PROG C — td3

version **#0.0.1-dirty**

ASSISTANTS C/UNIX 2024 <assistants@tickets.forge.epita.fr>

Copyright

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

Contents

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

1 Recursion

1.1 Introduction on Algorithms

An algorithm is, by essence, a sequence of instructions, describing how to perform a task. In the field of computer science, we could be more specific when defining this notion. One must understand that an algorithm always has a purpose, it is written to solve a precise and well defined issue. Moreover, one must keep in mind that efficiency is a key goal to factor in when writing algorithms.

> [...] we want good algorithms in some loosely defined aesthetic sense. One criterion [...] is the length of time taken to perform the algorithm [...]. Other criteria are adaptability of the algorithm to computers, its simplicity and elegance, etc
>
> ---*The Art of Computer Programming: Introduction to Algorithms* - **Donald Knuth**

1.2 Definition

> Many useful algorithms are **recursive** in structure: to solve a given problem, they call themselves one or more times to deal with closely related subproblems. [...] They break the problem into several subproblems that are similar to the original problem but smaller in size, [...] and then combine these solutions to create a solution to the original problem.
>
> ---*Introduction to Algorithms (3rd edition)* - **Cormen, Leiserson, Rivest, Stein**

In other words, recursion is a programming technique where a function or an algorithm **calls itself** until a stopping condition is met.

At each call, the input changes from the last input call, until it matches the stopping condition. At this point, the function returns a value and the recursion is completed, from the last call to the first one.

Thus, a generic pattern of a recursive function can be defined with four different parts:

1. A **stopping condition** (to prevent infinite recursion)
2. Pre-order operation (before the `recursive call`)
3. The **recursive call**
4. Post-order operation (after the `recursive call`)

1.3 Example: print_sequence

Here is an example:

```c
#include <stdio.h>

void print_sequence(int x)
{
    if (-1 == x) /* Stopping condition */
        return;
```

(continues on next page)

(continued from previous page)

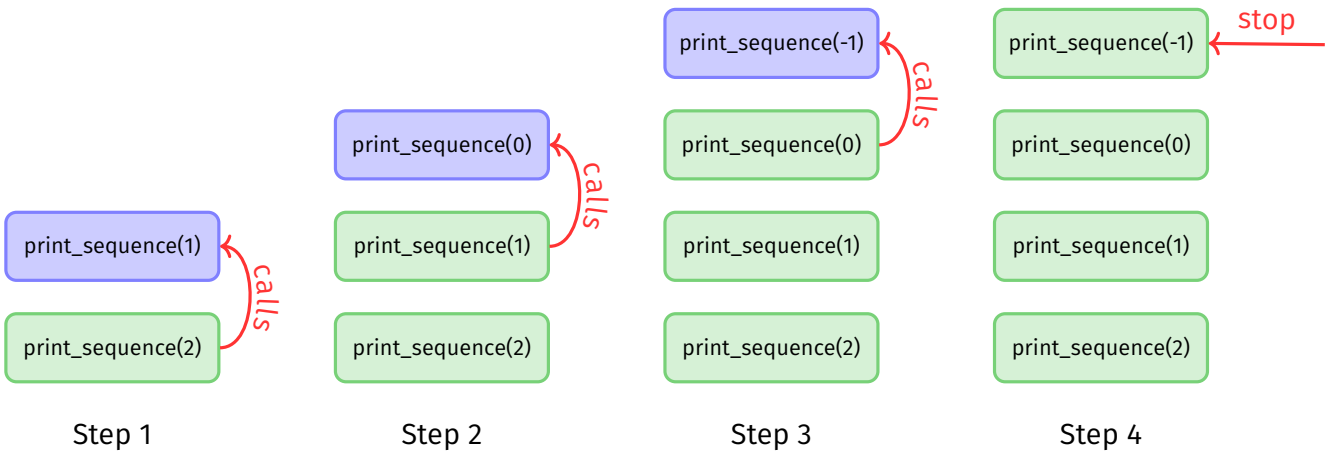
```
    // A pre-order operation could be here
    print_sequence(x - 1); /* Recursive call */
    printf("%d\n", x); /* Post-order operation */
}

int main(void)
{
    print_sequence(100);
    return 0;
}
```

