Prog C — prog-c

version **#0.0.1**

Assistants C/Unix 2025 <assistants@tickets.assistants.epita.fr>

Copyright

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

Contents

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

1 Core Language

Look at your first *C* file:

```c
/**
** \file hello.c
*/

#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello World!\n");

    return 0;
}
```

Congratulations, you just read some *C*! It is ok if you do not understand everything, we will explain it in details in the tutorial.

Copy the code above to a file named `hello.c` You can compile and execute it with the following command:

```
42sh$ gcc hello.c -o hello
42sh$ ./hello
Hello World!
```

Be careful!

Here, `42sh$` represents your shell prompt. It looks like `[xavier.login@r02p09 ~]$` on your computer.

Even if you have never read *C* code, try to understand the purpose of these lines. Do not worry if it does not make sense yet, we will explain this code step by step.

Be careful!

Keep `hello.c` opened, as the whole tutorial will explain different notions taking it as an example.

You may notice that each line ends with a ";". In *C*, it is the instruction separator. Each instruction must be separated by a ";". Without the semicolon, the code does not make sense anymore. It would be similar to removing every punctuation mark in a sentence.

1.1 The C language

C is a computer programming language. A programming language is a set of instructions that allows you to write a program. *C* was created in 1972 by Dennis Ritchie and is widely used thanks to its portability (the fact that it can be run on computers with different architectures). The version of *C* you will use all year is *C99*, you must compile with the option `-std=c99` to do so. This workshop is entirely dedicated to this language.

You will soon have to read a lot of *C* code. You have used *Python* before, and you might have already noticed an important difference between *C* and *Python*: **compilation**.

As opposed to *Python* which is an interpreted language, *C* is a **compiled** language. You cannot run a *C* source file directly: you have to give it to a *compiler*, whose job is to translate your *C* file into an *executable* file. In practice, all computers do not run the same way, so the *compiler* will adapt the executable file depending on the computer's architecture. There are a lot of different compilers, and the one we will mostly use is *gcc*.

Copy the code below in a file named `file.c`:

```c
#include <stdio.h>

int count_vowels(char str[])
{
    int i = 0;
    int count = 0;

    while (str[i] != '\0')
    {
        if (str[i] == 'a' || str[i] == 'e' || str[i] == 'i' ||
            str[i] == 'o' || str[i] == 'u' || str[i] == 'y')
        {
            count++;
        }
        i++;
    }
    return count;
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        puts("[USAGE]: ./file <sentence>");
        return 1;
    }

    int result = count_vowels(argv[1]);

    printf("In \"%s\" there is %d vowels.\n", argv[1], result);

    return 0;
}
```

Tips

Do not worry if you do not understand all of this code yet.

You can now compile your file with `gcc`:

```
42sh$ gcc -std=c99 -Wall -Wextra -Werror -Wvla -pedantic file.c
42sh$ ls
file.c a.out [...]
```

By default, the *compiler*, once it has finished, outputs an executable file named `a.out`.

You can now run your program by typing "`./a.out`". You should have the following output:

```
42sh$ ./a.out "Hello World!"
In "Hello World!" there is 3 vowels.
```

1.2 Compiling in C

gcc has many different options, but during this semester you will mainly use these: `-std=c99`, `-pedantic`, `-Wextra`, `-Wall`, `-Wvla` and `-Werror`. If you want more information on *gcc* options, use `man 1 gcc`. Here is how you compile a program:

```
42sh$ gcc -Wall -Wextra -Werror -Wvla -pedantic -std=c99 file.c -o file
```

Be careful!

The use of compiler flags is really important. Unless explicitely mentionned, all your exercises and projects will be tested with the above flags: if you do not use them, you will miss errors and your project will not pass the tests.

Tips

The `-o` option allows you to specify an output file for compilation. The default output file is `a.out`.

After typing `gcc` and its options, specify the arguments. In this case the binary output file and then the *C* file. Once the binary file is generated, you can run your program:

```
42sh$ ./file "Hello World!"
In "Hello World!" there is 3 vowels.
42sh$
```

If your program is a valid C program, it will compile. However, this will not always be the case and you will make mistakes. The compiler can detect a lot of different errors and tell you about them.

For instance, this code:

```c
int main(void)
{
    char c = -6;
    int i = integer;
    c = i * "bla bla";
    return ();
}
```

Will generate the following errors:

```
42sh$ gcc -Wall -Wextra -Werror -pedantic -std=c99 -o test error.c
test.c: In function 'main':
test.c:4:11: error: 'integer' undeclared (first use in this function)
 int i = integer;
         ^~~~~~~
test.c:4:11: note: each undeclared identifier is reported only once for each function it␣
↪appears in
test.c:5:9: error: invalid operands to binary * (have 'int' and 'char *')
 c = i * "bla bla";
```

(continues on next page)

(continued from previous page)

```
       ^
test.c:6:11: error: expected expression before ')' token
 return ();
          ^
test.c:3:8: error: variable 'c' set but not used [-Werror=unused-but-set-variable]
 char c = -6;
```

It is normal not to understand everything in this paragraph, as you will learn step by step on your own. Our goal here is to show you that the compiler detects and explains all errors that block the generation of the binary file. As such, your compiler is your ally.

1.3 Comments in C

When you write code, you might want to add information that will not be taken into account by the compiler and only be read by programmers. These are called *comments* and they help people have a better understanding of the source code.

In *C*, there are two types of comments: single-line comments and multi-line comments.

Here are the comments syntaxes allowed by the EPITA standard:

```
// Single-line comment

/* Single-line comment in multi-line style */

/*
** Multi-line comment
*/
```

Commenting your code may not sound very useful at first, but it is a good practice to get into. It will help you understand your code when you come back to it after a while, and it will help other people understand your code if you share it with them. Commenting your code goes along with writing clean code.

In professional environments or in the open-source community, commenting is a must as it allows large teams to work for years on large projets. In your context, practicing commenting now will greatly help your for the many projects of the year.

Here is an example of the classical FizzBuzz program:

```
1  void fizzbuzz(unsigned int number)
2  {
3      for (unsigned int n = 1; n <= number; ++n)
4      {
5          if (n % 3 == 0)
6              printf("Fizz");
7          if (n % 5 == 0)
8              printf("Buzz");
9
10         if (n % 3 != 0 && n % 5 != 0)
11             printf("%u", n);
12
```

(continues on next page)

(continued from previous page)

```
        if (n < number)
            printf(", ");
    }
    putchar('\n');
}
```

It can be hard to understand what this function does and why it does it. Let us add some comments to make it more readable:

```
void fizzbuzz(unsigned int number)
{
    // For each integer in [1, number]
    for (unsigned int n = 1; n <= number; ++n)
    {
        if (n % 3 == 0)
            printf("Fizz");
        if (n % 5 == 0);
            printf("Buzz");

        // If n is a multiple of 15, the two above conditions print 'FizzBuzz'

        // If we do not print neither Fizz, Buzz nor FizzBuzz, print the number itself
        if (n % 3 != 0 && n % 5 != 0)
            printf("%u", n);

        // Add a separator between all numbers but not after the last one
        if (n < number)
            printf(", ");
    }

    putchar('\n');
}
```

Called with 16, the `fizzbuzz` function outputs:

```
1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz, 16
```

1.4 The main function

Computers need an entry point to start running a program. In *C*, the entry point is called `main`. Everything written in `main` will be executed by the computer.

A *C* `main` function looks like this:

```
int main(void)
{
    // C code
    return 0;
}
```

The first line is the `main` declaration. For now it will always be like this, but explanations about this line will be given later in this workshop.

You may also notice the braces `{` and `}`. Here, everything between braces belongs to `main` and will be executed. Do not mind the line '`return 0;`', it will be explained later.

If you look at the `main` function of `histogram.c`, you can see that the declaration is a bit different:

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

This allows us to pass arguments to our program. For now, just remember that there are multiple ways to declare the `main` function.

1.5 Variables

A variable associates an identifier (a name) to a value. As the name suggests, the value of the variable can change during the execution of the program. In *C*, each variable must have a unique name in its *Scopes*.

To **declare** a variable, we must specify its type and its name. Next, we can assign a value to the variable using the = sign. This part is called the **definition**.

```
int x; // declaration
x = 6; // assignation
int y = 23; // definition
x = x + 1;
int z = x + y;
```

In the above example, `x`, `y` and `z` are variables of type `int` (short for *integer*). The purpose of variable types will be described in another part.

1.5.1 Scopes

In *C*, a *scope* is enclosed between curly braces `{` and `}`. Variable names have to be unique inside a single scope. For instance, this is invalid code:

```
int a = 1; /* first definition of a */
int b = a + 2;
int a = b * 2; /* second definition of a, ERROR */
```

This can be fixed with scopes:

```
int a = 1; /* first definition of a */
int b = a + 2;
{
    int a = b * 2; /* overwrites the previous one */
}
```

As you can see, nested scopes have access to the variables declared in parent scopes but they can overwrite them.

1.5.2 Practice

List out all variables used in `histogram.c`.

1.6 Types

We just presented what a variable is, but we did not explain types. A type defines the kinds of values that a variable can contain. Once a variable is declared, its type cannot be changed and the variable cannot take other values than the ones allowed by its type. If we consider a variable as a box, the type can be seen as the shape of the box – only values with the same shape can fit within that box.

