Piscine — Tutorial D2 AM

version #

Assistants C/Unix 2022 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.assistants.epita.fr

1 Strings

1.1 Definition

A string is an ordered sequence of characters.

In *C*, there is no built-in type to represent strings. So, in order to represent this kind of data, we use a **character array** (`char[]`).

Furthermore, one of the most important characteristic of a string is that it ends with `'\0'`, a special character symbolizing the end of the string.

There are many special characters including:

`\n`	line break
`\t`	tabulation
`\0`	end of string character, equals 0
`\\`	to write \
`\"`	double quote
`'`	single quote '

Let us see with a basic example:

```c
#include <stdio.h>

int main(void)
{
    char s[] =
    {
        'T', 'e', 's', 't', '.', '\0'
    };

    puts(s);
    return 0;
}
```

We can easily see that writing strings in this form is not practical at all. Fortunately for us, the *C* language provides a simple way to write strings: **string literals** (or *constant strings*).

The following example is semantically identical to the one above, and the terminating character (`'\0'`) is automatically added at the end of the string:

```c
#include <stdio.h>

int main(void)
{
    char s[] = "Test.";

    puts(s);
    return 0;
}
```

Another advantage of string literals: they can be passed directly to functions taking `char[]` as arguments!

```c
#include <stdio.h>

int main(void)
{
    puts("Test.");
    return 0;
}
```

Going further...

You may have noticed by reading the man of `puts(3)` that, in the prototype of this function, the argument type is `const char *s`. Do not worry about this at the moment: it is exactly the same as `const char s[]` when passing arguments.

Another example with `const`:

```c
char str1[19] = "The cake is a lie.";
char str2[]   = "The cake is a lie.";
const char str3[] = "The cake is a lie.";

str1[14] = 'p'; // OK
str2[14] = 'p'; // OK
str3[14] = 'p'; // Compilation error
```

Finally, the length of a string is the number of characters before `\0`. Please notice that it is different from `sizeof(string)`, which returns the number of bytes used by the array, including the `\0`.

```c
char str1[] = "Portable";
char str2[] = "Por\0table";
```

Try to print these two strings with `puts`. You will see that the first string has 8 characters while the second one has only 3 characters.

Be careful, string literals are null-terminated (`\0` added automatically at the end), but arrays declared with braces are not.

Also, you cannot always provide a string literal to a function like:

```c
foo("bar");
```

The string `bar` cannot be modified. If the `foo` function tries to change it, the program will crash. If you need to modify the string, declare it:

```c
char str[] = "bar";
foo(str);
```

Tips

Again, this part refers to the notion of `pointers` that will be introduced in the next section. Do not hesitate to read this part again *after* reading the section on pointers. For now, just keep in mind that the following function declarations are equivalent:

```c
void foo(char arr[]);
void foo(char *arr);
```

Thus:

```c
#include <stdio.h>

void foo(char *str)
{
    str[0] = 'p';
}

int main(void)
{
    char str[] = "toto";
    printf("%s\n", str);    // "toto"
    foo(str);
    printf("%s\n", str);    // "poto"
}
```

However note that when declaring the string literal like this:

```c
char *str = "toto";
```

The string is **read-only**, meaning you cannot modify its content.

1.2 Exercises

1.2.1 My strlen

Goal

You must implement the following `strlen(3)` function :

```c
size_t my_strlen(const char *s);
```

This function must have the same behavior as the `strlen(3)` function described in the third section of the man. Your code will not be tested with `NULL` pointers.

1.2.2 My strlowcase

Goal

You must implement a function to changes all upper case ASCII letters into lower case. `NULL` pointer will not be tested.

```c
void my_strlowcase(char *str);
```

```c
#include <stdio.h>
#include "my_strlowcase.h"

int main(void)
{
    char str[] = "azerty1234XYZ &(";
    my_strlowcase(str);
    puts(str);
    return 0;
}
```

```
42sh$ ./my_strlowcase | cat -e
azerty1234xyz &($
```

2 Pointers

2.1 Introduction

Be careful!

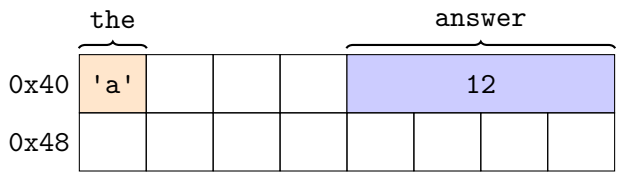
Pointers are a fundamental concept, pay extra attention!