As you see above, the function `print_sequence(int x)` has a condition that will stop the recursion when `-1 == x` and a call to itself `print_sequence(x - 1);` inside its body. Let us see what it does:

```
42$ ./print_sequence
0
1
2
3
4
[...]
97
98
99
100
```

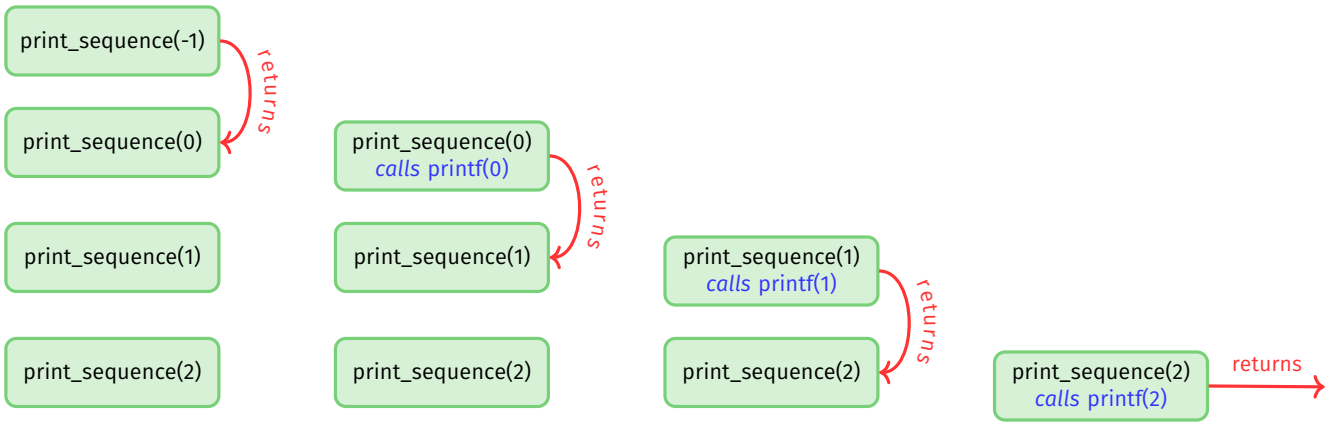
The function will print all integers from `0` to `x`.

Let us see what happens in detail what happens if we call `print_sequence(2)`.



- In Step 1, we have the first call to `print_sequence(int x)` with `x = 2`. It calls `print_sequence(1)`.
- In Step 2, the same happens, except it is `print_sequence(0)` that is called.
- In Step 3, `print_sequence(0)` calls `print_sequence(-1)`.
- In Step 4, `print_sequence(-1)` reaches the stopping condition (`-1 == x`).

Thus, it returns.



- In Step 5, as `print_sequence(-1)` is fully executed, `print_sequence(0)` can be processed.
- In Step 6, `print_sequence(0)` executes the next instruction, which is `printf(0)`, and then it returns.
- In Step 7, same as Step 6, it executes `printf(1)`, and returns.
- In Step 8, finally, `print_sequence(2)` will print 2 and return.

The recursion ends.

1.4 Example: print_sequencev2

Let us continue the previous example:

```
void print_sequencev2(int x)
{
    if (-1 == x) /* Stopping condition */
        return;

    printf("%d\n", x); /* Pre-order operation */
    print_sequencev2(x - 1); // Recursive call
}
```

The function `print_sequencev2(int x)` is a little different than `print_sequence(int x)`. First we print the integer, then we make the recursive call. Try to understand why the two functions have different execution flows.

The examples above demonstrate that the instructions before the recursion are executed from the first call to the last call. Whatever comes after the recursion is executed from the last call to the first call.

It is crucial to have a stopping condition in recursive functions. Try the following function `infinite_recursion(void)` below and see what happens.

```c
void infinite_recursion(void)
{
    infinite_recursion();
}

int main(void)
{
    infinite_recursion();
    return 0;
}
```

When you execute this program, it will respond with `Segmentation fault (core dumped)`. Each call to the recursive function will occupy space in memory until the maximum capacity is exceeded, causing the program to be aborted.

Tips

In case the execution of your program is too long, you can press `Ctrl + C` in your terminal in order to stop the execution.

Going further...

