Any Other Bussiness

Luca Tornatore - I.N.A.F.

"Foundation of HPC" course

DATA SCIENCE & SCIENTIFIC COMPUTING

2021-2022 @ Università di Trieste

Outline

Sparse and different topics.
Either concepts and notions that were preparatory for the course or on-the-spot in-depth details that were aftermath of Q&A

THE
C
PROGRAMMING
LANGUAGE

Some sparse topic on C

| Pointers

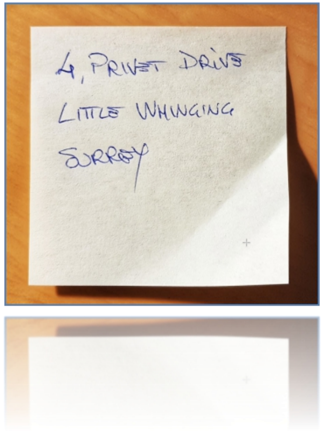
Although it is extraordinarly simple, this topic is usually a terrifying one for most people.
Although it’s plenty of good primaries on the web, and I encourage you to search for them, I think that some simple words are in order.

| Pointers

As first, let’s start with a very simple concept.
I guess you have a special physical place, however you love to imagine it, that you call “home”.

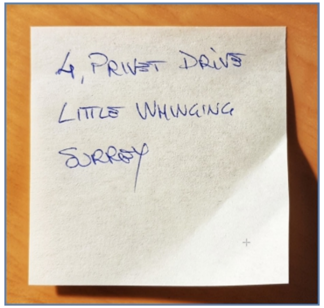
Let’s suppose you are boring normal people, and that your place has an *address:*

4, Privet Drive, Little Whinging, Surrey



You appreciate the fact that this address needs some memory storage to be kept; in my case, a simple sticky note.

Pointers

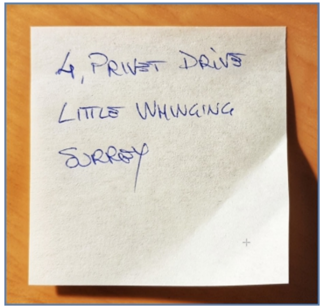


4, Privet Drive, Little Whinging, Surrey

You appreciate the fact that this address needs some memory storage to be kept; in my case, as we said, a simple stick note.

You also appreciate that there is a clear difference between the string written on the note and your actual home
(try to inhabit my note if don't see it).

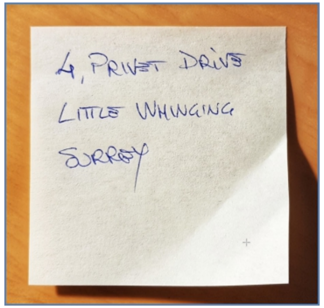
Pointers



So, my note *points* to your home, and occupy some well-defined physical space for the purpose (the sticky note sheet), but it is *not* your home whose physical occupancy does not depend on my note (it's hard to know from the note whether it's a castle or a roulotte).

Conversely, your home is somewhere else, in a well-defined place that is reachable - let us say addressed - by using my note.

Pointers



If you change something in your house, the address on my note is still valid and can be useful to reach you.

Nobody noticed that you renewed your bathroom.

Pointers

12, Grimmaud Place, London

If you move, and your home has now a different address, I need to know it to get there and invite myself for dinner.
To save your new address, I can still use the same sticky note sheet, i.e. the same physical storage of the same size.

The physical location has changed, but I can still use the same sticky note to reach you.

Pointers

All in all, then:

- a ***pointer*** is a variable, i.e. a memory location of fixed size (8B in 64bits systems) which contains an *address*, specifically a memory address and not your place's one.
That address is the *starting point* of a memory area.
So, a pointer can point to an integer (4B), a double (8B) an array of 1G items. Whatever stays in memory has some location where it is stored and that location can be pointed to by a pointer variable.