Every variable used in your program needs to be present in your computer's memory somewhere in order to be accessed. By knowing a variable's type and its address you can access your memory to look-up the current value of your variable.

Tips

All the addresses written in the examples are arbitrarily chosen.

```c
char the = 'a'; /* address: 0x40 */
int answer = 12; /* address: 0x44 */
```



Tips

`the` and `answer` do not take the same amount of space in memory, because they are different types of different size. On the PIE an `int` takes four bytes, and `char` takes one byte of memory to be stored.

This memory-address and type combinaison is called a pointer. The type associated to an address is needed to make sense of the value located at that address. A pointer is an address associated with a type. Here, `0x40` and `char` allows us to create the pointer to `the`.

You can write out a pointer type like so: `<pointed type>*`. For example `int*` is a pointer to `int` type.

Tips

The * indicates that we are adding a level of indirection to access an `int`.

You can hold a pointer inside a variable, which introduces the following syntax for such a variable declaration: `<pointed type> *<var name>;`.

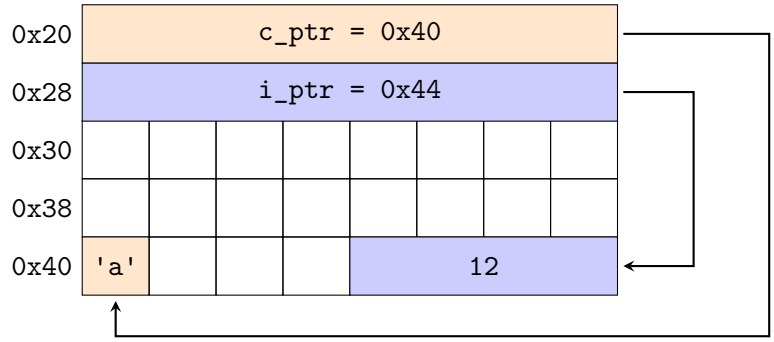
```
char *c_ptr;
int *i_ptr;
```

Here we declared a variable named `c_ptr` whose type is `pointer to char`, and `i_ptr` whose type is `pointer to int`.

Like any other variable, you can assign a value to your pointer. To do so, you just need an address to assign to your variable with the correct associated type. We can use those variables to point to `the` and `answer` respectively.

`<type> *<variable> = <address>.`

```
char *c_ptr = 0x40; /* Address: 0x20, value 0x40 */
int *i_ptr = 0x44; /* Address: 0x28, value 0x44 */
```



Tips

As you can see on the above diagram, pointer variables take space in memory too, which makes sense because they are variables. Because pointers are also a type, they have a size. On the PIE it turns out that pointer variables takes eight bytes of memory space.

Be careful!

Because memory addresses are not guaranteed to stay the same, you should never hard-code them directly into your code. We are only showing you this for the example.

2.2 Variable address

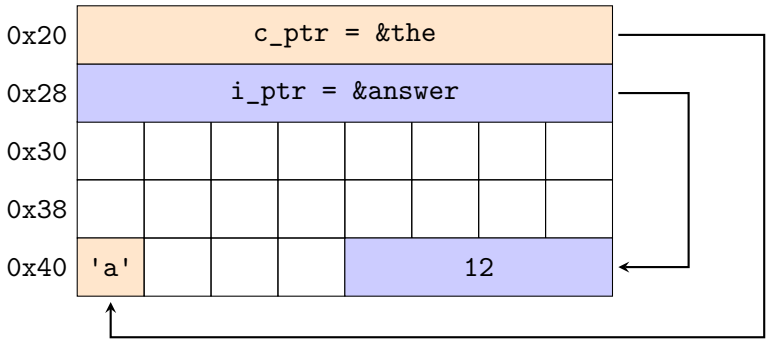
Where can we find an address to assign to a pointer? As you just saw, any variable in your program lives somewhere in the computer's memory. You can, therefore, use its address for your pointer.

To get the address of a variable, we need to use the operator `&`. For example, getting the address of our variable `answer` would return `0x44`.

```
&the; /* 0x40 pointing to char */
&answer; /* 0x44 pointing to int */
```

Now that we know the basics of pointers and how to get a variable's address, we can initialize a pointer variable with another variable's address:

```
c_ptr = &the;
i_ptr = &answer;
```



Tips

You will see other ways to get valid memory addresses to point to later.