When defining a variable, we have to provide its type before its name. Initializing variables is not mandatory but is strongly recommended:

```
variable_type variable_name = value;
```

Tips

Note that if you do not assign a value to your variable, your variable will **not** take a default value, like 0. Therefore, the variable can have any value, and you should not use such a variable without giving it a value first.

The *C* language provides predefined types, such as:

- `char`: a character
- `int`: an integer
- `unsigned`: a positive integer, equivalent to `unsigned int`
- `float`: a floating-point number
- `double`: a floating-point number with a bigger precision

Going further...

In C floating-point literals such as `42.11` are actually of type `double`. To write a literal of type `float`, you must add the 'f' suffix: `41.11f`.

Be careful!

In C, there is no builtin type `string`. A string is a sequence of characters and is represented by an array of `char`. We will see later how it works. For now, we will only use `char` to represent a single character using single quotes. In Python, there were no difference when writing `'a'` or `"a"`, but in *C*, `'a'` is a char and `"a"` is a string.

Examples:

```
int my_integer = 20137;
char letter = 'T';
float pi = 3.1415f;
```

Going further...

In *C*, `char` values are represented by integers following the *ASCII table*. Open a terminal and type `man 7 ascii` to see the table.

1.6.1 Limits

In *C*, each type has a size which depends on the system and architecture. For example, an `int` is usually represented on 32 bits, which means that it can take values between -2^{31} (-2147483648) and $2^{31}-1$ (2147483647). If you increase the value of an `int` variable beyond its maximum value, it will *overflow* and take the minimum value. If you decrease the value of an `int` variable beyond its minimum value, it will *underflow* and take the maximum value.

When a variable *underflows* it will take the value of $2^{31}-1-offset$ at which it underflows. For example, $-2^{31}-2$ will result in $2^{31}-1-1$.

1.6.2 Practice

Here are a few literal values in *C*. What types do they belong to?

```
2025
'0'
41.5
41.5f
-32
'!'
```

1.7 Arithmetic operators

In *C*, there are five arithmetic operators: `+`, `-`, `*`, `/` and `%`. Their significations and priorities are the same as in math:

Operator	Description
+	Adds two operands.
-	Subtracts second operand from the first.
*	Multiplies both operands.
/	Divides numerator by denominator.
%	Modulus Operator and remainder of an euclidean division.

Example:

```c
int x = 1 + 1;
int y = x + 1 * 3;
int z = y % x;
```

In this example, `x` is equal to 2, `y` is equal to 5, and `z` is equal to 1.

Tips

You can combine any operator with "=" in order to do an operation and assign the value to the variable.

For example, these two lines do the exact same thing:

```
x = x + 3;
x += 3;
```

You can increment or decrement a variable by one using the operators ++ and --.

```
int d = 1;
int c = 2;
int k = 5;
int q = ++d + --c + k++;
```

Be careful!

Placing the operator before the variable (`++d`) or after the variable (`d++`) have different meanings. The latter is called *pre-incrementation* and the former *post-incrementation*.

```
int d = 1;
printf("%d\n", ++d);    // prints 2
printf("%d\n", d++);    // prints 2
printf("%d\n", d);      // prints 3
```

1.7.1 Practice

Now take a look at line 23 of `histogram.c`. Can you simplify it?

```
histo[0] = histo[0] + 1;
```

Tips

Note that lines 20, 22 and 24 of `histogram.c` show you equivalent notations to increment a variable.

What are the values of the following variables?

```
int a = 40 + 2;
int b = 40 + 2.0;
int c = 2.5;
int d = 1 / 2;
float e = 1 / 2;
float f = 1 / 2.0;
char g = 1 * '0';
unsigned h = -1;
int i = 4.2 % 3;
```

2 Control Flow

2.1 Conditional branching

Conditional branching is used to execute different blocks of code depending on a condition. The condition is a boolean expression, which can be evaluated to `true` or `false`.

Be careful!

In *C*, there is no builtin type boolean. `false` is represented by zero. All non-zero values represent `true`.

2.1.1 Relational operators

Relational operators check the relationship between two operands. They return `1` if the relation is true or `0` if it is false.

Be careful!

In *C*, **there is no builtin type boolean**. `false` is represented by zero. All non-zero values represent `true`.

Relational operators and their meanings in *C* are:

Operator	Description
==	equal to
!=	not equal to
<	lower than
<=	lower or equal to
>	greater than
>=	greater or equal to

Practice

Now that you know about variables and operators, try to guess the value of `a` at the end of this program:

```c
int a = 6;
int b = 4;
int c = a - b;
a = a - c * 2;
int d = a <= 1 + (b == 4);
a = c + d;
```

Tips

Note that the priority of arithmetic operators priority is higher than that of relational operators.

As such, `1 + 3 == 6 - 2` is equivalent to `(1 + 3) == (6 - 2)`.

2.1.2 Logical operators

Sometimes, you want to check multiple chained conditions. To do so, you can use two types of logical operators:

- **OR** operator `||`:

```
condition1 || condition2 || ... || conditionN
```

  The previous expression is evaluated to true (`1`) if at least one of the conditions is evaluated to `true` (`1`).

- **AND** operator `&&`:

```
condition1 && condition2 && ... && conditionN
```

  The previous expression is evaluated to `true` if all the conditions are evaluated to `true`.

The **NOT** operator, written `!`, is used to reverse the value of the following condition:

- if `condition` evaluates to `true`, `!condition` evaluates to `false`
- if `condition` evaluates to `false`, `!condition` evaluates to `true`

Practice

What would be the values of the following variables?

```c
int a = !1;
int b = 1 && 0;
int c = 1 && 1;
int d = 1 || 0;
int e = !(1 || 0);
int f = 1 && 1 && 0;
int g = 1 && (0 || 1);
int h = 1 && 1 || 0 && 1;
```

Tips

Remember that the priority of `&&` is higher than the priority of `||`.

Be careful!

If you have a doubt, you should use parenthesis to group the conditions you want to check.

2.1.3 If

Here is the syntax of an `if` block:

```
if (/* condition */)
{
    /* body */
}
```

The body is executed only if the condition evaluates to true. Optionally, you can add an `else` block which is executed if the condition evaluates to false:

```
if (condition)
{
    // executed if 'condition' is true
}
else
{
    // executed if 'condition' is false
}
```

If there is more than two cases, it is possible to check conditions one by one following the specified order by combining `else` and `if`:

```
if (condition1)
{
   // executed if condition1 is true
}
else if (condition2)
{
   // executed if condition1 if false and condition2 is true
}
else
{
   // executed if both condition1 and condition2 are false
}
```

Be careful!

When using `else if`, the evaluation stops at the first condition evaluated to `true`.

2.1.4 Practice

Write a program, using an `if`, that checks the values of a variable `t`. This variable can only take digits as value, and your program must follow these rules:

- if `t < 3`, print "`t < 3`"
- if `t` belongs to `[|3, 6|]`, print "`3 <= t <= 6`"
- else print "`t > 6`"

```c
int main(void)
{
    int t = 3;

    /* Add your code here */

    return 0;
}
```

Check your branching with different values of `t`.

2.2 Loops

Loops are a kind of control structure. Their aim is to execute the same piece of code multiple times.

All loops are composed of at least:

- a condition whose truth value may change over time
- a body that is executed as long as the condition is verified

There exist three types of loops: `while` loops, `do while` loops and `for` loops.

2.2.1 While loop

`while` loops let you execute commands *while* a certain condition evaluates to true. Variables used inside the condition must be declared before the loop.

```c
while (/* condition */)
{
    /* ... */
}
```

Example:

```c
int i = 0;
while (i < 42)
    i++;
```

Tips

Just like for `if` blocks, braces are mandatory only if there is more than one instruction in the loop.

Going further...

`break` and `continue` are two keywords that can be used inside a loop. If multiple loops are nested, only the one on the same level than the key word will be affected.

- `break`: stop the execution of the current loop
- `continue`: skip the current iteration of a loop and goes directly to the next one

Going further...

The `do-while` loop is similar to the while loop except the first iteration of the loop is always executed:

```
do
{
    /* ... */
} while (/* condition */);
```

2.2.2 Practice

Write a `while` loop that prints all digits from 9 to 0. The output should be:

```
42sh$ ./print_all_digits
9 8 7 6 5 4 3 2 1 0
42sh$
```

Be careful!

There should not be any space before the new line.

Write a `while` loop that computes the factorial of ten and then print it. The output should be:

```
3628800
```

Tips

To print an integer, you can use the `printf` function. You will see it more in depth later, but for now know that you can use `printf` this way:

```
#include <stdio.h> // Make sure to add this line at the beginning

int main(void)
{
    int i = 2;

    /*
    ** Note that the use of "%d" means "print an integer",
    ** and is followed by the name of the variable you want to print, here "i".
    */
    printf("%d\n", i);

    return 0;
}
```

```
42sh$ gcc -Wall -Wextra -Werror -pedantic -std=c99 file.c
42sh$ ./a.out
2
42sh$
```

2.2.3 For loop

`for` loops are similar to while loops except the syntax includes a way to initialize a variable and execute code after each iteration:

```
for (/* init */; /* condition */; /* expression */)
{
    /* body */
}
```

Consider the following example, which computes 2^{10}:

```
int result = 1;

for (int i = 0; i < 10; ++i)
    result *= 2
```

Here is an equivalent code using a while loop:

```
int result = 1;
int i = 1;

while (i < 10)
{
    result *= 2;
    ++i;
}
```

Going further...

