C Pointers and Arrays

Table of contents

Arrays 1
Pointers 2
Arrays are pointers to their first element 3
Pointer arithmetic 3
Traversing an array 4
Strings, briefly 5
See also 5

Arrays, pointers, and the fact that connects them: **an array name is the address of its first element**, and **pointer arithmetic counts in elements, not cells**. Both are made concrete in the AVR's 8-bit data memory (memory-models[1]), where an `int` is two cells and so `a + 5` is ten cells past `a`. First met in 2026-09-10-more-c-programming-on-the-avr[2]; the language basics are in c-language-basics[3].

Arrays

Declaring: `type variable_name[size];`

```
char message[16];
int  values[10];
uint8_t seven_seg[10] = { 63,6,91,79,102,109,125,7,127,111 };   /* with initial
↪  values */
```

Accessing: `variable_name[index]`, with **index = 0 ... size−1** — **zero-based indexing**. A 365-element array has elements `[0]` through `[364]`; `[365]` is not one of them.

[1] courses/csse2010/memory-models.pdf
[2] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[3] courses/csse2010/c-language-basics.pdf

```
message[0] = 'c';
values[9] = values[8]++;        /* copies the OLD values[8], then increments it */
```

The declaration reserves `size` elements of the type **contiguously** — one after another with nothing between — which is what makes both the indexing and the pointer arithmetic below work. The elements are ordinary variables of the element type; `values[3]` is an `int` like any other.

C does not check the index. `values[10]` and `values[-1]` compile, and at run time they read or write whichever cell happens to sit past the end or before the start. The `seven_seg[digit]` lookup is guarded with `if(digit < 10)` for exactly this reason (c-control-flow[4]).

Pointers

A pointer variable stores a memory address. Declaration: `type *variable_name;` — *variable_name* is a pointer to something of the given *type*.

```
char a, b;
char *ptr;          /* or all on one line: char a,b,*ptr; */

ptr = &a;           /* & - address-of: ptr now holds a's address  */
b = *ptr;           /* * - indirection: the value at that address */
```

The two operators are inverses:

	From	To	
`&a`	a variable	its address	*address-of* — makes a pointer
`*ptr`	an address	the value stored there	*indirection* — follows a pointer

In memory, say `a` holds 8 at address 400, `b` is at 434, and `ptr` is at 576 holding **400** — an address, not a value (drawn in 2026-09-10-more-c-programming-on-the-avr[5]). `ptr` is a variable like the others, with its own cell; only its *contents* are special. On the AVR a data address is 16 bits (memory-models[6]), so a pointer is two cells whatever it points at.

`*ptr` can also be assigned to: `*ptr = 0;` writes through the pointer into whatever it points at. That is how the pointer traversal below clears an array.

[4] courses/csse2010/c-control-flow.pdf
[5] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[6] courses/csse2010/memory-models.pdf

The `*` in a declaration and the `*` in an expression are different things. `char *ptr;` declares `ptr` as a pointer; `*ptr` later *uses* it. The declaration form reads as *the thing `*ptr` is a `char`*.

The hardware view

A pointer is the **indirect addressing mode** of addressing-modes[7] as a language feature. `ptr` plays the part of the X, Y or Z register pair, `*ptr` is `ld rd, X`, and `*ptr = v` is `st X, rr`. `ptr++` is the post-increment form `ld rd, X+` that the instruction set provides precisely because walking a table is so common. The compiler chooses which of X, Y, Z it uses.

Arrays are pointers to their first element

An array name can be treated as a pointer to the first element — the address of the first element.

```
int a[10];
int *ptr;

/* the following are the same */
ptr = a;
ptr = &a[0];
```

So `a` on its own is an address, `a[0]` is the `int` there, and `*a` is the same `int`. In general **`a[i]` is `*(a + i)`** — the bracket notation is shorthand for pointer arithmetic, which is why the arithmetic has the rule it has.

Pointer arithmetic

Addition and subtraction on pointers work in multiples of the size of the object pointed to.

```
int a[10];
int *ptr;

/* the following are the same */
ptr = a+5;
ptr = &a[5];
```

[7] courses/csse2010/addressing-modes.pdf

If `a` starts at 300 and an `int` is 16 bits — two 8-bit cells — then `a + 5` is **310**, not 305: five elements on, two cells each. The sum is `300 + 2·5 = 310`, not `300 + 5 = 305`. The array is drawn in memory in 2026-09-10-more-c-programming-on-the-avr[8].