- ***de-referenceing a pointer*** means to get to the pointer variable, i.e. at the memory location that the variable occupies, to read those bytes acquiring the address and then to get to that memory address

Pointers

- a *pointer* is declared as

```
type *ptr_variable_name;
```

examples:

`char *c;`	points to a char
`double *d;`	points to a double
`struct who_knows *w;`	points to a struct `who_knows`

you assign a value to a pointer variable by assignment:

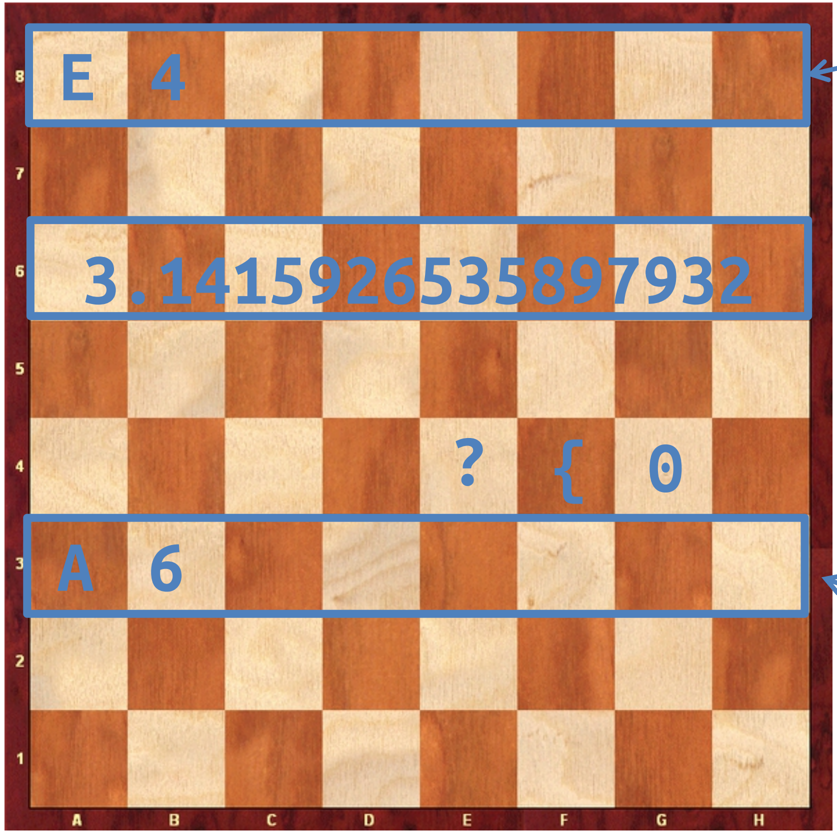
c = 0x123456; c = &my_preferred_letter;

you read the address it points to by de-referencing:

*c is actually the content of the byte pointed by c, not the c's value

note that &c is the c's own address.

Pointers



`char *c;` points to memory location E4. It resides at memory location A8, and lasts 8 bytes until H8, so the c memory address, &c, is A8.

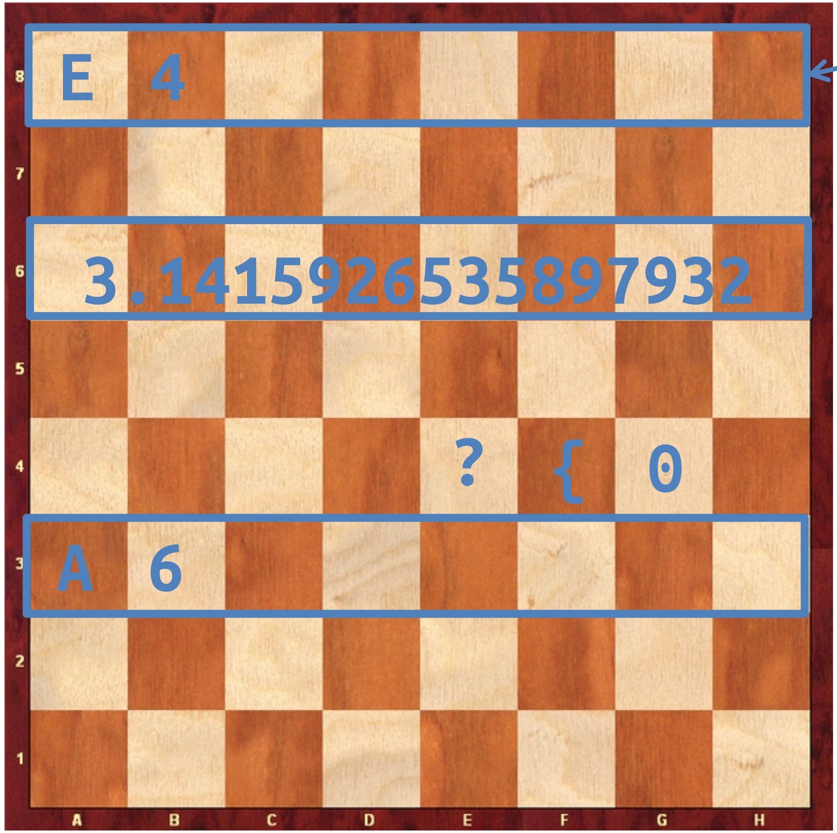
the memory bytes from A6 to H6 are interpreted as a double if you access them starting from A6 by de-referencing the pointer **d** by **`*d`**;