Each of the three sections of the `for` loop are optional. Here is for instance a way to write an infinite loop:

```
for (;;)
{
    /* ... */
}
```

3 Functions

A function is a group of instructions that performs a task. Every function has:

- A prototype: the signature of the function, *i.e.* all information needed to call it.
- A body: The instructions that are executed when the function is called.

3.1 Functions declaration

3.1.1 Prototype

```
return_type function_name(type1 param1, type2 param2, ...);
```

A function *prototype* can be divided in three parts:

- A return type: specifies the type of the function's return value
- A name: it allows to uniquely identify the function and should give information about its behaviour
- An argument list: the list of what input the function needs in order to be called

Consider the following example:

```
int add(int a, int b);
```

- `int` is the return type of the function
- `add` is the name of the function
- `(int a, int b)` are the parameters of the function

Be careful!

If your function does not accept any arguments, you must put the keyword `void` in the prototype, like so: `int my_func(void)`.

3.1.2 Body

A function *body* is made up of the instructions executed when the function is called. All variables passed as arguments to the function are local to the function.

`return` statements allow you to stop the execution of the function and return a value to the caller.

Example:

```
int add(int a, int b)
{
    int result = a + b;
    return result;
}
```

Tips

Sometimes, we do not need to return anything. In that case, it is called a procedure. The return type must be `void` in that case.

Be careful!

The return statement stops the execution of the function.

Example:

```c
void print_hello(void)
{
    puts("Hello world!");
    return; // nothing is returned as the return type is 'void'
    puts("This line will never be printed.");
}
```

3.2 Function calls

Function calls are really useful as they allow you to reuse code that already exists without having to re-implement it each time you need it.

The syntax of a function call is the following:

```c
function_name(argument1, argument2, ...);
```

Consider the following example calling the function `add` defined above:

```c
int x = 3;
int y = 2;
int z = add(x, y); // 5
```

In this example, we call `add` with `3` and `2` as arguments. The variable `z` is thus declared as an integer, and initialized with the return value of the function: `5`.

Going further...

Arguments in *C* are passed by copy. This means the compiler creates another variable that contains the value of the argument. Any change made inside the function will not be propagated to the orignal variable.

3.3 Practice

Identify all the function calls in `histogram.c`.

3.3.1 sum_n

Write the function `sum_n` that returns the sum of the first *n* positive integers.

Here is its prototype:

```c
unsigned sum_n(unsigned n);
```

3.3.2 print_sum

Write the function `print_sum` which wraps the result of `sum_n` using the following rules:

- if n < 0, print "`Negative number!`".
- if n >= 0, print "`Result: <result>`"

Here is the expected prototype:

```c
void print_sum(int n);
```

These are the expected outputs:

```c
#include <stdio.h>

int main(void)
{
    print_sum(-25); // Negative number!

    print_sum(0); // Result: 0

    print_sum(6); // Result: 21

    return 0;
}
```

Tips

We will see `printf(3)` and `puts(3)` in a future section. If you are curious, you can check out `man 3 puts` and `man 3 printf`.

For instance, you can use `printf` to display an `unsigned` like this:

```c
unsigned answer = 42;
printf("%u\n", answer);
```

Tips

The line `#include <stdio.h>` is necessary to use `puts` and `printf`. It will be explained in the next section.

4 Includes and Headers

4.1 Includes

The `#include` directive is required when you need to include code that is declared elsewhere. This is typically useful to use functions that are not declared locally (in the same source file).

C's standard library provides a wide range of useful functions declared in files called system header files. Those are stored in specific directories on your computer.

`stdio.h` is such a file. Here is an example using it:

```c
#include <stdio.h>

int main(void)
{
    puts("Hello world!");
    return 0;
}
```

There is no function named `puts(3)` declared in this *C* file, yet it can still be called. This is thanks to the `#include` directive which makes the `puts(3)` function of `stdio.h` available.

Tips

See `man 3 puts`.

Be careful!

An include directive does not end with a `;` as it is not an instruction but a directive.

When you want to use a specifc function from the standard library, you can check its manpage with the `man` command to see which include is required.

For instance, on the `puts(3)` manpage, you can see under `SYNOPSIS` that `#include <stdio.h>` is written. That means that this include is needed in order to use the functions above.

4.1.1 Practice

Easy print

Open a file `tiny_print.c`. Write a `main` function, that prints "`She sells C shells.`" followed by a new line.

Tips

You must use `stdio.h`.

Duplicated but different

The following code requires you to enter a character which will then be printed.

```c
#include <stdio.h>

int main(void)
{
    puts("Please enter a character:");
    char c = getchar();
    puts("You pressed: ");
    putchar(c);
    putchar('\n');
    return 0;
}
```

`getchar(3)` is a function that reads the next character of the standard input (cf. `man 3 getchar`).

You must create 2 different versions of this code:

- One where you must print the pressed key twice
- One where you must ask for two characters and then print them

Be careful!

If you want to enter multiple characters using multiple calls to `getchar(3)`, you should press the `Return` key before entering all the needed characters. In fact, the newline, or `\n`, is a character and will therefore be understood as the next character read by `getchar(3)`.

4.2 Headers

Until now, you have seen how to write functions and use them within the same file. When writing your **C** projects, you will often want to share the implementation of some functions accross your source files.

Let's say we have `file.c` in which we have implemented `my_function`:

```c
/**
** \file file.c
*/

#include <stdio.h>

 void my_function(void)
 {
     puts("Do you wrestle with dreams? Do you contend with shadows?");
 }
```

And another file `main.c` in which we want to call `my_function`:

```c
/**
** \file main.c
*/

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

Let's try to compile those two files:

```
42sh$ gcc -Wall -Wextra -Werror -pedantic -std=c99 main.c file.c -o main
42sh$ main.c: In function ‘main’:
main.c:5:5: error: implicit declaration of function ‘my_function’ [-Werror=implicit-function-
↪declaration]
5 |     my_function();
  |     ^~~~~~~~~~~
```

gcc yields an error since it could not find a declaration of `my_function` within the file that tries to call it. To tell **gcc** that the declaration of the function is actually in another file, you need to specify where the compiler should look by using a **header file**. **Header files** have a **.h** extension.

To export `my_function`, which is implemented in `file.c`, you must create the file `file.h` and write `my_function`'s prototype like so:

```c
/**
** \file file.h
*/

#ifndef FILE_H
#define FILE_H

void my_function(void);

#endif /* ! FILE_H */
```

Be careful!

You should **never** include *.c* files. It is the source of many errors because contrary to header files which contain only declarations, *.c* files contain definitions. You will later see how multiple definitions result in errors. For now, remember that it is a huge mistake to include *.c* files.

Using the same `#include` directive you used with `stdio.h`, include `file.h` in the **.c** file in which you want to use the exported functions. Do so with the following syntax:

```c
/**
** \file main.c
*/

#include "file.h"

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

Tips

When including a system header file (such as `stdio.h`), we use angle brackets (< and >) around the header name. When including a local header we use double quotes (").

Now, when compiling `main.c`, **gcc** knows that `my_function` exists and is implemented in `file.c`!

```
42sh$ gcc -Wall -Wextra -Werror -pedantic -std=c99 main.c file.c -o main
42sh$ ls
file.c file.h main main.c
42sh$ ./main
Do you wrestle with dreams? Do you contend with shadows?
```

Going further...

`#include` is called a **preprocessor directive**. You will gain a better understanding of what the preprocessor is during the Piscine. For now, you can see include directives as copy-pastes. Writing `#include "file.h"` in `main.c` basically pastes the content of `file.h` in `main.c`.

Preprocessor directives form a separate language translated by a program called the **preprocessor**[1]. You can for example define **macros** with the directive `#define`:

```c
#include <stdio.h>

#define ANSWER 42

int main(void)
{
    printf("The answer is %d\n", ANSWER);
    return 0;
}
```

Here, `ANSWER` is not a variable but a macro. All occurrences of `ANSWER` in the file are effectively replaced by `42`.

Now, look at `file.h` again:

```c
#ifndef FILE_H
#define FILE_H

void my_function(void);

#endif /* ! FILE_H */
```

Using preprocessor macros, the highlighted lines correspond to **include guards**. They prevent the content of the file to be pasted multiple times if it is include multiple times.

[1] When you compile your code, **gcc** calls the preprocessor before doing anything else.

5 Printf

5.1 Printf introduction

You will often need to display the content of a variable to the user. The function `printf(3)` lets you do this easily. `printf` allows you to integrate the values of variables into a *format string* in order to display it. A *format string* describes the message you wish to display using special sequences of characters that serve as placeholders to be replaced by actual values.

```c
char letter = 'A';
int offset = 11;
printf("%c + %d = %c\n", letter, offset, letter + offset); // A + 11 = R
```

In this example, you can see that the format string contains three special sequences that start with '%'. Those are called **conversion sequences** and will be replaced by the values passed as arguments to `printf`. The character following that percent corresponds to the type of the replacement value. For instance, you can see that `c` formats the value as a character while `d` is used for integers. Many

conversion characters are supported by `printf` and they can be combined to further specify the type. The following table contains some of the conversion characters that you will mainly use:

Conversion character	Corresponding type
`d`	**int**
`u`	**unsigned**
`c`	**char** in ascii representation
`zu`	**size_t**
`f`	**double**

Tips

If you wish to display a % in your output, you must put %% in your format string.

Check `man 3 printf` for more details.

Be careful!

`printf(3)` does not automatically put a line feed at the end of the format string, you will need to add a **\n**.

Going further...