2.3 Dereferencing

Pointers allow us to manipulate memory. At some point we need access to the value at this address: this is called **dereferencing**.

The dereferencing syntax is `*<pointer>`.

Be careful!

Be careful, do not mistake `*ptr;` (dereferencing) for a pointer declaration like `int *ptr;`

```
int foo = *i_ptr; /* Address 0x48 , value 12 */
foo += 1; /* foo: value to 13, answer: value to 12 (unchanged) */
*i_ptr += 2; /* foo and i_ptr values do not change, answer: value changes to 14 */
```

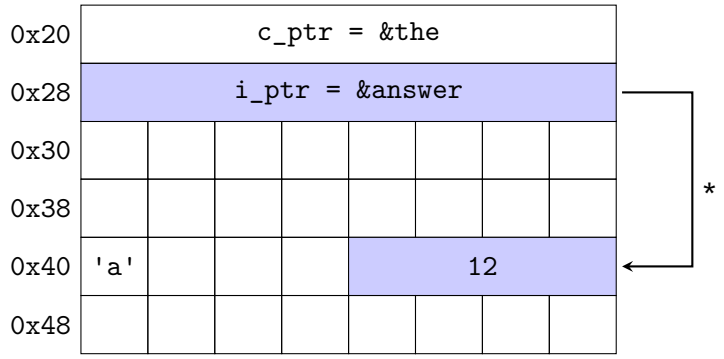


Fig. 1: `i_ptr` is dereferenced, accessing the value at address `0x44`

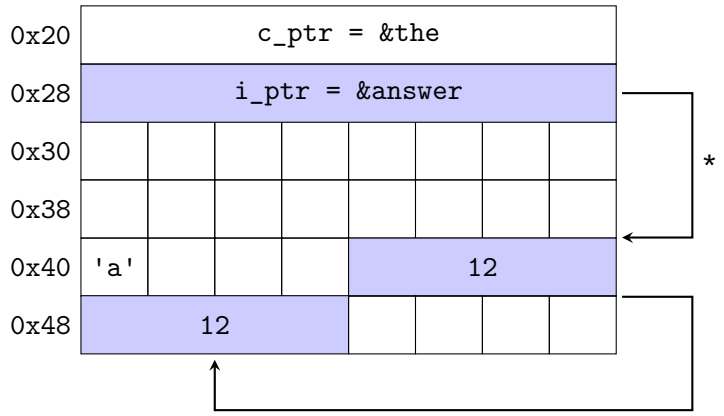


Fig. 2: The value at `0x44` is copied in `foo`, at `0x48`

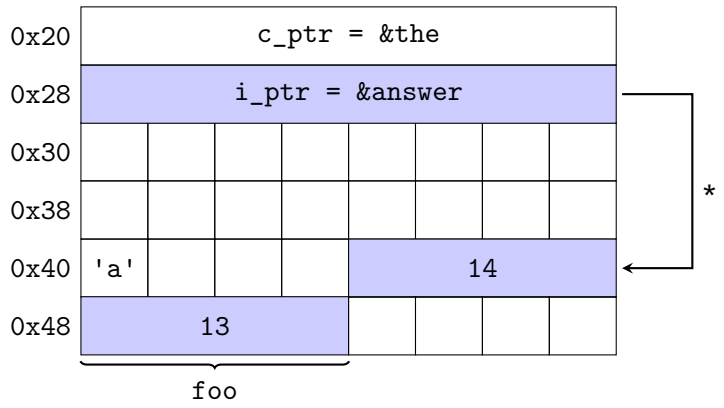


Fig. 3: `foo` and `*i_ptr` are different values

2.3.1 Get int value

Write a function that takes a pointer to an `int` as parameter and returns its value. You don't have to handle the case where the pointer is `NULL`.

```
int get_int_value(int *n)
```

2.4 Variable address & dereferencing: Practical example

Please compile the following pieces of code without the `-pedantic` flag. Otherwise, you would get a warning when printing `&x`. Be careful, this is for illustrative purposes only, you will never have to print `&x` again. Do not forget to compile with the `-pedantic` flag in other situations. What does the following code print?

```c
#include <stdio.h>

int main(void)
{
    int x = 42;
    printf("%d\n", x);  /* show the value of x */
    printf("%p\n", &x); /* show the address of x */
    return 0;
}
```