the memory byte at E4 is interpreted as a char if you access it by de-referencing the pointer **c** by **`*c`**;

`double *d;` points to memory location A6. It resides at memory location A3, and lasts 8 bytes until H3, so the d memory address, &d, is A3.
Pointers always occupy 8 bytes.

THE C PROGRAMMING LANGUAGE

| Pointers' arithmetic



as we've seen, **c** points to E4, which contains a byte that interpreted as a character reads as "?".

c is a normal variable (actually, I remind you: it is 8B that we interpret as an address) and as such can be used in arithmetic operations.

For instance, what **c+1** is ?
The operation +1 increases the *value* of **c** by 1; as a result, **c** will point to F4 (*), i.e. to "{".

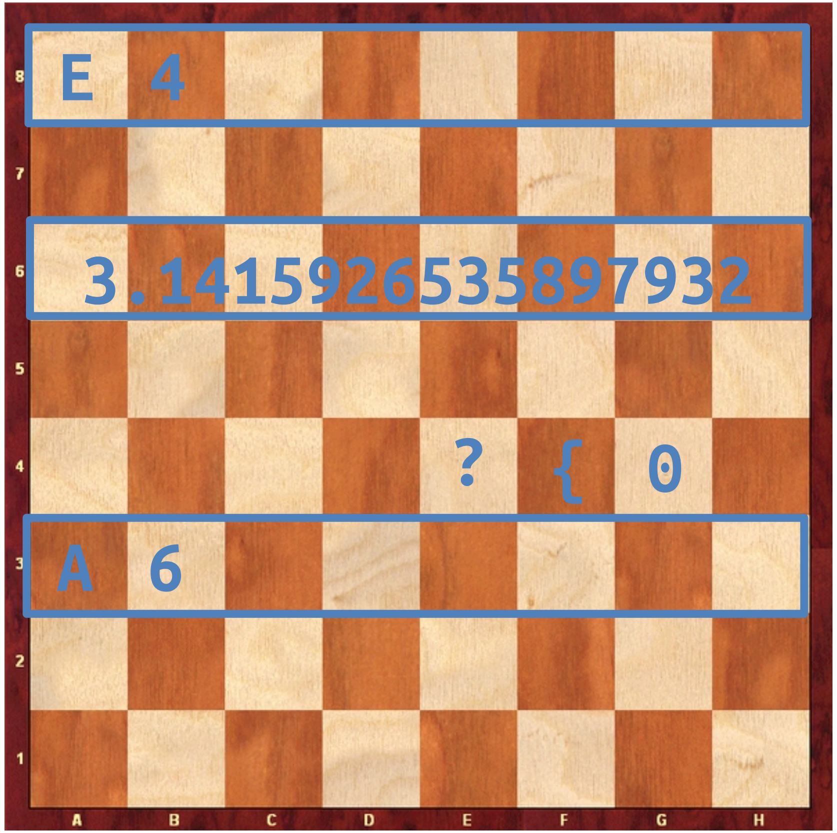
Instead, since **`*c`** means "read the value at the location **c**", **`(*c)++`** increases the value pointed by **c**. In other words, in E4 we'll have ?+1.

What happens if you increase by 1 the value of **d**, the pointer to double ? will it point to A7?

(*) let's imagine che in our chessboard-memory the addresses increases along the rows, so as A1, B1, C1,.. H1, A2, B2,..H2, A3, etc.

