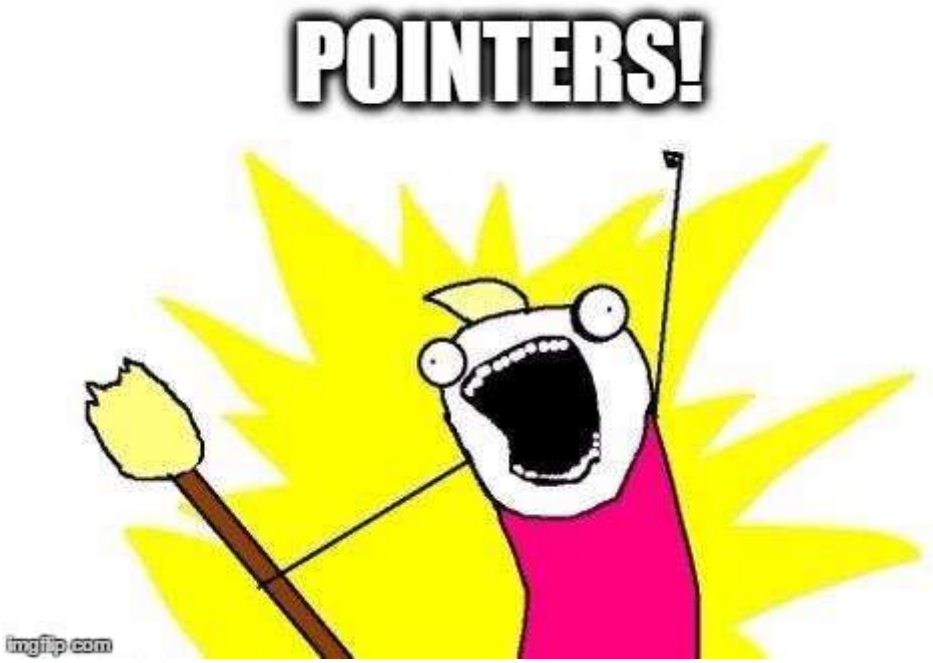
(/)

Pointers and arrays

Read this page, write down and test all the examples (do not copy-paste!).

Pointers



A pointer is a variable which contains a memory address.

Types and memory

Every time you declare a variable, the computer will reserve memory for this variable. The memory reserved will then store the value of the variable.
Depending on the type of the variable, the computer will reserve more or less memory. The size of each type is generally defined in bytes (1 byte = 8 bits, each bit being 0 or 1). The sizes of the types also depend on the computer you are using. Here are the sizes of the most common types on most 64-bit Linux machines:

- char -> 1 byte
- int -> 4 bytes
- float -> 4 bytes

To determine the size of those types on your computer, you can use the `sizeof` operator.

```
ubuntu@ip-172-31-63-244:~/julien$ cat 0-main.c
#include <stdio.h>

/**
 * main - using sizeof to dynamically determine the size of types char, int and float
 *
 * Return: Always 0.
 */
int main(void)
{
   int n;

   printf("Size of type 'char' on my computer: %lu bytes\n", sizeof(char));
   printf("Size of type 'int' on my computer: %lu bytes\n", sizeof(int));
   printf("Size of type 'float' on my computer: %lu bytes\n", sizeof(float));
   printf("Size of type of my variable n on my computer: %lu bytes\n", sizeof(n));
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc -Wall -Wextra -Werror -pedantic -std=gnu89 0-main.c -o
sizeof && ./sizeof
Size of type 'char' on my computer: 1 bytes
Size of type 'int' on my computer: 4 bytes
Size of type 'float' on my computer: 4 bytes
Size of type of my variable n on my computer: 4 bytes
```

The size of a type will determine how many different possible values a variable of this type can hold. For instance, a `char` variable could only hold 256 (2^8, 8 being the number of bits) different values: from -128 to 127. And because the size of `int` is 4 bytes, so 32 bits, an `int` variable can hold 2^32 different possible values.