Here, `&x` corresponds to the address of the `x` variable. Instead of its value, it returns a pointer to the variable (notice the `%p` in `printf(3)`).

```c
#include <stdio.h>

int main(void)
{
    int x = 42;
    int *ptr_x = &x;
    printf("%d\n", x);      /* shows the value of x */
    printf("%p\n", &x);     /* shows the address of x */
    printf("%p\n", ptr_x);  /* shows the value of ptr_x (which is the address of x) */
    printf("%d\n", *ptr_x); /* shows the value of what ptr_x points to: x */
    return 0;
}
```

`int*` is a pointer to an integer and here we call it `ptr_x`. The `ptr_x` pointer is then set to point to the address of `x` which is `&x`. Calling a pointer with the `*` dereferences the pointer and accesses the pointed value. Indeed, `*ptr_x` returns the value that the address `&x` points to, instead of the numerical value of the address.

2.5 Passed by copy

In *C*, variables are passed **by copy**. This means that the parameters will be **copied** for the function, and therefore will have a different address. Here is an example:

```c
void do_the_magic(int i, int j)
{
    /* i ->           Value:   42      Address: Somewhere else in the memory */
    /* j ->           Value:   51      Address: Somewhere else in the memory */
    i = 12;
    j = 27;

    printf("i: %d, j: %d\n", i, j); // Prints "i: 12, j: 27"
}

int main(void)
{
    int foo = 42;  /* Value:   42      Address: 0x52 */
    int bar = 51;  /* Value:   51      Address: 0x35 */

    printf("foo: %d, bar: %d\n", foo, bar); // Prints "foo: 42, bar 51"
    do_the_magic(foo, bar);
    printf("foo: %d, bar: %d\n", foo, bar); // Prints "foo: 42, bar 51"

    return 0;
}
```

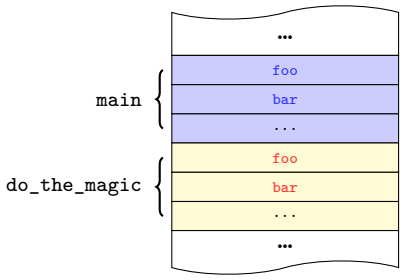


Fig. 4: Local copies are modified.

Because we are passing them by copy, `i` and `j` inside `do_the_magic` do not have the same address as `foo` and `bar`. But we want to reference the same address, therefore we need to provide their address to the function: by using pointers.

```c
void do_the_magic(int *i, int *j)
{
    /* *i ->          Value:   42      Address: 0x52                          */
    /* *j ->          Value:   51      Address: 0x35                          */
    /* i  ->          Value:   0x52    Address: Somewhere else in the memory */
    /* j  ->          Value:   0x35    Address: Somewhere else in the memory */
    *i = 12;
    *j = 27;
    printf("*i: %d, *j: %d\n", *i, *j); // Prints "*i: 12, *j: 27"
}
```

(continues on next page)

(continued from previous page)

```c
int main(void)
{
    int foo = 42;  /* Value:   42      Address: 0x52 */
    int bar = 51;  /* Value:   51      Address: 0x35 */

    printf("foo: %d, bar: %d\n", foo, bar); // Prints "foo: 42, bar 51"
    do_the_magic(&foo, &bar);
    printf("foo: %d, bar: %d\n", foo, bar); // Prints "foo: 12, bar 27"

    return 0;
}
```

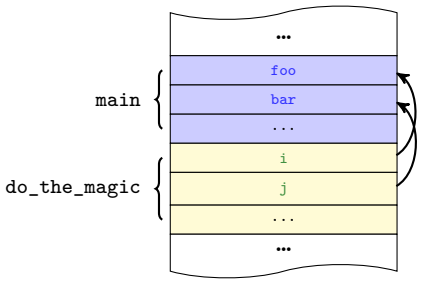


Fig. 5: Have pointers to `foo` and `bar` variables to modify their values in the `main` context.

Be careful!

When writing `int *i`, we are declaring a variable named `i` of type `int *`; when writing `*i`, we are dereferencing the variable `i`.