THE C PROGRAMMING LANGUAGE

Pointers' arithmetic



What happens if you increase by 1 the value of **d**, the pointer to double ? will it point to A7 ?

The answer is no, because of the pointers arithmetic: when we increase by 1 a pointer, its value - i.e. the address it points to - is increased by the byte-size of the type it points to.

If a pointer points to `char`, since `sizeof(char)` is 1, the increase is 1.
If a pointer points to a `double`, since `sizeof(double)` is 8, the increase is 8bytes, or 1 in `double` units.
If a pointer points to an `int`, the increase is 4 bytes and so on.

```
double *array = malloc(sizeof(double)*N);
double *end = &array[N-1];
double *ptr = array;
```

these 2 loops are equivalent

```
for ( int i = 0; i < N; i++ )
   S += array[i];

for ( ; ptr <= end; ptr++ )
   S += *ptr;
```

Pointers zoo

what this strange creature is ?

```
char **monster;
```

THE C PROGRAMMING LANGUAGE

| Pointers zoo

what this strange creature is ?

```
char **monster;
```

Let’s reason by steps, from right to left (as you should read a C declaration)

1) `monster` → we declare a variable monster
2) `*monster` → it is a pointer
3) `**monster` → it points to a pointer (remind, a pointer is just a variable and it can be pointed)
4) `char **monster` → the pointed pointer points to a `char`

so `*monster` is a pointer which points to a `char`.
Good to know. But what does it mean?

| Pointers zoo

Before fully understand this, we should stress a fact: a pointer points to a precise memory location *but you are not limited to access only that location.*

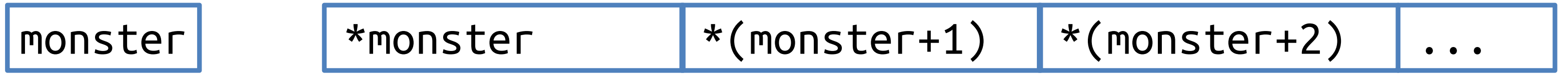
Actually, pointers are the way in C you address dynamically-allocated array:

```
double *array = malloc( sizeof(double) * N );
```

you have allocated room for `N` double entries and the location at the beginning of that memory region is stored in `array`.
Hence, you can access the *ith* element both by `*(array + i)` or by `array[i].`

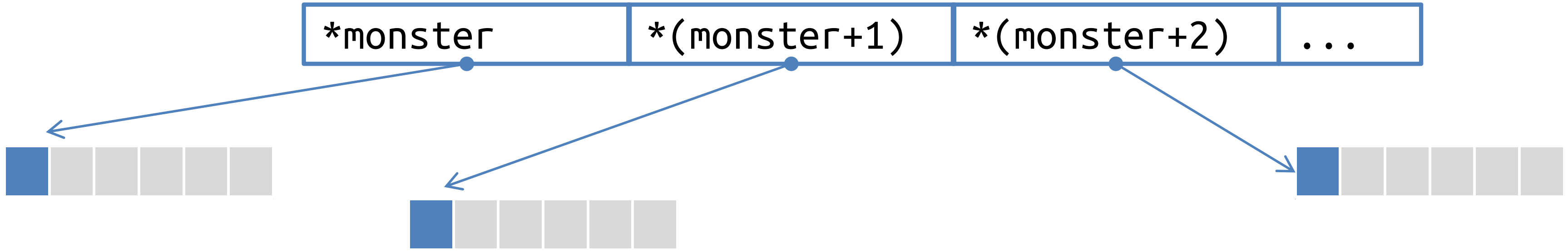
Pointers zoo

Then, it happens that since **`monster`** points to a pointer, also **`monster+1`** points to a pointer (8bytes away because a pointer is 8B long), and so on:



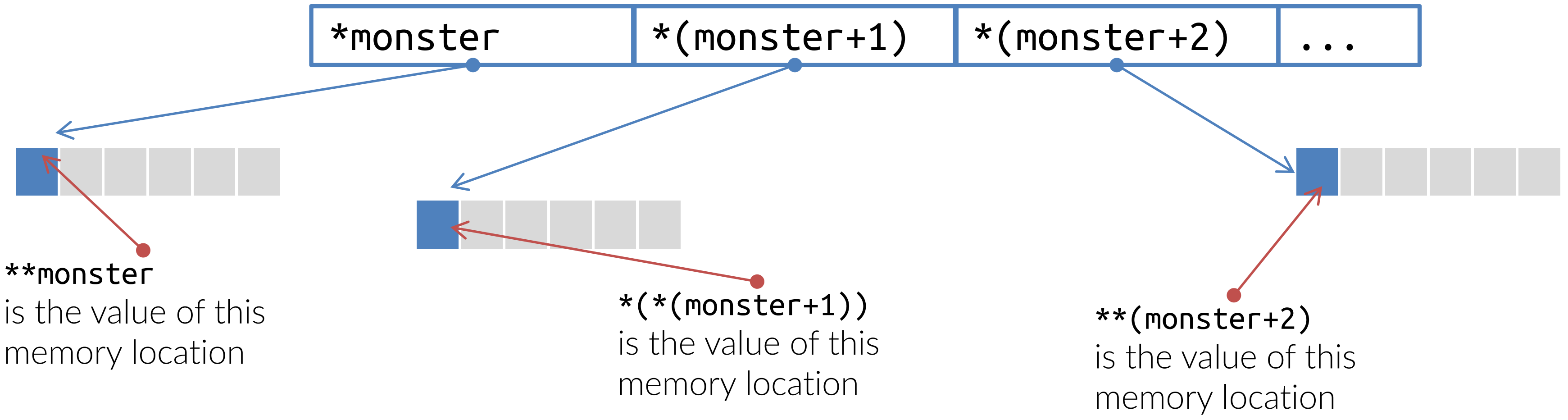
note that *monster+n is *very* different than *(monster+n)

`*(monster+n)` are all interpretable (formally they are since we are referencing them through **`monster`**) as pointers to **`char`**:



Pointers zoo

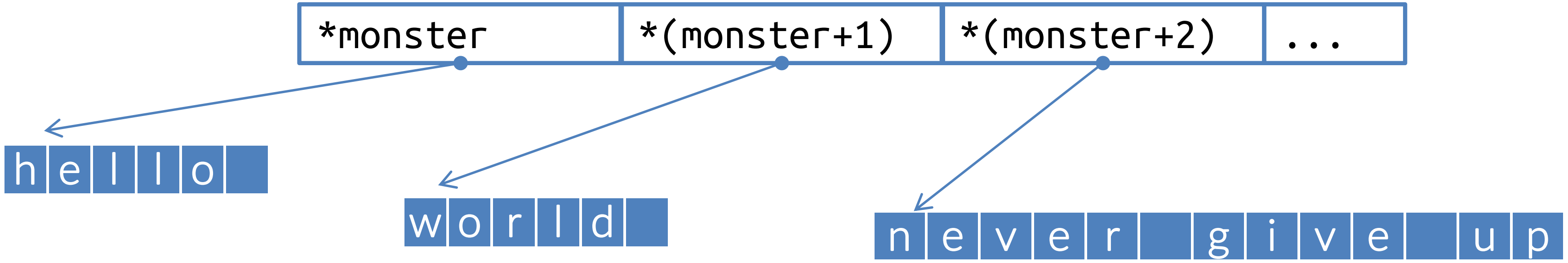
`**(monster+n)` are the value at the byte pointed by `*(monster+n)`


*monster
*(monster+1)
*(monster+2)
...
**monster
is the value of this memory location
((monster+1))
is the value of this memory location
**(monster+2)
is the value of this memory location

`*(*(monster+n)+j)` is the *jth* byte after `*(monster+n)` which actually starts to look like a string..

Pointers zoo

`**(monster+n)` are the value at the byte pointed by `*(monster+n)`



then

`**monster` = ‘h’, `*((*monster)+1)` =’e’, `*((*monster)+2)` =’l’, …

`**(monster+1)` = ‘w’, `*(*(monster+1)+1)` =’o’, `*(*(monster+1)+2)`=’r’, …

or, in other words,

`*monster` = “hello”, `*(monster+1)` = “world”, `*(monster+2)` = “never give up”

Pointers zoo

That is actually how you access the command-line's arguments, for instance:

```
int main (int argc, char **argv)
  {
     ... let's see the worked example, arguments.c
  }
```