Later you will see what a `Segmentation fault` is and how it is triggered. For now, you just need to know that a `Segmentation fault` is triggered because your program has had a problem with its memory.

Here is another example with a function returning a numerical value, computing the n-th power of two:

```c
int pow_of_two(unsigned n)
{
    if (0 == n) /* Stopping condition */
        return 1;

    return 2 * pow_of_two(n - 1);
}

int main(void)
{
    int res = pow_of_two(3);
    printf("Result : %d\n", res);
    return 0;
}
```

Tips

The `unsigned` qualifier means that the variable can only take positive values.

Going further...

If you try to run the above program with the value 31, you will cause an **overflow**. It happens when you try to store a value larger than the maximum value the variable can hold.

```c
// int takes value from [-2^31, 2^31 - 1] or [-2 147 483 648, 2 147 483 647]
int i = 2 147 483 647; /* i = 2^31 - 1 */
i = i + 1; /* i = -2 147 483 648 */
// Here i = 2^31 - 1 + 1 = 2^31. However int takes value until 2^31 - 1.
// Hence, here i overflowed and started again the range of possible value.
// It took the first value available, that is -2^31.

// char takes value from [-2^7, 2^7 - 1] or [-128, 127]
char c = 120;
c = c + 30; /* 120 + 30 = 128 + 22 = -128 + 22 = - 106 */
```

Be careful!

Take special care about the stopping condition of your recursion. You need to handle all the cases properly, otherwise you will have an infinite recursion with the same error as above (`Segmentation fault`). For example, what happens if, in the last example, you call `pow_of_two` with the value `-2` ?

Going further...

Let us talk about tail recursion.

A tail recursive function is a function in which recursive calls are the last evaluated expressions. As such, a recursive call cannot be part of another expression.

Let us takes the factorial function as example:

```c
unsigned long facto(unsigned long n)
{
    if (n <= 1)
        return n;
    return n * facto(n - 1);
}
```

In this example, `facto(n - 1)` is not the last evaluated expression, it is `n * facto(n - 1)`. Therefore this function cannot be considered as a tail recursive function.

But, why tail recursive function are different from normal recursive function ?

Some compilers and interpreters can use these functions to perform an optimization called `tail call optimization`. They will transform the tail recursive function into an iterative process, a loop.

This optimization solves the two mains issues of recursion: the performance and the call stack limitation.

The performance issue is about the cost of a function call, you may not see it now, but function calls are expensive operations compared to a loop. Therefore, for the same computation, recursion will be slower than iteration.

About the call stack limitation. Your computer has a range of reserved memory called the **stack**. This is where some variables are stored and where function calls are stored too.

When multiple function calls are stacked in the stack we call this a **call stack**. You have an example of **call stack** in the diagrams above. Recursion can create huge call stack and this is a problem because the stack has a limited size. Therefore, if you stack more function calls than the stack can

handles, it will lead to a stack-overflow error. You don't have this problem with loops because it does not need the stack.

That is why **tail call optimization** and **tail recursive** functions are great. With the translation into a loop, we erase the function calls, which removes all risks of stack-overflow errors and increases the performance of the function.

Here is by example the tail recursive version of `factorial`:

```
unsigned long facto(unsigned long n, unsigned long result)
{
    if (n <= 1)
        return result;
    return facto(n - 1, n * result);
}
```

1.5 Exercises

1.5.1 Fact and fibo

For this part, you should do the exercises `Factorial` and `Fibonacci`.

Fact

Goal

Implement the function `fact` with the following prototype:

```
unsigned long fact(unsigned n);
```

It computes the factorial of `n` ($n!$) and returns the result. As a recall:

$$\forall n \in \mathbb{N}, n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n \neq 0 \end{cases}$$

You **must** use recursion (i.e.: loops are forbidden).

Fibo

Goal

Write the function that computes the *Fibonacci sequence* recursively. This sequence is defined by U_n as follow:

- $U_0 = 0$
- $U_1 = 1$
- $U_n = U_{n-1} + U_{n-2}$

```c
unsigned long fibonacci(unsigned long n);
```

Practice

Now that you have written fact and fibo in recursive, try to rewrite these programs in iterative.

Tips

The difference between a recursive and iterative program is that the recursive one calls itself, to execute instructions, while the iterative one uses loops.