The prototype of `printf(3)` is the following:

```
printf(const char *restrict format, ...);
```

Notice the ellipsis (. . .) after the parameter `format`. This means that `printf` is a *variadic* function and accepts an arbitrary number of arguments. In practice, this depends on the format string.

5.1.1 Practice

Write a program that displays positive integers up to 100 using `printf(3)`.

5.1.2 Alphabet

On the assistants' intranet, do the exercise `Alphabet`.

6 Arrays

An array is a collection of elements **of the same type**. Each element is identified by an index specifying its position within the array.

6.1 Declaration

```
type var_name[N];
```

With `N` being the total number of items that can be stored in the array. `N` is called the dimension of the array and is always a positive integer. Moreover `N` **must** be known at compile time.

Be careful!

Once the size of an array is fixed, it can **not** be changed anymore. There is actually a solution to this problem but the concepts needed to understand will be introduced later on.

Going further...

Compile time refers to the period when the code is converted to machine code, whereas the run time is when the program is executed. Literals such as `10` or `'a'` are values known at compile time. These notions will be further explained later, for now you can look at the Compile Time page on Wikipedia

Example:

```
int tab[8];
```

Here, `tab` is an array that contains eight integers.

Tips

Elements in an array are designated by their indexes (which correspond to their position) in that array. In *C*, arrays are indexed from 0. This means that the indexes of their N elements go from 0 to N-1.

6.2 Initialization

An array can be initialized as any other variable.

```
int arr[5] =
{
    3, 42, 51, 90, 34
};
```

variable name:	arr				
elements:	**3**	**42**	**51**	**90**	**34**
index:	0	1	2	3	4

Table 1: int array of dimension 5

Or, by specifying only the first few elements of the array:

```
int arr[5] =
{
```

(continues on next page)

(continued from previous page)

```
    1, 2, 3
};
```

variable name:	arr				
elements:	**1**	**2**	**3**	**0**	**0**
index:	0	1	2	3	4

Table 2: int array of dimension 5 with only the 3 first elements initialized

The two non-specified elements are then initialized to 0. Thus, it is possible to initialize an array entirely to 0 this way:

```
int arr[24] =
{
    0
};
```

or this way:

```
int arr[24] = {};
```

However, the following syntax cause an *undefined behaviour* as we do not initialiaze the variable. We only declare it.

```
int arr[24];
```

Be careful!

The expression *undefined behaviour* means that the behavior for this action (in this case, initializing an array of length 24 without value) is not specified by the language, and therefore depends on the specific compiler implementation. This means that you do not know the values contained in the array. The execution may continue, potentially leaving your program in an erroneous state.

It is also possible to omit the dimension of an array if (and **only** if) you completely initialize it during its definition. The dimension will then be determined based on the number of values:

```
int arr[] =
{
    3, 42, 51, 90, 34
};
```

Here, `arr` is an int array of dimension 5.

6.3 Accessing elements

To access an element of an array, we use the bracket operator `[]`:

```
arr[index]
```

- where `index` can go from `0` to `N - 1` (`N` being the dimension of the array).
- `index` can, for example, be an arithmetic expression.

```
int arr[5] =
{
    1, 2, 3, 4, 5
};
int a = 0;

a = arr[2];          /* OK */
a = arr[3 + 1];      /* OK */
a = arr[5];      /* Undefined behaviour */
```

variable name:	arr					
elements:	**1**	**2**	**3**	**4**	**5**	**?**
index:	0	1	2	3	4	5

Table 3: int array of dimension 5 with access to index 5

The bracket operator `[]` is also used to assign a value in an array:

```
int arr[5] =
{
    1, 2, 3, 4, 5
};
```

```
arr[2] = 42;         /* {1, 2, 42, 4, 5} */
```

Going further...

When manipulating arrays, it is a good practice to store values like indexes or array length into `size_t` variables. The `size_t` type stores only positive integers and is defined in the `stddef.h` header.

6.3.1 Practice

Now that you know how to handle arrays in **C**, let's practice.

- What is the index of the first value of an array in C ?
- Is this initialization correct ? If it isn't, why ?

```
int arr[5] =
{
    0, 1, 2, 3, 4, 5
};
```

- What is the value of `a` in the following example ?

```
int arr[6] =
{
    1, 2, 3, 4, 5, 6
};

int a = arr[2];
```

- And in this one ?

```
int arr[6] =
{
    1, 2, 3, 4, 5, 6
};

int a = arr[6];
```

6.4 The main function

If arguments are to be passed to a 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 containing the arguments passed to the program.

Going further...

You can read `char *argv[]` as "`argv` is an array of `char *`". This is actually true and you will see later that `char *` corresponds to the string type. This means that `argv` is an array of strings.

Here is an example of passing arguments to a program:

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

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

- `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 that `argv[argc]` is always `NULL`. `NULL` is a special value we will see later on.

6.4.1 Practice: Max array

Goal

Write a function that returns the maximum value of an array of integers given as argument. If the array is empty you should return INT_MIN. The size of the array will always be correct.

```
int max_array(const int array[], size_t size);
```

6.4.2 Practice: Array vice max

Goal

Write a function that returns the vice-maximum (the *second* largest value) of an array of integers given as argument. Assume that the vector always contains at least two elements, that its size will always be correct and that all elements will have a different value.

Prototype:

```
int array_vice_max(const int vec[], size_t size);
```

7 Recursion

7.1 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**

7.2 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 arguments change from the last call until a **stopping condition** is met, it usually is a certain argument matching a specific value. 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`)

7.3 Example: print_sequence

Here is an example:

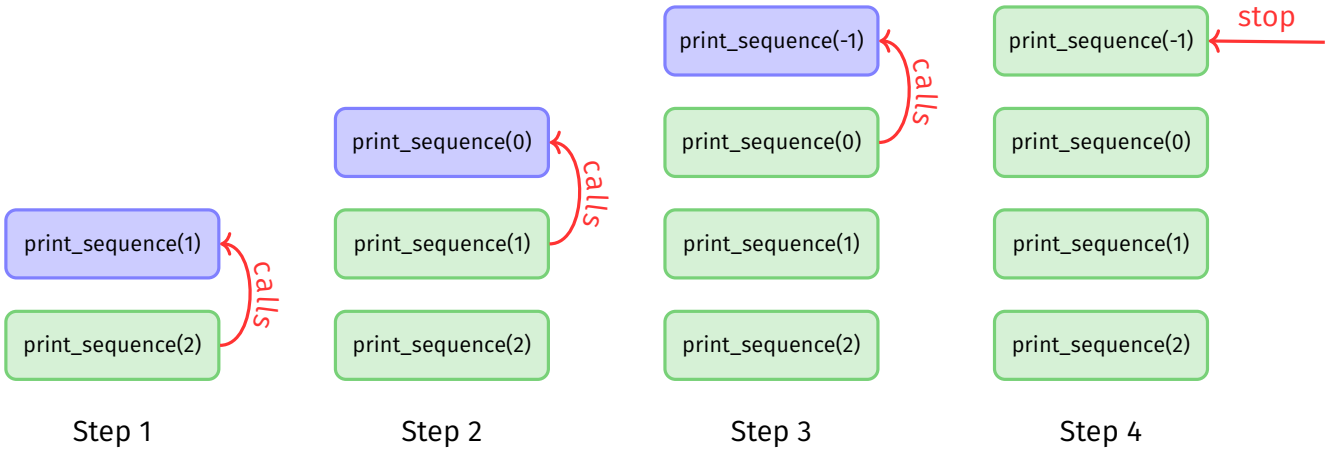
```c
#include <stdio.h>

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

    // 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
```

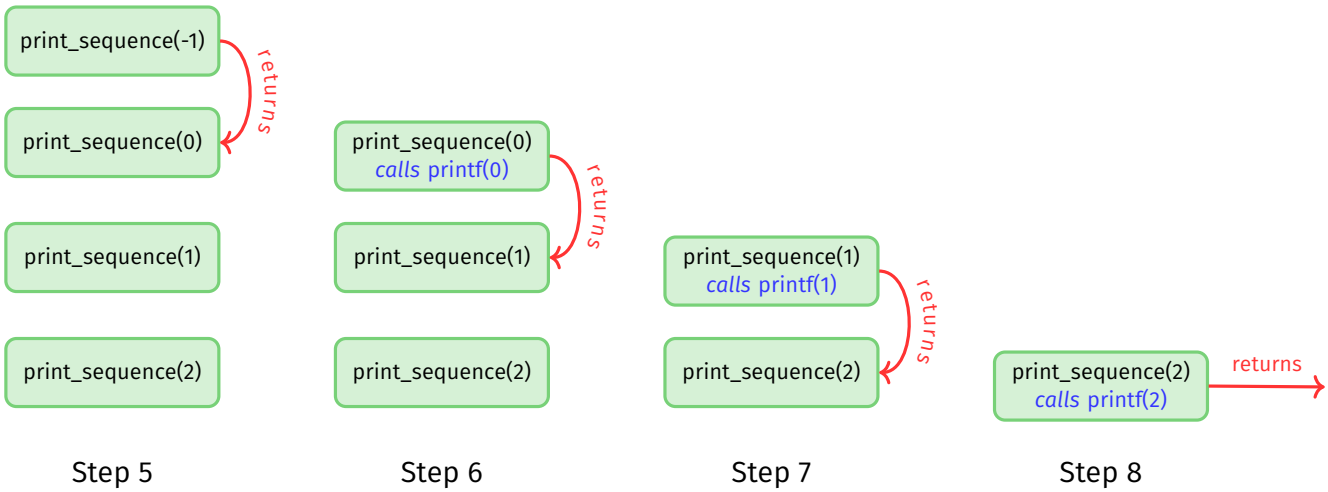
The function will print all integers from `0` to `x`.