Here `foo` and `bar` are passed by address.

2.5.1 Practical example

```c
void ineffective_swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

int main(void)
{
    int a = 42;
    int b = 51;

    ineffective_swap(a, b);   /* No effect. */
    printf("%d %d\n", a, b);  /* 42 51 */
    return 0;
}
```

This `ineffective_swap` function has no effect because the arguments are passed by copy.

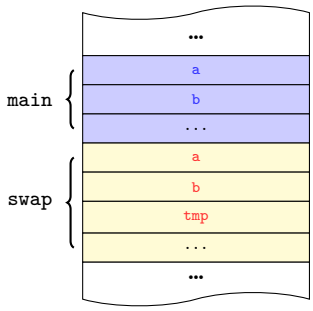


Fig. 6: Local copies are swapped.

Here, pointers offer us a solution:

```c
void swap(int *pa, int *pb)
{
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}

int main(void)
{
    int a = 42;
    int b = 51;

    swap(&a, &b);             /* a's value and b's value are switched! */
    printf("%d %d\n", a, b); /* 51 42 */
    return 0;
}
```

The two arguments of `swap` are again passed by copy, but this time, the copied values are two pointers to integers (`int*`), not integers (`int`). Then in the `swap` function, we *dereference* those pointers to modify the value at the memory location they point to. In the above example, those two pointers contains the memory addresses of the `a` and `b` variable declared in the `main` function, so the `swap` function will effectively swap the values of `a` and `b`.

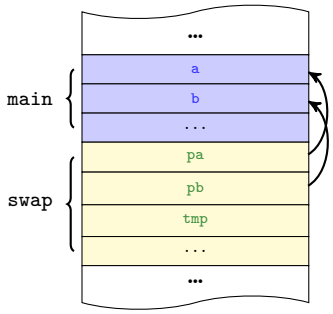


Fig. 7: Have pointers to `a` and `b` variables to swap their values in the `main` context.

2.6 NULL

The following code:

```
int *foo; /* not initialized */
int bar = *foo;
```

Is **undefined behavior**, meaning the *C* Language Specification has not specified any particular behavior when dereferencing a pointer variable that was initialized without a value (just like any variable). Thus, you should always initialize it either with a valid address:

```
int bar = 42;
int *foo = &bar;
```

Or the `NULL` macro:

```
int *foo = NULL;
```

You cannot dereference a `NULL` pointer, or you will have a **segmentation fault**:

```
int *foo = NULL;
int bar = *foo; /* segfault */
```

Tips

A segmentation fault is a specific type of error where you try to access some memory which you do not have access to.

The `NULL` macro usually corresponds to `0x0` the first address in your memory space, and thus it evaluates to `false`:

```
int *foo = NULL;

if (!foo)
    printf("Foo is NULL.\n");
else
    printf("Foo is not NULL.\n");

/* Will print "Foo is NULL.\n". */
```

Tips

When we say you should always initialize your pointers, it means you **must** always initialize your pointers. If you dereference a pointer that was not initialized, the outcome is *undefined*. It may work, it may not work, or it may segfault. That random behavior will definitely not help during debugging (it is way easier to debug a guaranteed segfault, rather than a random segfault).

2.7 Array notation & pointer arithmetic

We have seen before that we cannot predict the memory location of a variable in advance. However, we can make an assumption with arrays: all elements are contiguous in memory.

```
int arr[] = { 12, 27, 42, 51 };
// the array is stored between 0x50 and 0x60
```

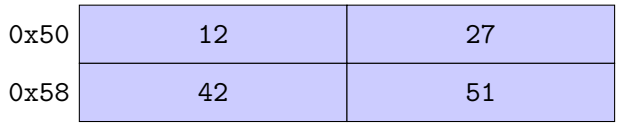


Fig. 8: The array is contiguous in memory, starting at 0x50

To access an element of an array, you use the `[<index>]` operator, the first element being at index 0, the second at index 1, etc...

If `arr[0]` is located at `0x50` then, `arr[1]` would be at `0x54`, and `arr[2]` at `0x58` (remember that an `int` is four bytes long on the PIE). This is the reason why an array can only contain elements of a single type. By knowing the array elements' type we know their size and how to access them at any index in the array.