2 Pointer

2.1 Pointers

2.1.1 Introduction

Be careful!

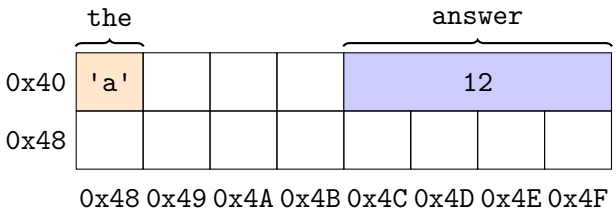
Pointers are a fundamental concept, pay extra attention!

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 combo is called a pointer. The type associated to an address is used to know the size of the variable stored.

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 an `int` variable.

Tips

`<pointed type>*` and `<pointed type> *` are both syntaxically correct. However, you will have to use the latter during this semester.

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;
```

Be careful!

You must always initialize your pointer variable. Otherwise you expose yourself to errors that you do not want to debug. The default case where you do not know the value of the pointer at initialiszation will be covered later on.

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

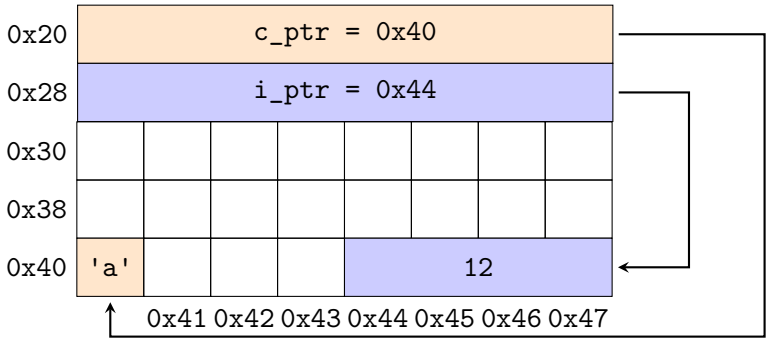
Like any other variable, you can assign a value to your pointer. To do so, you just need an address to assign to your variable with the correct associated type. We can use those variables to point to `the` and `answer` respectively.

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

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

Be careful!

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



As you can see on the above diagram, pointer variables take space in memory too, which makes sense because they are variables.

Tips

Because pointers are also a type, they have a size. On the PIE it turns out that pointer variables take eight bytes of memory space.

2.1.2 Initialization

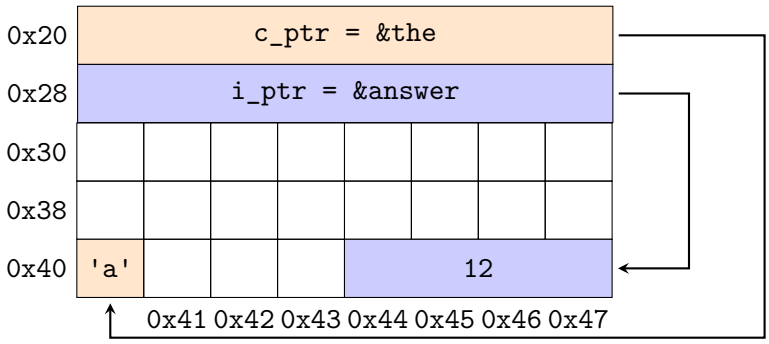
Where can we find an address to assign to a pointer? As you just saw, any variable in your program lives somewhere in the computer's memory. You can, therefore, use its address for your pointer.

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.

2.1.3 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;`