When we declare a variable, the computer will reserve the right amount of space for the variable in the memory (depending on its type). The space reserved for the variable is its address. And when we assign a value to this variable, the computer will store this value at its address.

```
char c;
```

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable									c											
Value	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

In this example, we simply declare a variable `c` of type `char`. The address of `c` is `8`. At this stage, you have not assigned a value to your variable `c`. So you do not know its value. It depends on what this memory address was used for before. You should never assumed that it is 0.

```
char c;

c = 'H';
```

When we assign the value `'H'` to `c`, then `'H'` is stored at `c`'s address.

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable									c											
Value	?	?	?	?	?	?	?	?	'H'	?	?	?	?	?	?	?	?	?	?	?

But you know that a byte can only store numbers. So actually, the byte will not hold exactly the letter `'H'`, but its ascii code, which is `72` (man ascii). So it really looks like this in memory:

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable									c											
Value	?	?	?	?	?	?	?	?	72	?	?	?	?	?	?	?	?	?	?	?

Since integers are stored within four bytes of memory, the same example with an `int` variable would look like this:

```
int n;

n = 98;
```

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable							n													
Value	?	?	?	?	?	?	98				?	?	?	?	?	?	?	?	?	?

In this example, the address of the variable `n` is the smallest address of its bytes, so in this example, `26`.
Note that you do not control the address where the variable is stored. In order to know what is the address of a variable, you can use the "address-of unary operator" `&`.

```
ubuntu@ip-172-31-63-244:~/julien$ cat 1-main.c
#include <stdio.h>

/**
 * main - addresses of variables
 *
 * Return: Always 0.
 */
int main(void)
{
   char c;
   int n;

   printf("Address of variable 'c': %p\n", &c);
   printf("Address of variable 'n': %p\n", &n);
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc 1-main.c -o address && ./address
Address of variable 'c': 0x7ffc370ef13b
Address of variable 'n': 0x7ffc370ef13c
```

Note that:

- You can use `%p` to print addresses (the values of pointers) with `printf`
- This example will not compile with our regular flags `-Wall -Wextra -Werror -pedantic -std=gnu89`. You'll learn why later

Storing memory addresses

Now that we know how to get an address, we can store it in a... pointer! :)
A pointer is simply the address of a piece of data in memory. A pointer variable is a variable that stores the address of that piece of data. Like any other variable it needs to be declared. General form is:

```
var_type *var;
```

(/)

- The `*` tells that the variable `var` is a pointer…
- … that points to a var_type.
- The value of `var` will be a memory address holding a value of type `var_type`

```
int *ptr;
```

In this example, `ptr` is the name of the variable, of type "pointer to an integer". Anything that is on the left of the last `*` before the name of the variable will give you the type that the pointer points to.

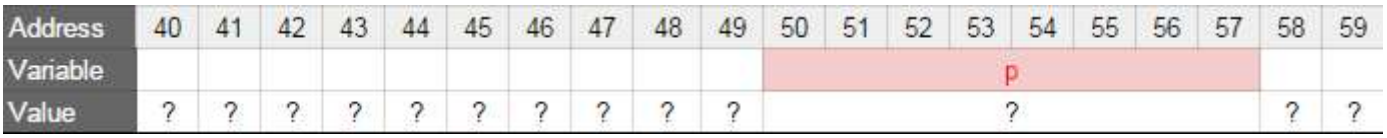
```
/* ptr2 is a pointer to a char */
char *ptr2
```

Because a pointer is like any other variable, the computer will also reserve the right amount of memory for it to store its value. On most 64 bits machines, the size of pointers is 8 bytes.

```
ubuntu@ip-172-31-63-244:~/julien$ cat 2-main.c
#include <stdio.h>

/**
 * main - printing the size, in bytes, of a pointer
 *
 * Return: Always 0.
 */
int main(void)
{
   int *p;

   printf("Size of pointer: %lu\n", sizeof(p));
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc -Wall -Werror -pedantic -Wextra -std=gnu89 2-main.c -o
psize && ./psize
Size of pointer: 8
```



To get the address where a pointer is stored, you can use the same technique as for any other variable: use the `&` operator.

```
ubuntu@ip-172-31-63-244:~/julien$ cat 3-main.c
#include <stdio.h>

/**
 * main - printing the address of a pointer
 *
 * Return: Always 0.
 */
int main(void)
{
  int *p;

  printf("Address of variable 'p': %p\n", &p);
  return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc 3-main.c -o paddress && ./paddress
Address of variable 'p': 0x7ffc9efc0de8
```

Ok, now let's store the address of a variable into a pointer.

```
int n;
int *p;

n = 98;
p = &n;
```

Because `&n` gives us the address of the variable `n`, the variable `p` now holds the address of the variable `n`: `p` points to `n`. If the variable `n`'s address were 26, then the value of our pointer `p` would be 26.