Let us see what happens in detail 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.

7.4 Example: print_sequencev2

Let us continue the previous example:

```c
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 from `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)
{
```

(continues on next page)

(continued from previous page)

```c
    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:

```
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 are tail recursive functions different from normal recursive functions?

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 main 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.

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 a 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 handle, it will lead to a stack-overflow error. You don't have this problem with loops because they do 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 for 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);
}
```

7.5 Exercises

7.5.1 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:

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

```
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.

8 Pointer

8.1 Pointers

8.1.1 Introduction

Be careful!

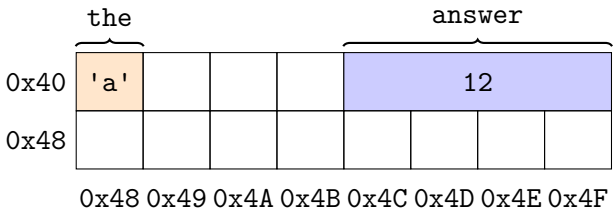
Pointers are a fundamental concept, pay extra attention and do not hesitate to ask questions! Any doubt you can have is valid, do not be afraid to ask what you might think is a "stupid question." There are no stupid questions.

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 the pointer type for a variable of type `int`.

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>;`.

```c
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 initialization 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`.

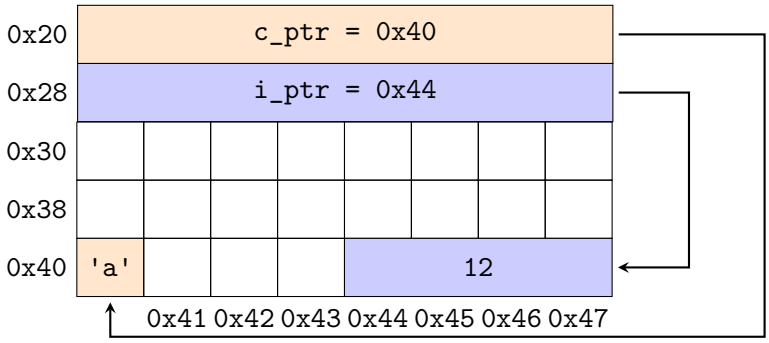
Like any other variable, you can assign a value to your pointer. To do so, you just need an address to assign to your variable with the correct associated type. We can use those variables to point to `the` and `answer` respectively.

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

```c
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 as an 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.

8.1.2 Initialization

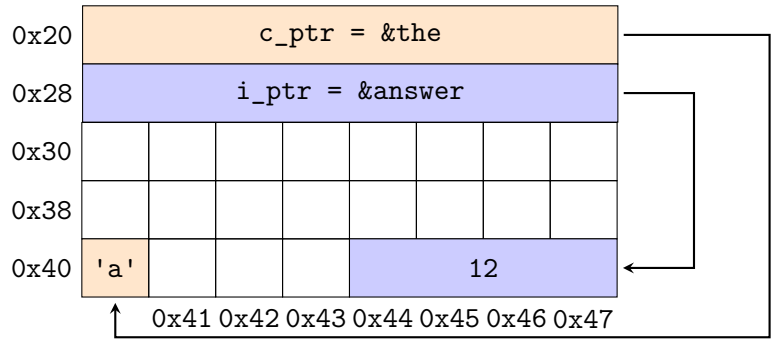
Where can we find an address to assign to a pointer? As you just saw, any variable in your program lives somewhere in your computer's memory. You can therefore use its address for your pointer.

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.

8.1.3 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;`

```
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 */
```

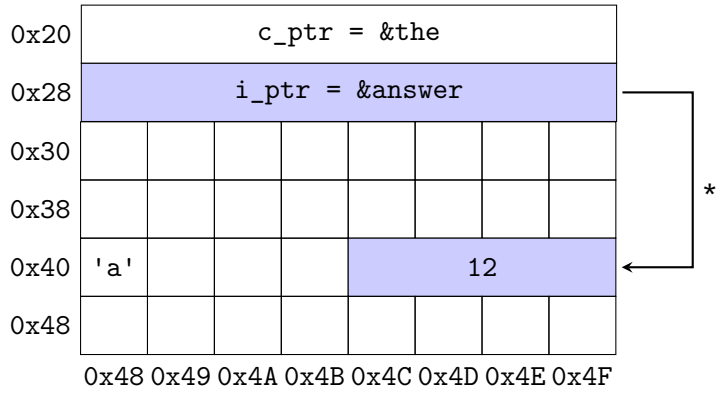


Fig. 1: `i_ptr` is dereferenced, accessing the value at address `0x44`

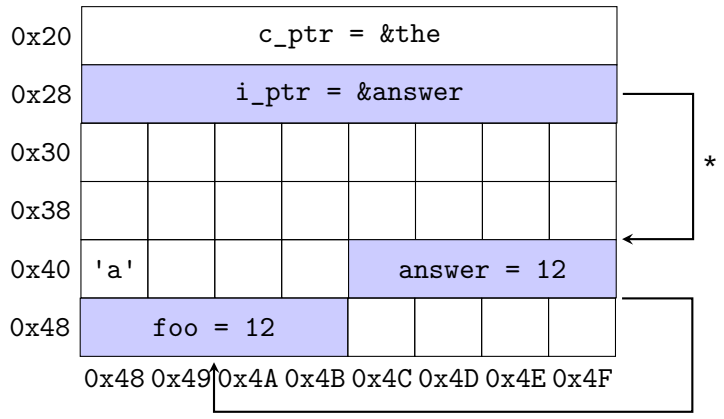


Fig. 2: The value at `0x44` is copied in `foo`, at `0x48`

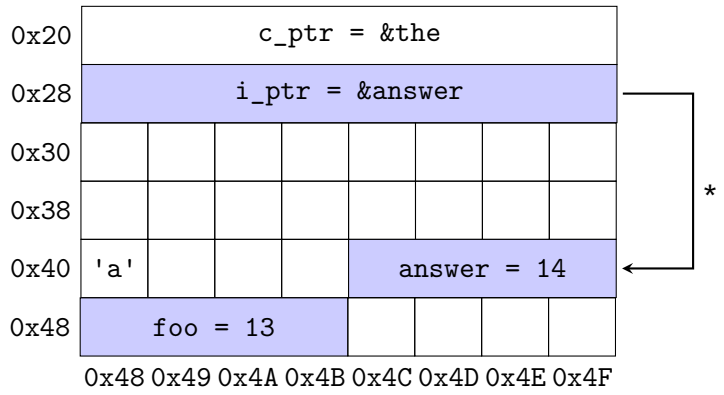


Fig. 3: `foo` and `*i_ptr` are different values

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);
```

8.1.4 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.

8.1.5 Passed by copy or by reference

In *C*, variables are passed **by copy**. This means that parameters will be **copied** for the function call, 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;
}
```

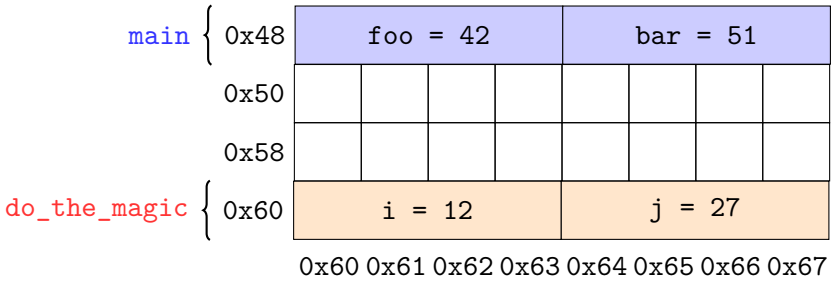


Fig. 4: Local copies are different variables in memory.