```c
int answer = 12; /* address: 0x44 */
int *i_ptr = &answer; /* Address: 0x28, value 0x44 */
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 */
```

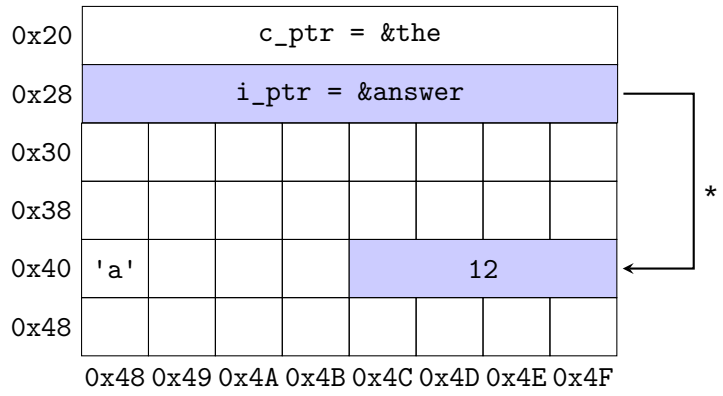


Fig. 1: `i_ptr` is dereferenced, accessing the value at address `0x44`

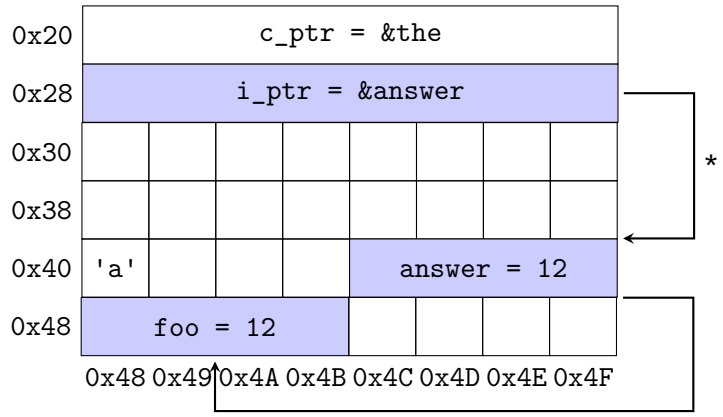


Fig. 2: The value at `0x44` is copied in `foo`, at `0x48`

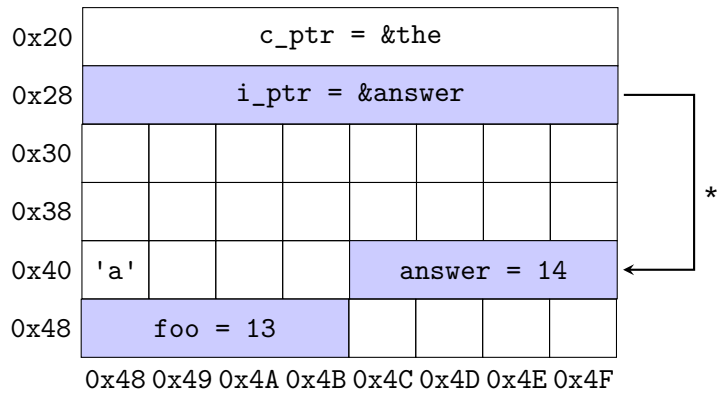


Fig. 3: `foo` and `*i_ptr` are different values

Get int value

Goal

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);
```

2.1.4 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 won't ever need to print `&x` again. Don't 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 operator `*` **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.

2.1.5 Passed by copy or by reference

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. It will create a copy of the variable and pass this copy to the function. 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: 0x48 */
    int bar = 51;  /* Value:   51      Address: 0x4C */

    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;
}
```

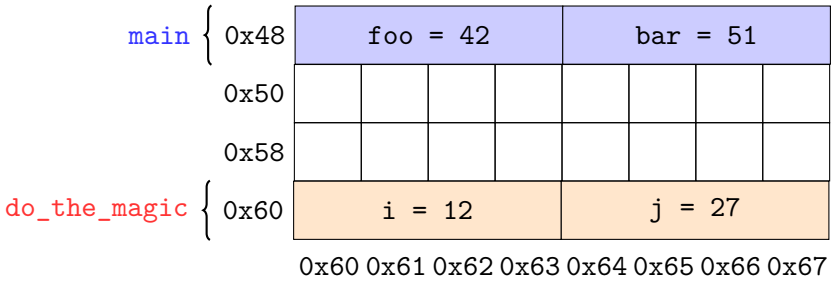


Fig. 4: Local copies are different variables in memory.

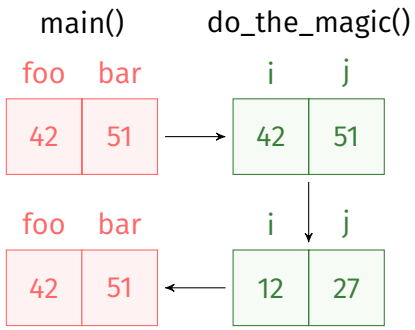


Fig. 5: Passing by copy to `do_the_magic()`

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 instead. The pointers will also be passed by copy as all function argument in **C** are passed by copy. The pointed value will remain the same and could be edited inside the function.