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable							n													
Value	?	?	?	?	?	?	98				?	?	?	?	?	?	?	?	?	?

Address	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Variable											p									
Value	?	?	?	?	?	?	?	?	?	?	26								?	?

```
ubuntu@ip-172-31-63-244:~/julien$ cat 4-main.c
#include <stdio.h>

/**
 * main - storing the address of variable into a pointer
 *
 * Return: Always 0.
 */
int main(void)
{
  int n;
  int *p;

  n = 98;
  p = &n;
  printf("Address of 'n': %p\n", &n);
  printf("Value of 'p': %p\n", p);
  return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc 4-main.c -o pn && ./pn
Address of 'n': 0x7ffc6f64b6d4
Value of 'p': 0x7ffc6f64b6d4
```

Remember that a pointer can only point to a variable of the type it is supposed to point to. The following example is incorrect:

```
char c;
int *p;

p = &c;
```

Dereferencing

You could argue that so far, we could have used an integer (or a `unsigned long int` that is also 8 bytes long on most 64 bits computers) to store an address, since it is a number. The real power of pointers is that they can manipulate values stored at the memory address they point to. This is called dereferencing. To do this, you can use the dereference operator `*`.

```
ubuntu@ip-172-31-63-244:~/julien$ cat 5-main.c
#include <stdio.h>

/**
 * main - derefencing pointers
 *
 * Return: Always 0.
 */
int main(void)
{
   int n;
   int *p;

   n = 98;
   p = &n;
   printf("Value of 'n': %d\n", n);
   printf("Address of 'n': %p\n", &n);
   printf("Value of 'p': %p\n", p);
   *p = 402;
   printf("Value of 'n': %d\n", n);
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc 5-main.c -o dereference && ./dereference
Value of 'n': 98
Address of 'n': 0x7ffd9c1969a4
Value of 'p': 0x7ffd9c1969a4
Value of 'n': 402
```

Let's walk through this example:

- `int *p;` : `*` is used in the declaration: `p` is a pointer to an integer, and so, after dereferencing, `*p` is an integer.
- `p = &n;` : `&` takes the address of `n`. So now `p == &n`, so `*p == n`

At this point, the memory looks like this:

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable							n													
Value	?	?	?	?	?	?	98				?	?	?	?	?	?	?	?	?	?

Address	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Variable											p									
Value	?	?	?	?	?	?	?	?	?	?	26								?	?

- `*p = 402;` : equivalent to `n = 402`, since `p == &n`. Now `*p == 402` so `n == 402`.

This is what it looks like after this statement in memory:

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable							n													
Value	?	?	?	?	?	?	402				?	?	?	?	?	?	?	?	?	?

Address	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Variable											p									
Value	?	?	?	?	?	?	?	?	?	?	26								?	?

This works exactly the same for other types:

(/)

```
ubuntu@ip-172-31-63-244:~/julien$ cat 6-main.c
#include <stdio.h>

/**
 * main - derefencing pointers, example with int and char types
 *
 * Return: Always 0.
 */
int main(void)
{
   int n;
   int *p;
   char c;
   char *pp;

   c = 'H';
   pp = &c;
   n = 98;
   p = &n;
   printf("Value of 'c': %d ('%c')\n", c, c);
   printf("Address of 'c': %p\n", &c);
   printf("Value of 'pp': %p\n", pp);
   printf("Value of 'n': %d\n", n);
   printf("Address of 'n': %p\n", &n);
   printf("Value of 'p': %p\n", p);
   *p = 402;
   *pp = 'o';
   printf("Value of 'n': %d\n", n);
   printf("Value of '*pp': %d\n", *pp);
   printf("Value of 'c': %d ('%c')\n", c, c);
   printf("Value of '*pp': %d ('%c')\n", *pp, *pp);
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc 6-main.c -o dereference2 && ./dereference2
Value of 'c': 72 ('H')
Address of 'c': 0x7ffe57e9cc7b
Value of 'pp': 0x7ffe57e9cc7b
Value of 'n': 98
Address of 'n': 0x7ffe57e9cc7c
Value of 'p': 0x7ffe57e9cc7c
Value of 'n': 402
Value of '*pp': 111
Value of 'c': 111 ('o')
Value of '*pp': 111 ('o')
```

Note that `*` has a different meaning depending on the context (declaring vs dereferencing pointers).