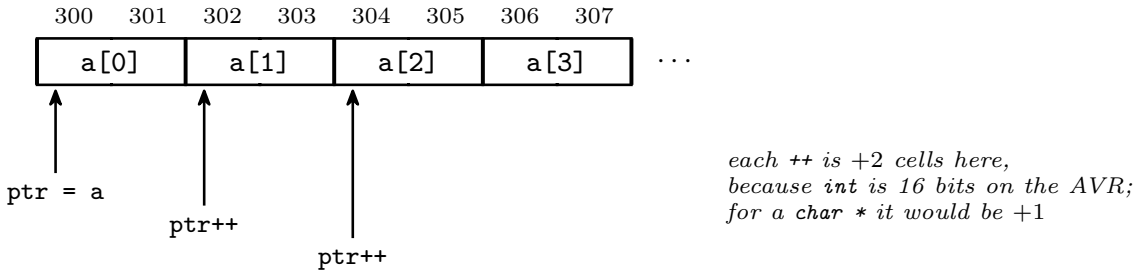


Figure 1: `ptr++` on a pointer to `int`, on the AVR. Each step moves the pointer by one *element*, which is two 8-bit cells, so the addresses it visits are 300, 302, 304, … and it lands on every element exactly once. The same `ptr++` on a `char` pointer would step by one cell. The compiler multiplies the step by the element size; the source says `++` either way. The pointer is the traversal loop's, drawn at three successive steps.

The consequences:

- **The same expression means different addresses for different types.** `a + 5` is 310 for `int a[]` and 305 for `char a[]`. The type is part of the arithmetic.
- **`ptr - a` counts elements**, not cells: for `ptr = &a[5]` it is 5.
- **Comparison works in addresses.** `ptr < a + 10` is true while `ptr` is inside the array.
- `int` is 16 bits *on the AVR*; on a 32-bit machine `a + 5` would be twenty cells on. The rule is stated in terms of *the size of the object* because the number is machine-dependent (c-language-basics[9]).

Traversing an array

Two ways to clear `int a[10]`:

```
/* using an index */
int a[10];
int index;
for(index=0; index<10; index++) {
    a[index] = 0;
}
```

[8] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf
[9] courses/csse2010/c-language-basics.pdf

```c
/* using a pointer */
int a[10], *ptr;
for(ptr=a; ptr < a+10; ptr++) {
    *ptr = 0;
}
```

Adding one to an array pointer makes it point to the next element in the array. The two loops visit the same ten cells in the same order. The index loop computes `a + index` afresh each pass; the pointer loop keeps that address in `ptr` and steps it. The bound is the same in both — the address one past the last element, tested with `<` so it is never touched. See c-control-flow[10] for the `for` itself.

The pointer form is closer to what the compiler emits either way — a register pair holding an address, an `st X+, r` in the body — which is why C has it.

Strings, briefly

A string constant `"abc"` (c-language-basics[11]) is an array of `char` with a terminating zero byte after the last character, so `"abc"` occupies four cells. `char message[16];` therefore holds at most fifteen characters, though `message[16]` looks like room for sixteen. The terminator is standard C.

See also

- c-language-basics[12] · c-operators[13] · c-control-flow[14] · c-preprocessor-and-avr-headers[15]
- memory-models[16] — 8-bit data cells, 16-bit addresses, alignment
- addressing-modes[17] — indirect addressing, which a pointer is
- avr-instructions[18] — `ld`/`st` with X, Y, Z and their post-increment forms
- seven-segment-display[19] — the `seven_seg[10]` lookup table
- 2026-09-10-more-c-programming-on-the-avr[20] — Lecture 14

[10] courses/csse2010/c-control-flow.pdf
[11] courses/csse2010/c-language-basics.pdf
[12] courses/csse2010/c-language-basics.pdf
[13] courses/csse2010/c-operators.pdf
[14] courses/csse2010/c-control-flow.pdf
[15] courses/csse2010/c-preprocessor-and-avr-headers.pdf
[16] courses/csse2010/memory-models.pdf
[17] courses/csse2010/addressing-modes.pdf
[18] courses/csse2010/avr-instructions.pdf
[19] courses/csse2010/seven-segment-display.pdf
[20] courses/csse2010/2026-09-10-more-c-programming-on-the-avr.pdf