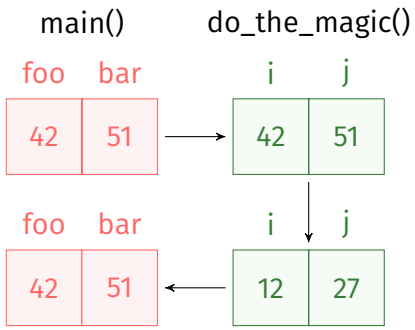


Fig. 5: Passing by copy to `do_the_magic()`

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 arguments 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;
}
```

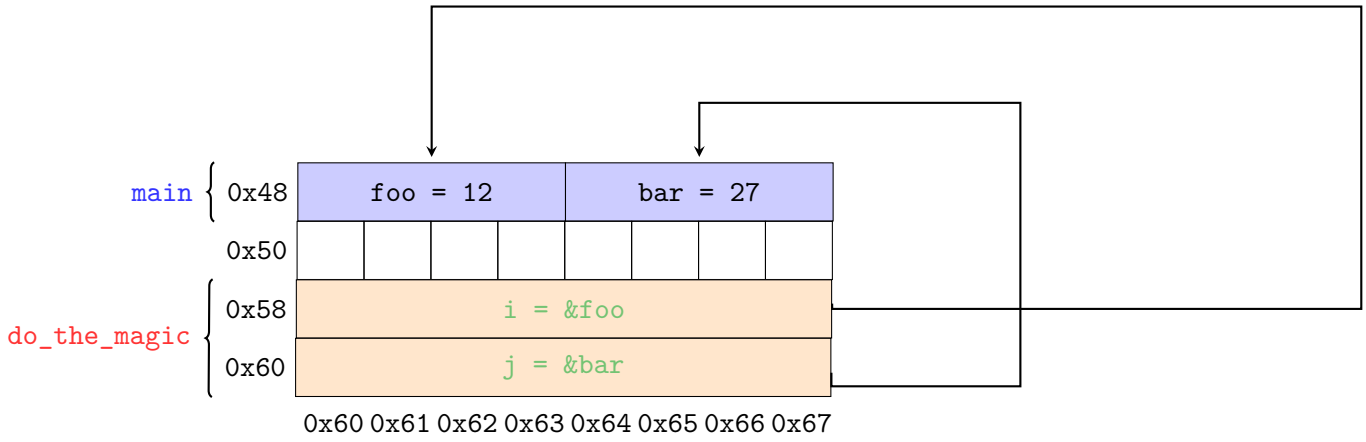


Fig. 6: Have pointers to `foo` and `bar` variables to modify their values in the `main` context.

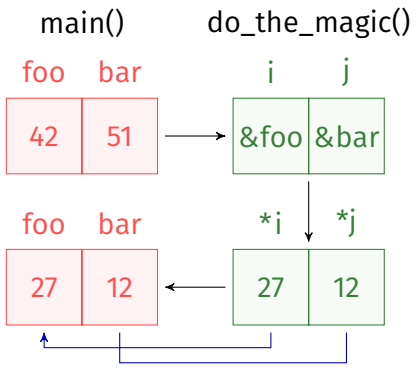


Fig. 7: Passing by reference to `do_the_magic()`

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**.

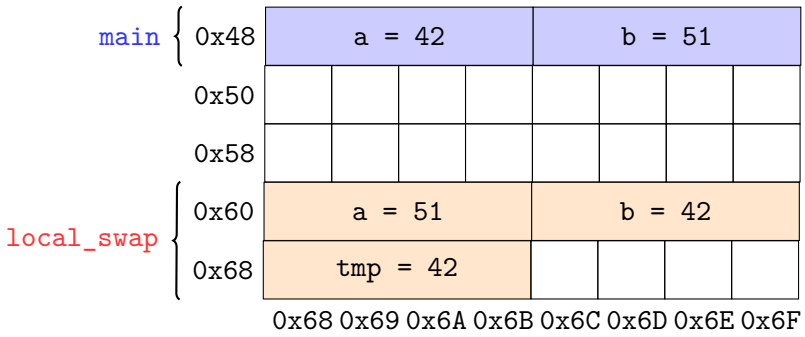


Fig. 8: Local copies are swapped.

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`.

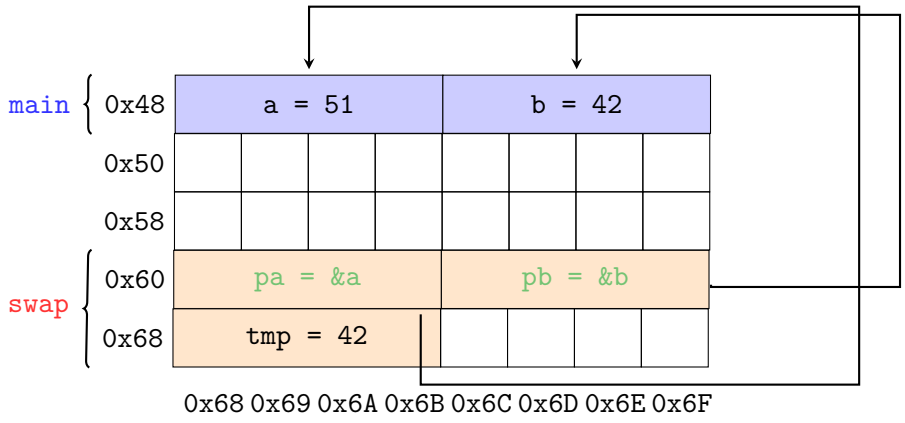


Fig. 9: Have pointers to `a` and `b` variables to swap their values in the `main` context.

8.1.6 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` pointer is constant and always evaluates to zero, *0x0* being the first address in your memory space. It corresponds to nothing and thus 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).

8.1.7 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
```

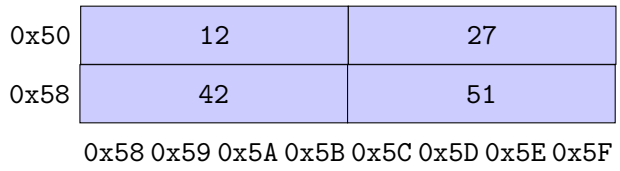


Fig. 10: The array is contiguous in memory, starting at 0x50

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.

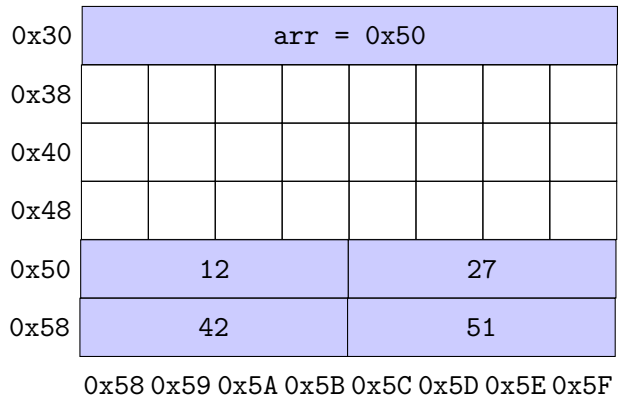


Fig. 11: `arr` contains the address of the first value of the array, `0x50`

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

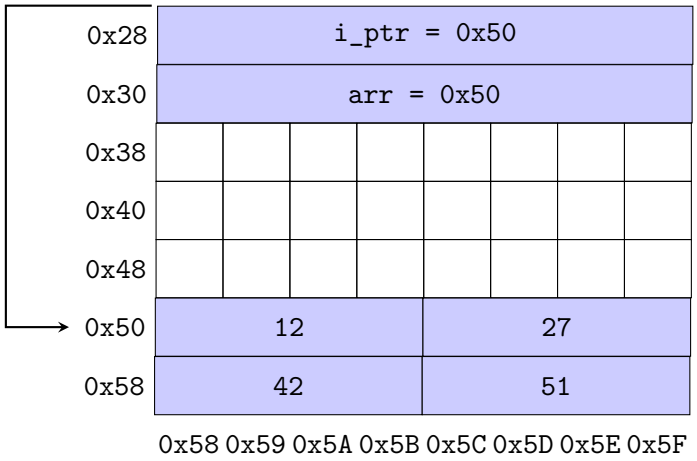


Fig. 12: `i_ptr` is now pointing to `0x50`

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 because 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 allow an easier manipulation of the memory by grouping their elements in one place, making them easy to index in relation to one-another.

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

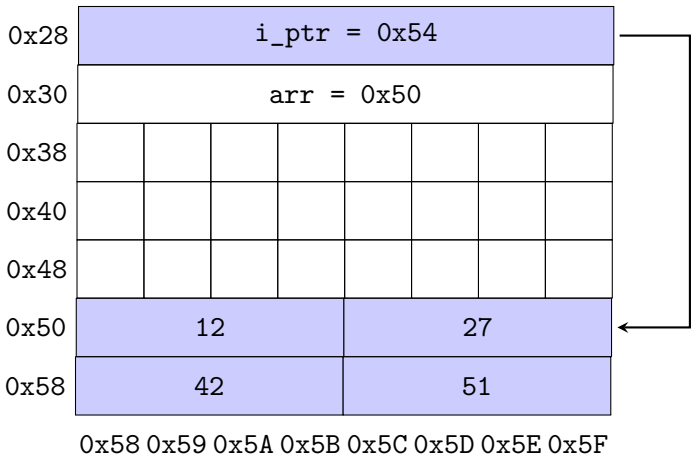


Fig. 13: `i_ptr` is now pointing to `0x54`

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 argument 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;
```

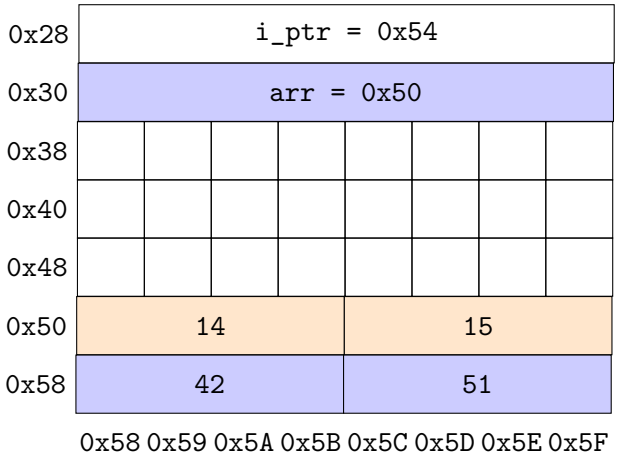


Fig. 14: Values at `0x50` and `0x54` are changed to `14` and `15` respectively

8.1.8 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 cannot 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 *`. Pointers 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.

9 Constness

The *C* language features the `const` keyword. This keyword is called a `type qualifier`. This is an additional information that we attach to a type and that is understood by the compiler. This keyword adds the notion of **constness** in the *C* language.

When declaring a variable with the keyword `const`, the variable's value cannot be modified. The only value it can hold is the value the variable is initialized to. We can compare `const` to a deal with the compiler. The compiler ensures us that the variable's value will not change during the execution of the program.

It is also a very useful information for readers to know if a variable is subject to change or not.

Let us take a look at some examples:

```
1 const int bar; /* valid but will cause an error with the required flags */
2 const int foo = 5; /* valid */
3 char const c = 'c'; /* valid, keywords can be placed anywhere before the name */
```

The first example is valid. However, when using the flag `-Wall` there will be one warning. Either that `bar` is defined but not used or `bar` is used uninitialized.

When your variables are not initialized, the compiler has the freedom to give them the value of its choice. Therefore, `bar` has a value that you do not know and would not be able to modify, which is perfectly useless.

Be careful!

Remember, always initialize your variables!

The second example is correct and serves its purpose, it is declared and initialized with a value that you specified.

The last example is correct syntaxically but not allowed by the coding style[1] at EPITA. You will need to respect this coding style during the piscine and all along the year. For now, just remember it exists.

[1] A coding style is a set of rules or guidelines used when writing the source code for a computer program. It enforces a standard for writing code, making it cleaner and more readable for any reader. At EPITA, the respect of the school's coding-style is enforced.

As said before, the constness is ensured by the compiler. Below is an example of the error message you will get if you try to modify a const variable.