- at declaration, it is used to declare a variable of type pointer to something. Example: `int *n;`
- when used inside the code it is used to dereference pointers. Example `*n = 98;`

Functions parameters are passed by value

When we call a function in C, parameters are copied.

```
/**(/)
 * modif_my_param - this function does not modify n
 * @m: a useless integer
 *
 * Return: nothing.
 */
void modif_my_param(int m)
{
   m = 402;
}

/**
 * main - parameters are passed by value
 *
 * Return: Always 0.
 */
int main(void)
{
   int n;

   n = 98;
   modif_my_param(n);
   return (0);
}
```

In this example, when we call the `modif_my_param` function, the value of n (98) is copied inside a new variable m, only available in the `modif_my_param` function. Here is what it looks like in memory before the line `m = 402;` is executed:

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable																	m			
Value	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	98			

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable							n													
Value	?	?	?	?	?	?	98				?	?	?	?	?	?	?	?	?	?

After `m = 402;` the memory looks like this:

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable																	m			
Value	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	402			

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable							n													
Value	?	?	?	?	?	?	98				?	?	?	?	?	?	?	?	?	?

When we leave the function `modif_my_param` the variable `m` is destroyed and does not exist anymore. Its value though, stays in memory until this space is used by the program for another variable or something else.

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable																				
Value	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?		402		

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable							n													
Value	?	?	?	?	?	?	98				?	?	?	?	?	?	?	?	?	?

This rule applies to any type of variable. But since the values of pointers are addresses, it is possible to modify a variable from outside the function it is declared, using a pointer.

```
ubuntu@ip-172-31-63-244:~/julien$ cat 7-main.c
#include <stdio.h>

/**
 * modif_my_param - set the integer to 402
 * @m: a pointer the integer we want to set to 402
 *
 * Return: nothing
 */
void modif_my_param(int *m)
{
   printf("Value of 'm': %p\n", m);
   printf("Address of 'm': %p\n", &m);
   *m = 402;
}

/**
 * main - how to change the value of a variable from outside the function
 * it is declared in, using a pointer.
 *
 * Return: Always 0.
 */
int main(void)
{
   int n;
   int *p;

   p = &n;
   n = 98;
   printf("Value of 'n' before the call: %d\n", n);
   printf("Address of 'n': %p\n", &n);
   printf("Value of 'p': %p\n", p);
   printf("Address of 'p': %p\n", &p);
   modif_my_param(p);
   printf("Value of 'n' after the call: %d\n", n);
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc 7-main.c -o modify_param && ./modify_param
Value of 'n' before the call: 98
Address of 'n': 0x7ffd70432494
Value of 'p': 0x7ffd70432494
Address of 'p': 0x7ffd70432498
Value of 'm': 0x7ffd70432494
Address of 'm': 0x7ffd70432478
Value of 'n' after the call: 402
```

In this example, here is what happens: Before the call to the function `modif_my_param` , the memory looks like this:

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable									p											
Value	?	?	?	?	?	?	?	?	26								?	?	?	?

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable							n													
Value	?	?	?	?	?	?	98				?	?	?	?	?	?	?	?	?	?

When we call the function `modif_my_param`, the value of `p` is stored in a new variable called `m`:

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable									p											
Value	?	?	?	?	?	?	?	?	26								?	?	?	?

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable							n				m									
Value	?	?	?	?	?	?	98				26									?

Since `m` stores the same memory address, it points to the same address, and so both `p` and `m` now point to `n`. Therefore, when we execute the line `*m = 402;` we modify the value of `n` and `n` now holds `402`.

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable									p											
Value	?	?	?	?	?	?	?	?	26								?	?	?	?