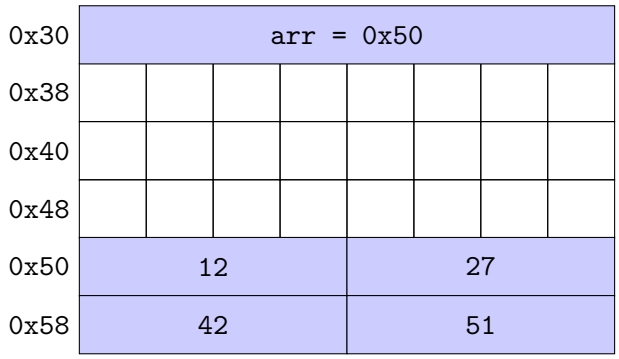


Fig. 9: `arr` contains the address of the first value of the array, `0x50`

```
i_ptr = arr;      /* i_ptr = arr = 0x50 */
arr[0] == i_ptr[0]; /* true */
&arr[0] == &i_ptr[0]; /* true */
```

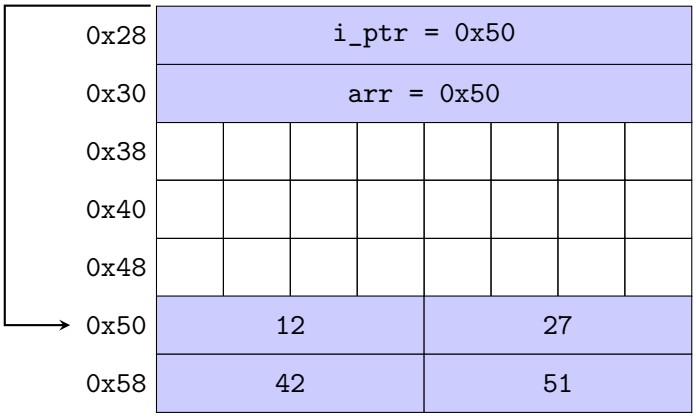


Fig. 10: `i_ptr` is now pointing to `0x50`

Note from the above code that pointers can be manipulated like an array whose first element is at the address being pointed to. Indeed they contain the same information, namely an address associated with a type (giving us the starting element, and the size of each element).

The reason we need the type of an array's elements is that when trying to access the n-th element of that array, we need to know at which offset in memory it is from the first element of the array to retrieve the queried value.

Arrays being contiguous in memory, they allow an easier manipulation of the memory by grouping its elements in one group, making them easy to index in relation to one-another.

```
i_ptr = arr + 1;
arr[1] == *i_ptr; /* true */
&arr[1] == i_ptr; /* true */
```

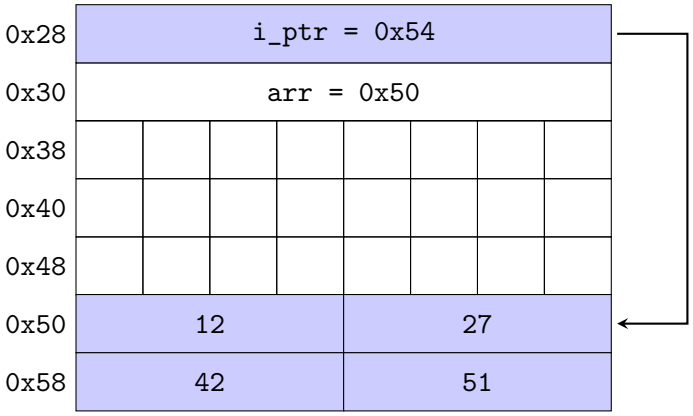


Fig. 11: `i_ptr` is now pointing to `0x54`

The operation `arr + 1` tries to compute the memory address corresponding to the element that comes right after `arr`'s beginning (i.e: its second element). The `+i` means you want to access the i-th element of the array. To do so, we first calculate the address of that element by using its size and the array's starting point, we can offset the address of the first element by `i * <size of an element>`, getting the address of the i-th element. At that point we can access the memory containing that element and manipulate its value.

So basically, just like you can use pointers to access elements of a type in memory, you can use array notation to do the same operation. The pointer notation (using `*`) and array notation with brackets (`[0]`) are mutually interchangeable.

Going further...

So, when using `a[i]` you are doing the same operation as when you write `*(a + i)` (which is also perfectly valid). However, using the array notation usually makes your code easier to read and reason about.

