]$$

In the first case, you have to look at each item, one by one.
You can't be sure until you look at every item.

In the second case, as soon as you look at the 6391 you can see that it can't be in first row.
You can then quickly discard the second half of the second row, and so on.

This is called *binary search*, because each step divides the size of the problem by 2.

Binary search

Exercise

Write a function `int binary_search(const int target, const int a[], const int len)`.
It requires (but does not `assert`) that `a` is sorted in increasing order, and returns the index where `a` contains an item, or `-1` if item does not appear in `a`.

Hint

Use variables `left` and `right` to keep track of the leftmost and rightmost location that `target` might appear.
Each time, find the middle; check if `target` is there.
If `target` is not in the middle, update either `left` or `right`, discarding the middle slot.
Continue this until you find `target`, or you run out of slots.

Multi-dimensional data

We can “flatten” a higher dimensional array to one dimension.

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

1	2	3
4	5	6

We can represent the data in a one-dimensional array: `int data[6] = { 1, 2, 3 , 7, 8, 9 };`

We need to separately store the “width” and “height” of the data; the length of the array must be (at least) as long as their product.

Exercise

Write a function `void print_matrix(const int a[], const int width, const int height)` that prints all items from `a`, viewed as an 2-dimensional array of given `width` and `height`.

Then if we define an array:

```
int m[12] =
  { 1,0,0,0,1,1,0,0,1,2,1,0 };
```

We can print like this →

```
print_matrix(m, 4, 3);
[ 1 0 0 0 ]
[ 1 1 0 0 ]
[ 1 2 1 0 ]
```

```
print_matrix(m, 3, 4);
[ 1 0 0 ]
[ 0 1 1 ]
[ 0 0 1 ]
[ 2 1 0 ]
```

Multi-dimensional data

C supports multiple-dimension arrays, but they are not covered in this course.

```
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.

See CP:AMA sections 8.2 & 12.4 for more details.

Goals of this Section

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

- define and initialize arrays
- 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