```c
/**
** \file const.c
*/

int main(void)
{
    const int nb = 3;
    nb += 2;
}
```

```
42sh$ gcc const.c
const.c: In function ‘main’:
const.c:4:8: error: assignment of read-only variable ‘nb’
  nb += 2;
```

9.1 Constness and pointers

The `const` keyword can easily be used with pointers. The syntax is the following:

```c
const char * j = NULL;
int * const i = NULL;
```

You can now open `man 3 strlen`. The prototype of `strlen` is the following:

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

This function takes a `const` parameter. It means that this parameter cannot be modified in the function `strlen`. The purpose of `strlen(3)` is to return the length of a char array, it does not modify its content. Therefore, it is logical to specify to the compiler that it would not be modified.

To be more precise, the pointer `s` is not constant but the value it points to is. Indeed, `const char *` means that the `char` pointed by the pointer is constant. Hence, we can not modifiy the pointed value in any way. However, we can change `s`, the pointer itself. This means that we can assign a new pointer to `s`.

On the contrary, if we use the syntax `char * const`, the pointer is then constant but not the pointed value. We cannot change the pointer itself, but we can modify the value its holds.

A little reminder, in *C* all arguments are passed by copy. Therefore, any modification made to a parameter would not be persistent after the function call. This means that qualifying a parameter as `const` is kind of useless. However, that is not the case with pointers, the modification of the pointed value of a pointer is persistent after the function call. So, qualifying a pointer parameter as `const` has a real utility.

Be careful!

You need to be extra carefull with pointers. It can get quite confusing mixing `const`, pointers and function parameters. As a rule of thumb, you can remember the constness is associated to the nearest keyword leftwise.

9.1.1 Recap

Let us look at the following function:

```c
void weird_constness(const int *a, int * const b)
{
      *b = 1; /* works well because the pointed value of b is not const */
      b = NULL; /* generates a compilation error because b is const */

      *a = 1; /* generates a compiler error because the pointed value of a is const */
      a = NULL; /* works well because a is not const */
}
```

Try to play a little with this function to fully understand the difference.

10 Strings

10.1 ASCII

In *C* a variable of type `char` can take values from `-128` to `127`. Each value in the range [0, 127] corresponds to a character from the ASCII table.

ASCII is a standard to encode letters as numbers. Typing `man 7 ascii` in your terminal displays the ASCII table, in which you can see the number associated to each letter.

Going further...

The extended ASCII table associates letters and symbols to numbers in the range [0, 255]. Hence, following this table, each value of a `char` or `unsigned char` can be matched by a human readable character. Remember, `unsigned` specifies that the range of possible values for one type is any positive value in the range [0, nb_values - 1] but it does not define the total number of values available. Indeed, -125 will correspond to the value 131 for an `unsigned char` and *ƒ* in the extended ASCII table.

We strongly recommend you take a look at the ASCII table and notice a few things:

- The character '0' does not have the value 0.
- Characters are sorted logically, 'a' to 'z' are contiguous, as well as 'A' to 'Z' and '0' to '9'.

The value of a `char` variable being a number, all arithmetic operators can be used on `char`. Variables of type `char` take any value possible for one byte. Therefore, they can be used as if there were smaller `int` of 4 bytes (on the PIE).

```c
#include <stdio.h>

int main(void)
{
    char a = 'A';
    a += 32;

    if (a >= 97 && a <= 122)
```

(continues on next page)

(continued from previous page)

```
    {
        puts("'a' has become a lowercase character!");
    }

    return 0;
}
```

It is very impractical to use ASCII codes instead of chars. Here is what we will prefer:

```
#include <stdio.h>

int main(void)
{
    char a = 'A';
    a += 'a' - 'A';

    if (a >= 'a' && a <= 'z')
        puts("'a' has become a lowercase character!");

    return 0;
}
```

Examples:

```
char a = 'a';
char b = 'b';
char z = 'z';
int result = (b - a) * z;
```

At the end of these instructions, the resulting value is `(98 - 97) * 122 = 122`, representing the character 'z' in the ASCII table.

10.1.1 Practice

Now can you tell if the following assertions are true or false?

```
'a' > 'A'
'b' > 'a'
'0' == 0
```

What conditions should you use to check if a character is uppercase?

10.2 Strings

A string is a contiguous sequence of characters terminated by a null character in memory. If there are multiple null characters in a sequence of characters, the first null character is considered as the end of the string.

In *C*, in order to represent strings, we use a **character array ended by a '\0'**, a special character symbolizing the end of the string.

Going further...

If you check the value of *\0* in the ASCII table you can see it is 0.

There are many special characters, including:

`\n`	line break
`\t`	tabulation
`\0`	end of string character
`\\`	backslash (\)
`\"`	double quote
`\'`	single quote

Let's look at 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: the **string literals** (or *constant strings*).

The following example is semantically identical to the one above. Moreover the termination 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 is that they can be passed directly to functions taking `char[]` as arguments!

Be careful!

A `char` is single quoted wheras a string is double quoted. Therefore, 'c' is a `char` of value 'c', while "c" is a string of length 1 with 'c' as it first character and null terminated '0'.

```c
#include <stdio.h>

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

Finally, the length of a string is the number of characters before `\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.

Be careful when passing a string literal to a function like this:

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

In this case, the string `bar` cannot be modified. If the `foo` function tries to change it, the program will crash. If you need to modify a string literal, you must put it in a variable of type `char[]` before calling the function:

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

The following function declarations are equivalent:

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

10.3 Practice

10.3.1 Print Reverse

Write a function that prints the string given as argument in reverse order. You must follow this example:

```c
#include <stdio.h>

void reverse_print(char s[], int size)
{
```

(continues on next page)

(continued from previous page)

```
    // FIXME
}

int main(void)
{
    reverse_print("Hello World!", 12);
    puts(""); // To print a blank line at the end
}
```

You must obtain this

```
42sh$ gcc -Wall -Wextra -Werror -pedantic -std=c99 reverse.c -o reverse
42sh$ ./reverse
!dlroW olleH
```

Tips

Don't forget to check `man 3 putchar` for some help.

10.3.2 My Strlen

Goal

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

```
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.

10.3.3 My Strupcase

Goal

You must implement the following `strupcase` function :

```
void my_strupcase(char *str);
```

This function changes all lower case ASCII letters into upper case. `NULL` pointer will not be tested.

Example

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

int main(void)
{
  char str[] = "azerty1234XYZ &(";
  my_strupcase(str);
  printf("%s\n", str);
  return 0;
}
```

```
42sh$ ./my_strupcase | cat -e
AZERTY1234XYZ &($
```

11 Structures

11.1 Introduction

Structures are **user-defined data types** that allow us to combine data items of any type together. It is somewhat similar to an array, but structures can store data of different types.

Let's start with an example: when using an array, you may want to keep associated information, like the number of elements inside this array. To make this possible, you would declare an array and on the side, an integer which forces you to maintain the two variables separately. However, these variables are co-dependent. Ideally, we want to combine the array and the integer to make it easier to update accordingly. This is what structures are for.

Suppose we want to implement coordinates in a three dimensional euclidean plan. We need a structure with three fields of type `int`.

```c
struct coord
{
     int x;
     int y;
     int z;
};
```

`struct` is the *C* keyword associated with structures and used with both definitions and declarations. `coord` is the name of the defined structure meanwhile `x` is one of its **fields**.

11.2 Initialization

Structures can be used like any other types. Therefore, there are no differences with other types when dealing with declarations.

```
struct coord c;
```

`struct coord` defines the actual type of the variable and `c` is the variable's name. Initializing it this way will leave all its fields blank, thus we need to fill them. One way to define and put data in the fields would be:

```
struct coord c =
{
      .x = 1,
      .y = -2,
      .z = 10,
};
```

11.3 Accessing structure members

Since a structure is a group of data, we need to be able to access each field individually. We use '**.**' or '**->**', which are called **accessors**, to both read and write in our structures. We use **.** when the structure is not a pointer and **->** when it is.