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable							n				m									
Value	?	?	?	?	?	?	402				26									?

When we leave the function `modif_my_param`, the variable `m` is destroyed, but `n`'s value is still `402`:

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable									p											
Value	?	?	?	?	?	?	?	?	26								?	?	?	?

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable							n													
Value	?	?	?	?	?	?	402				26									?

Using the same format, try to represent in memory what is happening at every step of the following program. When done, add some `printf`s and compile to verify your hypothesis.

```
/**(/)
 * modif_my_char_var - Solve me
 *
 * Return: nothing.
 */
void modif_my_char_var(char *cc, char ccc)
{
   *cc = 'o';
   ccc = 'l';
}

/**
 * main - Solve me
 *
 * Return: Always 0.
 */
int main(void)
{
   char c;
   char *p;

   p = &c;
   c = 'H';
   modif_my_char_var(p, c);
   return (0);
}
```

Arrays

Arrays in C are contiguous memory areas that hold a number of values of the same type. Unlike some other languages, in C, all elements of an array have the same type. To declare an array we use this syntax: `type var_name[number_of_elements];` , where `number_of_elements` is the number of elements of type `type` that we need.

```
int t[5];
```

In this example we declare an array of 5 integers. The computer will reserve a continuous space for 5 integers in memory. In memory, it would look like something like this.

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable	a																			
Value	?				?				?				?				?			

We access the different elements of an array this way: `t[0]` will access the first element, `t[1]` the second element, and so on.

```
ubuntu@ip-172-31-63-244:~/julien$ cat 9-main.c
#include <stdio.h>

/**
 * main - Accessing the different elements of an array
 *
 * Return: Always 0.
 */
int main(void)
{
   int a[5];

   a[0] = 98;
   a[1] = 198;
   a[2] = 298;
   a[3] = 398;
   a[4] = 498;
   printf("Value of a[0]: %d\n", a[0]);
   printf("Value of a[1]: %d\n", a[1]);
   printf("Value of a[2]: %d\n", a[2]);
   printf("Value of a[3]: %d\n", a[3]);
   printf("Value of a[4]: %d\n", a[4]);
   printf("Address of 'a[0]': %p\n", &(a[0]));
   printf("Address of 'a[1]': %p\n", &(a[1]));
   printf("Address of 'a[2]': %p\n", &(a[2]));
   printf("Address of 'a[3]': %p\n", &(a[3]));
   printf("Address of 'a[4]': %p\n", &(a[4]));
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc 9-main.c -o array && ./array
Value of a[0]: 98
Value of a[1]: 198
Value of a[2]: 298
Value of a[3]: 398
Value of a[4]: 498
Address of 'a[0]': 0x7ffcbca77dd0
Address of 'a[1]': 0x7ffcbca77dd4
Address of 'a[2]': 0x7ffcbca77dd8
Address of 'a[3]': 0x7ffcbca77ddc
Address of 'a[4]': 0x7ffcbca77de0
```

Memory would look like this before exiting the `main` function:

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable	a[0]				a[1]				a[2]				a[3]				a[4]			
Value	98				198				298				398				498			

Pointers vs Arrays

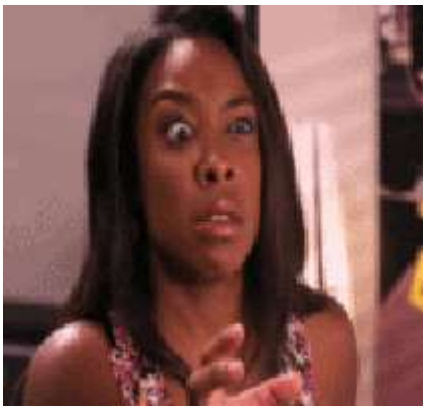
In C, **an array is NOT a pointer**, the variables we declare as arrays do not hold a memory address.