This also means that an array, when passed as arguments to a function, is not copied, only the pointer to its first element is copied. Thus every modifications done to the array in the function will persist.

Here are some examples:

```
void array_modifier(int *arr)
{
    arr[0] = 14;
    arr[1] = 15;
}

// After a call to `array_modifier` the elements will still be modified
```

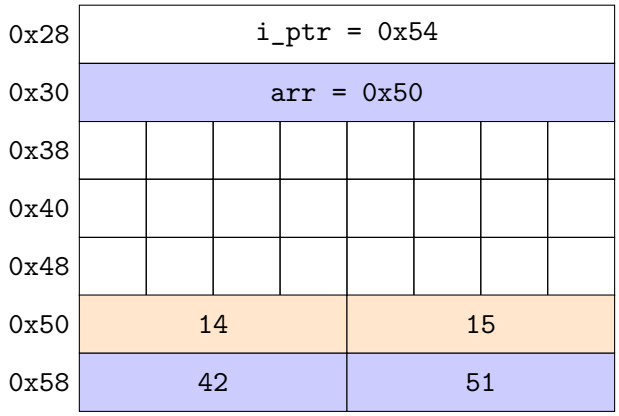


Fig. 12: Values at `0x50` and `0x54` are changed to `14` and `15` respectively

Going further...

You can also subtract two pointers of the same type, resulting in the number of elements of the type being pointed to between those two addresses.

```
&(arr[3]) - &(arr[0]); // The result is 3
```

2.8 Application

2.8.1 my_strlen

Now that we know more about pointers, strings, and the `NULL` macro, we can re-implement some known functions. Earlier today, you learned about the `strlen(3)` function which allows you to get the length of a given string.

```c
#include <stddef.h>

size_t my_strlen(const char *str)
{
    if (!str)
        return 0;

    size_t i = 0;

    while (str[i] != '\0')
        ++i;

    return i;
}
```

The `while`'s condition is made of two parts:

1. We test that `str` is `true`, i.e. it is not a `NULL` pointer. Otherwise we return 0.
2. We check that we have not yet reached the null-terminating character `\0` at the index `i`.

2.8.2 array_min_max

Goal

Write a function that takes an array `tab` of integers and its length `len`, along with two pointers and set the value of the two pointers to the maximum and minimum of the array.

If `tab` is `NULL` or `len` is equal to 0, the function does nothing.

Prototype:

```c
void array_max_min(int tab[], size_t len, int *max, int *min)
```

Example

```c
int main(void)
{
    int max = 0;
    int min = 0;
    int tab[] = { 5, 3, 1, 42, 53, 3, 47 };
    size_t len = 7;
```

(continues on next page)

(continued from previous page)

```
    array_max_min(tab, len, &max, &min);
    printf("max : %d\n", max);
    printf("min : %d\n", min);

    return 0;
}
```

Output:

```
42sh$ ./array_max_min
max : 53
min : 1
```

3 The `main` function

3.1 Definition

The `main` function is the first function to be executed when a *C* program is launched: this function is called the entry point of the program. Every *C* program must contain a single `main` function.

You already learnt the `main` function prototype when no arguments are to be passed to the program:

```
int main(void)
```

However, if arguments are to be passed to the program, the prototype of the `main` function will be as follows:

```
int main(int argc, char *argv[])
```

`argc` contains the number of arguments given to the program, plus one (because the first argument is the program's name), and `argv` is an array of strings. Here is an example of passing arguments to a program:

Tips

`argv` can also be declared as `char **argv`, you will understand this once you become familiar with dynamic arrays.

```
42sh$ ./path/to/my_prog toto titi tata
```

In the above example, `argc` will have the value `4` and `argv` will contain:

- `argv[0] = "./path/to/my_prog"`
- `argv[1] = "toto"`
- `argv[2] = "titi"`
- `argv[3] = "tata"`
- `argv[4] = NULL`

Tips

`argv` is always `NULL` terminated, meaning `argv[argc]` is always `NULL`.

3.2 Exercises

3.2.1 Print arguments

Write a program using `puts`(3) that displays all the arguments in `argv`, except `argv[0]`.

```
42sh$ ./print_arguments | cat -e
42sh$ ./print_arguments Epita my new home | cat -e
Epita$
my$
new$
home$
42sh$
```

It is my job to make sure you do yours.