```c
void do_the_magic(int *i, int *j)
{
    // *i ->            Value:   42      Address: 0x48
    // *j ->            Value:   51      Address: 0x4C
    // i  ->            Value:   0x48    Address: Somewhere else in the memory
    // j  ->            Value:   0x4C    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: 0x48 */
    int bar = 51;  /* Value:   51      Address: 0x4C */

    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;
}
```

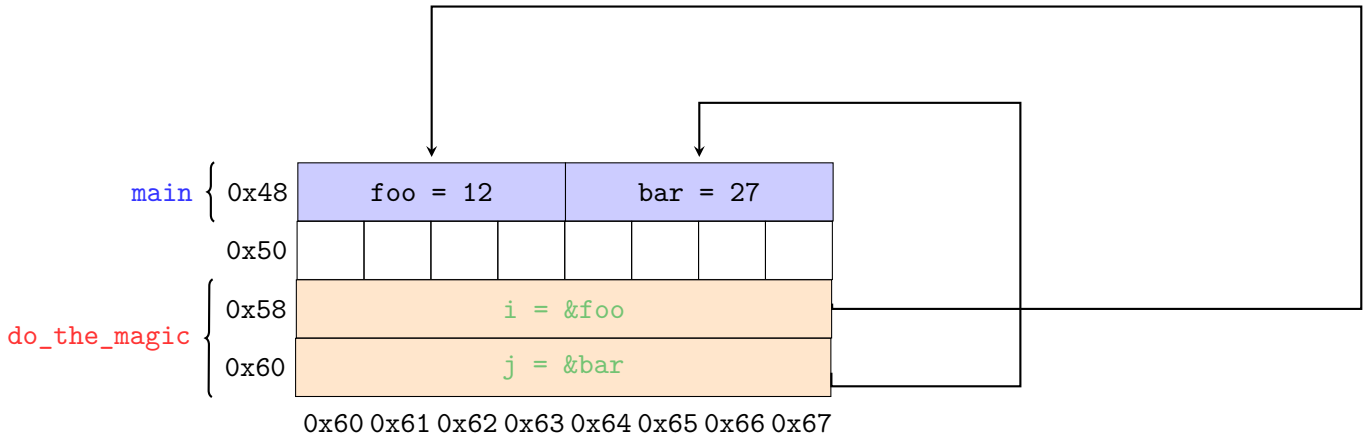


Fig. 6: Have pointers to `foo` and `bar` variables to modify their values in the `main` context.

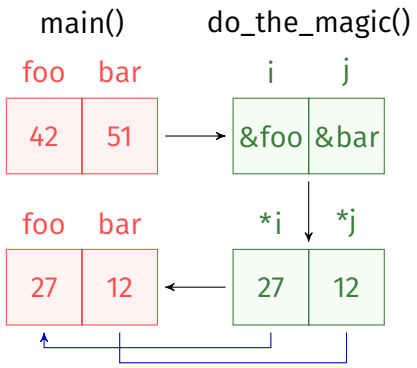


Fig. 7: Passing by reference to `do_the_magic()`

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 pointer.

Tips

Passing an argument by pointer implies passing the address of the variable instead of the variable itself.

Practical example

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

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

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

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

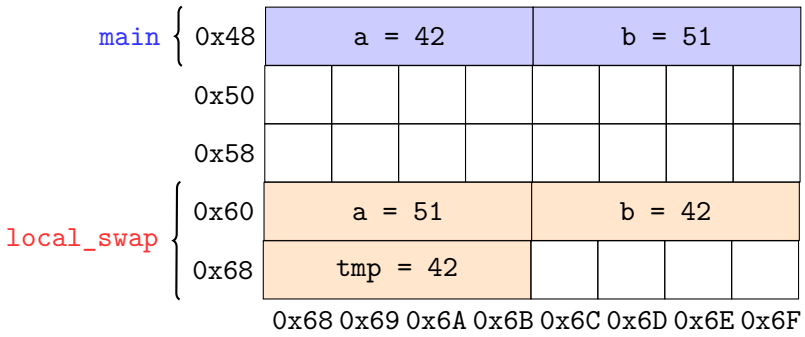


Fig. 8: Local copies are swapped.

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 contain the memory addresses of the `a` and `b` variables declared in the `main` function, so the `swap` function will effectively swap the values of `a` and `b`.

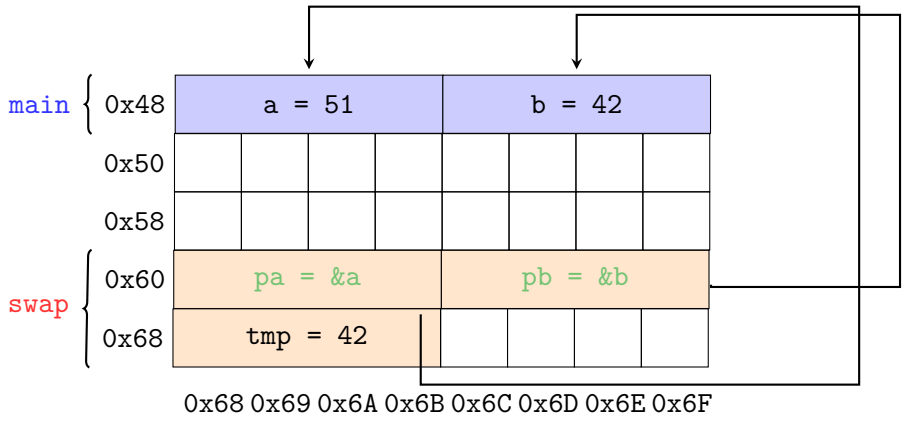


Fig. 9: Have pointers to `a` and `b` variables to swap their values in the `main` context.

2.1.6 NULL

The following code:

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

Is an **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` special value:

```
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` point constant always evaluate to zero, *0x0* being the first address in your memory space. It corresponds to nothing and thus it evaluates to `false`:

```
int *foo = NULL;

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

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

Be careful!

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 behaviour will definitely not help during debugging (it is way easier to debug a guaranteed segfault, rather than a random segfault).

2.1.7 Array notation

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
```

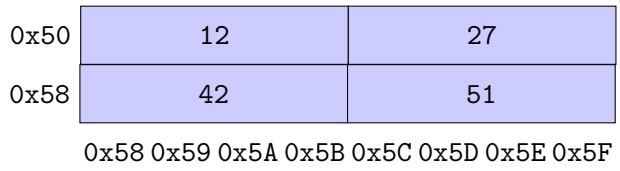


Fig. 10: The array is contiguous in memory, starting at 0x50

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). 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.

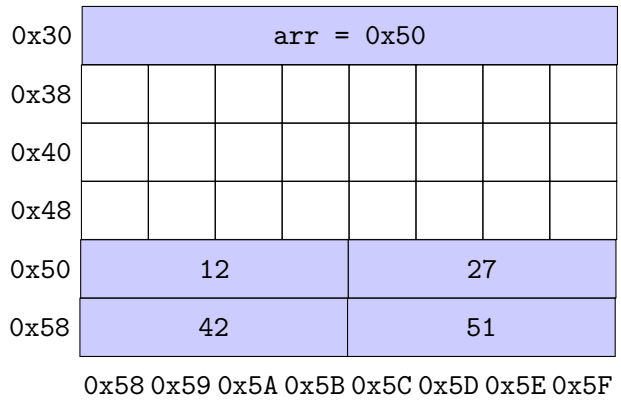


Fig. 11: `arr` contains the address of the first value of the array, `0x50`

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

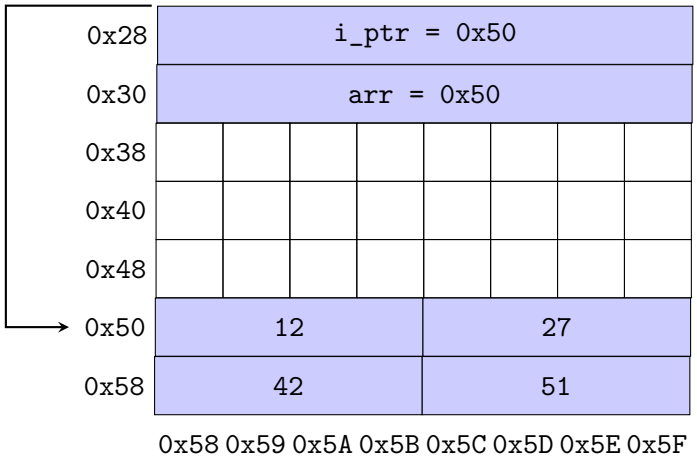


Fig. 12: `i_ptr` is now pointing to `0x50`

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 acces 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 */
```

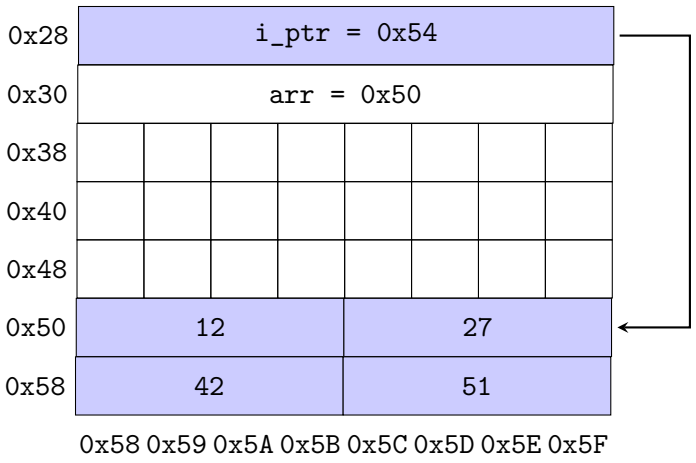


Fig. 13: `i_ptr` is now pointing to `0x54`

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 the array notation to do the same operation. The pointer notation (`*`) and the arrays notation with brackets (`[0]`) are mutually interchangeable.

Going further...

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.

```
i_ptr = arr;
arr[2] == *(i_ptr + 2); /* true */
&arr[4] == &arr[3] + 1; /* true */
```