When we declare an array, we use a name to refer to it, but it is only a name. Array names are identifiers that identify the entire array object. They are not pointers to anything. That is why we can not change the value of a 'variable' which is an array.

```
julien@ubuntu:~/c/$ cat 10-main.c
/**
 * main - You can not modify a
 *
 * Return: Always 0.
 */
int main(void)
{
    int a[5];
    int b;
    int c[5];

    a = 0; /* nop */
    a = &b; /* nop */
    a = c; /* nop */
    return (0);
}
julien@ubuntu:~/c/ideas_tests$ gcc 10-main.c
10-main.c: In function ‘main’:
10-main.c:12:4: error: assignment to expression with array type
  a = 0; /* nop */
    ^
10-main.c:13:4: error: assignment to expression with array type
  a = &b; /* nop */
    ^
10-main.c:14:4: error: assignment to expression with array type
  a = c; /* nop */
    ^
julien@ubuntu:~/c$
```

But, you can still use the name of the array in your code, and its value will be… the address of the first element of the array. Wait… WAT?



```
ubuntu@ip-172-31-63-244:~/julien$ cat 16-main.c
#include <stdio.h>

/**
 * main - An array is not a pointer, but...
 *
 * Return: Always 0.
 */
int main(void)
{
   int a[98]; /* array */

   printf("a: %p\n", a);
   printf("&a[0]: %p\n", &a[0]);
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc 16-main.c -o avsp && ./avsp
a: 0x7fffc211caa0
&a[0]: 0x7fffc211caa0
```

So what is happening here? If arrays are not pointers, why is it that the value of an array is the address of the first element of the same array. Isn't this the definition of a pointer: "a variable which contains the address"? Well... YES, this is a pointer, but the variable `a` (the array) is NOT. This is why:

When the name of an array is used in an expression, the array type gets automatically implicitly converted to pointer-to-element type in almost all contexts (this is often referred to as "array type decay"). The resultant pointer is a completely independent temporary value, that is the address of the first element of the array.

```
ubuntu@ip-172-31-63-244:~/julien$ cat 17-main.c
#include <stdio.h>
void f(int *a);

/**
 * main - Illustrating the array type decay
 *
 * Return: Always 0.
 */
int main(void)
{
   int *p;
   int t[10];

   p = t; /* This works because of the auto implicit conversion to (int *) */
   printf("t: %p\n", t);
   printf("&t[0]: %p\n", &t[0]);
   printf("p: %p\n", p);
   f(t);
   return (0);
}

/**
 * f - prints the value of a pointer of type int
 * @a: address of an integer we need to print
 *
 * Return: Nothing.
 */
void f(int *a)
{
   printf("a: %p\n", a);
   return;
}
ubuntu@ip-172-31-63-244:~/julien$ gcc 17-main.c && ./a.out
t: 0x7ffd553e7380
&t[0]: 0x7ffd553e7380
p: 0x7ffd553e7380
a: 0x7ffd553e7380
```

There are two exceptions to this: when the array name is an operand of `sizeof` or unary `&` (address-of), the name of the array then refers to the array object itself.

sizeof

In the context of `sizeof`, the name of the array refers to the array object itself (composed of all its elements). Thus, `sizeof` an array will give you the amount of memory space used by all its elements.

```
ubuntu@ip-172-31-63-244:~/julien$ cat 14-main.c
#include <stdio.h>

/**
 * main - printing the sizeof an array
 *
 * Return: Always 0.
 */
int main(void)
{
   int array[10];

   printf("sizeof array: %lu\n", sizeof(array));
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc -Wall -Werror -Wextra -pedantic -o sizeofarray 14-main.
c && ./sizeofarray
sizeof array: 40
```

&

In the context of `&`, the name of the array refers to the array object itself (composed of all its elements). So the `address-of` the array name will give you the address of the whole array, which is the same as the address of its first element.

```
ubuntu@ip-172-31-63-244:~/julien/curriculum_by_julien/low_level_programming/pointers$ cat 18-
main.c
#include <stdio.h>

/**
 * main - prints the address of an array
 *
 * Return: Always 0.
 */
int main(void)
{
   char b[98];

   printf("b: %p\n", b);
   printf("&b: %p\n", &b);
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien/curriculum_by_julien/low_level_programming/pointers$ gcc 18-
main.c && ./a.out
b: 0x7fff6130ef60
&b: 0x7fff6130ef60
```

As a consequence, the value of `b` is the same as the value of `&b`. But they do not have the same type:

- `b`: in this context, general rule applies, so using `b` will be evaluated as the address of the first element of the array. Thus, `b` is of type `char *`
- `&b`: in this context (one of the two exceptions to the general rule), `b` will be evaluated as the array object itself. so `b` is of type `char[98]` - an array of 98 characters - and then `&b` is of type `char (*)[98]`, a pointer to an array of 98 characters.

This is important because when we will manipulate those two pointers, they will behave differently, since the size of their type is different:

- `sizeof b` is `98` (see above)
- `sizeof &b` is 8, (if you are using a regular 64 bits machine) (see below)

```
ubuntu@ip-172-31-63-244:~/julien/curriculum_by_julien/low_level_programming/pointers$ cat 19-main.c
#include <stdio.h>

/**
 * main - prints the size of an array and the address of an array
 *
 * Return: Always 0.
 */
int main(void)
{
   char b[98];

   printf("b: %p\n", b);
   printf("&b: %p\n", &b);
   printf("sizeof(b): %lu\n", sizeof(b));
   printf("sizeof(&b): %lu\n", sizeof(&b));
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien/curriculum_by_julien/low_level_programming/pointers$ gcc 19-main.c && ./a.out
b: 0x7ffe0b9c62f0
&b: 0x7ffe0b9c62f0
sizeof(b): 98
sizeof(&b): 8
```

Pointers Arithmetic

Another way to access different elements of an array, is to use this other notation: `*(var + x)`, where `var` is the name of an array, and `x` is the (x+1)th element (starting counting elements at 0 of course) of this array. For instance if we declare:

```
int i[10];
```

then, when we will use `i` in the code, `i[5]` will be the same as `*(i + 5)`. We already knew that `i` is evaluated as a pointer to the first element of the array, since `i` is an array. But now, we also know that `i + 5` will be evaluated as a pointer to the 6th element of the array `i`.

```
ubuntu@ip-172-31-63-244:~/julien/curriculum_by_julien/low_level_programming/pointers$ cat 10-main.c
#include <stdio.h>

/**
 * main - illustrates pointers arithmetic
 *
 * Return: Always 0.
 */
int main(void)
{
   int a[5];

   *a = 98; /* same as *(a + 0) */
   *(a + 1) = 198;
   *(a + 2) = 298;
   *(a + 3) = 398;
   *(a + 4) = 498;
   printf("Value of a[0]: %d\n", *a);
   printf("Value of a[1]: %d\n", *(a + 1));
   printf("Value of a[2]: %d\n", *(a + 2));
   printf("Value of a[3]: %d\n", *(a + 3));
   printf("Value of a[4]: %d\n", *(a + 4));
   printf("----------------\n");
   printf("Value of 'a' (also address of a[0]): %p\n", a);
   printf("Address of 'a[1]': %p\n", (a + 1));
   printf("Address of 'a[1]': %p\n", &(*(a + 1)));
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien/curriculum_by_julien/low_level_programming/pointers$ gcc 10-main.c -o array && ./array
Value of a[0]: 98
Value of a[1]: 198
Value of a[2]: 298
Value of a[3]: 398
Value of a[4]: 498
----------------
Value of 'a' (also address of a[0]): 0x7ffff8f19240
Address of 'a[1]': 0x7ffff8f19244
Address of 'a[1]': 0x7ffff8f19244
```

But wait a second, if the value of `a` is `0x7ffff8f19240`, how come `a + 1 == 0x7ffff8f19244` and not `0x7ffff8f19241`?

This is the pointers arithmetic. The computer knows that `a` points to an integer. The computer also knows that the size of an integer in memory is `sizeof(int)` bytes - in this case 4 bytes - and concludes that the next element of this type will be stored 4 bytes later in memory.