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:

```
arr[0] = 14;
arr[1] = 15;
```

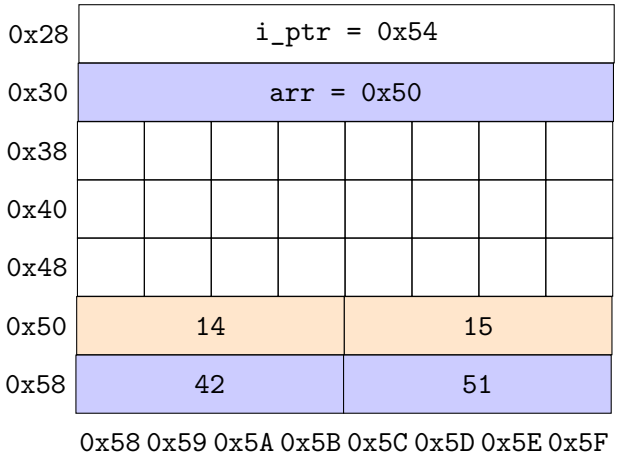


Fig. 14: Values at `0x50` and `0x54` are changed to `14` and `15` respectively

2.1.8 Application

Now that we know more about pointers, strings, and the `NULL` macro, we can reimplement some known functions. You've seen previously the function `strlen(3)` which allows you, given a string, to know its length.

```
#include <stddef.h>

size_t my_strlen(const char *str)
{
```

(continues on next page)

(continued from previous page)

```
    if (!str)
        return 0;

    size_t i = 0;

    while (str[i] != '\0')
        i += 1;

    return i;
}
```

Be careful!

Take special care of the type of the parameter `str`. Here, the function takes by copy the pointer str and the pointed type is const. Hence, this function can not modify the data pointed by str.

Two conditions are checked in this function :

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

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:

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

Example

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

    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
```

Going further...

Remember that you have seen two different prototypes of `main`: `void main(void)` and `void main(int argc, char** argv)`. You have already seen the notation `char **`. Simply put it is a pointer that points to a `char *`. Pointer are easily stackable. When you add a `*`, it simply means you are a pointer that points to the type left of the `*`. The left type can either be another pointer, a structure or a primitive type.

I must not fear. Fear is the mind-killer.