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable	*a				*(a + 1)				*(a + 2)				*(a + 3)				*(a + 4)			
Value	98				198				298				398				498			

If this works for arrays, which are evaluated as pointers in this context, this means that this arithmetic also works for “regular” pointers.

```
ubuntu@ip-172-31-63-244:~/julien$ cat 20-main.c
#include <stdio.h>

/**
 * main - pointers arithmetic
 *
 * Return: Always 0.
 */
int main(void)
{
   int *p;
   int a[2];
   int n;

   p = &n;
   printf("p = &n;\np: %p\n", p);
   printf("p + 1: %p\n", p + 1);
   printf("p + 2: %p\n", p + 2);
   printf("p + 10: %p\n", p + 10);
   /* possible since a is evaluated */
   /* as an int * in this context */
   p = a;
   printf("p = a;\np: %p\np + 1: %p\n", p, p + 1);
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc 20-main.c && ./a.out
p = &n;
p: 0x7ffd90a8cd2c
p + 1: 0x7ffd90a8cd30
p + 2: 0x7ffd90a8cd34
p + 10: 0x7ffd90a8cd54
p = a;
p: 0x7ffd90a8cd30
p + 1: 0x7ffd90a8cd34
```

Try to understand and draw the memory state at each step of the following program. Once you feel you understand everything, add some `printf` s, compile and run it to verify your hypothesis.

```
/**
 * main - Solve me
 *
 * Return: Always 0.
 */
int main(void)
{
   int a[5];
   int *p;
   int *p2;

   *a = 98;
   *(a + 1) = 198;
   *(a + 2) = 298;
   a[3] = 398;
   *(a + 4) = 498;
   p = a + 1;
   *p = 98;
   p2 = a + 3;
   *p2 = *p + 1337;
   return (0);
}
```

Strings

Of course, we can also create an array of chars. It would work exactly the same way.

```
ubuntu@ip-172-31-63-244:~/julien$ cat 12-main.c
#include <stdio.h>

/**
 * main - Creates an array of chars and prints it
 *
 * Return: Always 0.
 */
int main(void)
{
   char a[6];

   *a = 'S';
   *(a + 1) = 'c';
   *(a + 2) = 'h';
   a[3] = 'o';
   *(a + 4) = 'o';
   a[5] = 'l';
   printf("%c%c%c%c%c%c\n", a[0], a[1], a[2], a[3],
               a[4], a[5]);
   return (0);
}
ubuntu@ip-172-31-63-244:~/julien$ gcc -Wall -Werror -Wextra -pedantic -std=gnu89 -o achar 12-
main.c && ./achar
School
ubuntu@ip-172-31-63-244:~/julien$
```

In this example, memory looks like this before exiting the program:

Address	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Variable	a																			
Value	72	111	108	98	101	114	116	111	110	?	?	?	?	?	?	?	?	?	?	?

That would be one way to store strings, but this is not very efficient. If we wanted to store a very long text, imagine how much time would we spend storing each character of the string in our array!
Fortunately, we can also use text in double quotes `"`.

```
char a[7] = "School";
```

You probably noticed that in this example, we used an array of 7 characters to store a string of 6 characters. This is because in C, strings end with the char `'\0'` (ascii value = 0).

In the above example, note that this statement will actually copy the string “School” into the variable `a`.
Strings also have values in C: the address of their first character in memory:

```
ubuntu@ip-172-31-63-244:~/julien/curriculum_by_julien/low_level_programming/pointers$ cat 13-
main.c && gcc 13-main.c && ./a.out
#include <stdio.h>

/**
 * main - Prints the value of a string
 *
 * Return: Always 0.
 */
int main(void)
{
   char a[7] = "School";

   printf("%s\n", a);
   printf("Value of the last char of a: %d\n", a[9]);
   printf("Value of a: %p\n", a);
   printf("Value of \"School\": %p\n", "School");
   return (0);
}
School
Value of the last char of a: 0
Value of a: 0x7ffe1a23cc50
Value of "School": 0x400749
```

Memory would look like:

Adresse	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Variable	a																			
Value	83	99	104	111	111	108	0	?	?	?	?	?	?	?	?	?	?	?	?	?

Adresse	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
Variable										"School"										
Value	?	?	?	?	?	?	?	?	?	83	99	104	111	111	108	0	?	?	?	?

Since every string in C ends with a `'\0'` we do not need to know their size to use them. By knowing the address of the first character of strings (with a pointer to a char), C functions can easily print them using a loop, one character at a time, until they hit the character `'\0'`.

#cisfun #pointersarefun #everythingisawesome

BONUS

Our memory templates:

- Memory template.xlsx (/rltoken/0A41AUjhFj2TMIT5S_Q74Q)
- Memory template.ods (/rltoken/p1XS_Ei3qH-z3wbn_RUkHA)