Here is an example of writing data inside a structure, using both accessors:

```
struct coord c;

c.x = 10;
c.y = 5;
c.z = -2;

struct coord *c_ptr = &c; /* use the address of c */
c_ptr->x = 15;
c_ptr->y = -54;
c_ptr->z = 42;

// Set all other fields in the same way
```

Here is an example of reading data from a structure:

```
int x = c.x;
int y = c.y;
int z = c.z;
```

Tips

The syntax `ptr->field` for pointer is syntactic sugar[1] for `(*ptr).field`,

let's see an example:

```c
struct coord c = {.x=1, .y=3, .z=4};
struct coord *c_ptr = &c;

int a = c_ptr->x;
int b = (*c_ptr).x;

int d = a == b; // true
```

[1] Syntactic sugar is a syntax within a programming language that makes a construct easier to read or to express.

11.4 Function arguments

Since our structure is a type, it can of course be passed as a function argument. Let's say that we want to get the field `x` of our coord.

```c
int get_coord_x(const struct coord *coord)
{
    return coord->x;
}
```

Tips

As seen in the constness section, in a function where we do not modify the variable, it is good practice to use the `const` keyword so that the compiler knows it should never be modified in the function. Correctly used, it can ease the debugging of a program.

Be careful!

When passing a structure as an argument, it is most of the time better to pass it as a pointer. This is because the structure can be very large and copying it can be very expensive. Passing it as a pointer will only copy the address of the structure and not each of its fields, which is much faster and lighter.

11.5 Practice

- Create a structure `complex` representing a complex number which has two `int` fields: `re` and `im` representing the real and imaginary part of the complex number.
- Write a function `print_complex` printing the value held by a pointer to a complex number given as argument.
- Write a function `add_complex` taking two pointers of complex number as arguments and returning the sum of their value.
- Write a function `mul_complex` taking two pointers of complex number as arguments and returning the product of their value.

Here is an example of an output for a given `main`:

```c
#include <stdio.h>

int main(void)
{
    struct complex a = {.re = 3, .im = 5};
    struct complex b = {.re = -4, .im = 2};

    print_complex(&a);
    print_complex(&b);
    print_complex(add_complex(&a, &b));
    print_complex(mul_complex(&a, &b));

    return 0;
}
```

...will print:

```
3 + i * 5
-4 + i * 2
-1 + i * 7
-22 + i * -14
```

11.6 Headers

As you saw previously, to have a clear and great file organisation, the *C* language has header files (`.h`). You have seen that you can declare your function prototypes inside the header files.

Another usage of these files is to declare the structures inside too. This way, they will be available across the whole project.

By example:

```c
/**
** \file coord.h
*/

#ifndef COORD_H
#define COORD_H

struct coord
{
     int x;
     int y;
     int z;
};

#endif /* ! COORD_H */
```

11.7 Vectors

11.7.1 Explanation

You have just seen structures, now we will use them in combination with arrays. A **vector** is basically an array associated with its size.

Suppose we define a structure that contains an array of integers and the number of elements stored as an `int`.

This data structure used for integers will look like the one below:

```
struct int_vector
{
    int array[64];
    int size;
};
```

11.7.2 Practice

Vector min

Goal

Write a function that takes a vector of integers and returns the minimum value it contains. The size of the table will always be correct and superior to zero.

Prototype:

```
int int_vector_min(struct int_vector vec);
```

Vector max

Goal

Write a function that takes a vector of integers and returns the maximum value it contains. The size of the table will always be correct and superior to zero.

```
int int_vector_max(const struct int_vector vec);
```

Structure of the vector:

```
struct int_vector
{
    size_t size;
    int data[INT_VECTOR_LENGTH];
};
```

12 Dynamic Memory allocation

12.1 Use of memory allocation

You have already manipulated memory by declaring variables. In order for your program to run and access the values of your variables, the compiler has to know how much memory to allocate for each of them.

```c
int main(void)
{
    int a = 42;             /* 4 bytes or 32 bits */
    int b[3] = {1, 2, 3};   /* 3 * 4 bytes or 3 * 32 bits */

    return 0;
}
```

The memory is **automatically allocated** by the compiler during the compilation phase. The compiler knows how much memory to allocate for each variable and array because you told it so by declaring them. When the program exits, the memory is **automatically** freed or **deallocated**.

Sometimes you want to use some memory without knowing ahead of time how much you might need. In order to solve this problem, you need to manage the allocation of your memory by hand. The memory is **manually allocated** by you, during the execution of your program using the `malloc(3)` function. Therefore you need to **manually deallocate** this memory when you don't need it anymore. To do so you will use the `free(3)` function. Allocating memory manually during run-time is known as **dynamic memory allocation**.

For instance, say you want the user to enter an unknown number of integers. You can create a huge array to store them all, but if the user enters only one integer, a lot of memory is wasted. The right solution here is to use dynamic memory allocation to adapt the memory you use according to the space you need to store the user's input.

12.2 Dynamic memory

In *Python*, the management of the allocated memory space is **automatic**: dynamic allocations and deallocations are **implicitly** managed by the interpreter. Memory allocated by a call to `list()` is **automatically** deallocated when the list is not used anymore.

In *C*, the management of the allocated memory space is **manual**: dynamic allocations and deallocations are **explicitly** managed by you, the developer. Memory allocated by a call to `malloc(3)` is **not** automatically deallocated at the end of the function or at the end of the process. You have to call `free(3)` to deallocate it.

Be careful!

Every memory allocated by calling `malloc(3)` has to be freed using `free(3)`.

12.3 Memory allocation

The `malloc(3)` function allocates a chunk of memory of the specified size (in bytes) and returns a pointer to the beginning of this chunk. The `free(3)` function frees the memory previously allocated by a call to `malloc(3)`.

The following block of code is an example of using `malloc(3)` and `free(3)` declared in `stdlib.h`.

```c
/**
** \file my_tiny_int_array.c
*/

#include <stdlib.h>
#include <stdio.h>

int *create_my_int_array(size_t size)
{
    int *array = malloc(size * 4); /* number of elements times size of one element */

    if (NULL == array) /* It is mandatory to check the return value of malloc */
    {
        puts("Error(create_my_int_array): malloc returned NULL\n");
        return NULL;
    }

    for (size_t i = 0; i < size; i++)
    {
        array[i] = i;
    }

    return array;
}

int main(void)
{
    size_t size = 10;
    int *ptr = create_my_int_array(size);

    if (NULL == ptr)
    {
        puts("Error(main): malloc returned NULL\n");
        return 1;
    }

    for (size_t i = 0; i < size; i++)
    {
        printf("%d\n", ptr[i]);
    }

    free(ptr);
    return 0;
}
```

```
$ gcc -Wall -Wextra -Werror -pedantic -std=c99 my_tiny_int_array.c -o my_tiny_int_array
$ ./my_tiny_int_array
```

(continues on next page)

(continued from previous page)

```
0
1
2
3
4
5
6
7
8
9
```

Now let's see a more complex example:

```c
#include <stdlib.h>
#include <stdio.h>

char *strupcase_dup(char *s, size_t size)
{
    char *new_s = malloc((size + 1) * sizeof(char));

    if (NULL == new_s)
    {
        puts("Error: malloc returned NULL\n");
        return NULL;
    }

    for (size_t i = 0; s[i] != '\0'; i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z')
        {
            new_s[i] = s[i] - 'a' + 'A';
        }
        else
        {
            new_s[i] = s[i];
        }

    }
    new_s[size] = 0;
    return new_s;
}
```

As you can see, `malloc(3)` returns a pointer. Here the function `strupcase_dup` is returning a pointer to an area large enough to hold an array of `char` with the appropriate size. The `sizeof` keyword is used to determine the memory size required for the type `char`. We give this size to `malloc(3)` and the function will return a chunk of dedicated space in which you can store your `char` array.

Tips

The `sizeof` operator returns the size that a type occupies in the computer's memory. The type is passed as you would pass an argument to a function. For instance, on the PIE, `sizeof(int)` will be equal to four, since an `int` occupies 4 bytes in memory. Structures are types, thus `sizeof(struct vector)` will return the size of the whole structure, which depends on the combined size of all of

its fields.

We have seen two examples where `malloc(3)` is used to allocate memory for a `char` pointer and an `int` pointer. Let's have a look to the prototype of `malloc(3)`:

```
void *malloc(size_t size);
```

According to the prototype of the `malloc(3)` function, its return type is `void*`. This represents a generic pointer of an unknown data type. It is up to you to assign this pointer to a pointer of the specific type. Your compiler tries to ensure that you are correctly using your pointer. For example, if you specify a `char` pointer where an `int` pointer is expected, an error can be detected. This error would not have occurred had you been using a generic pointer.

Be careful!

You need to take special care when manipulating generic pointers. For instance, pointer arithmetic is not permitted by the C standard as the size of a *void* pointer is not defined.

`malloc(3)` will return `NULL` when it can't allocate memory, do not forget to check the return value of `malloc(3)`.

Be careful!

Always[1] check `malloc(3)`'s return value: if the allocation fails and returns `NULL`, your program will crash when it will use the pointer not correctly allocated (This may happen later in your code, making debugging needlessly harder).

[1] **Always**

We **strongly** advise you to always use `NULL` (defined in the header `stddef.h`, but included in `stdlib.h`) to initialize your pointers. Ideally, a pointer must contain either a valid address or `NULL`. Never leave an uninitialized pointer, because it can point to anything, and this address may not be a valid one, nor `NULL`.

12.4 Memory deallocation

As said previously, memory areas allocated by `malloc(3)` are not destroyed (freed or unallocated) automatically. We need a function to deallocate the memory's areas at the addresses returned by `malloc(3)`. This function is named `free(3)` and takes as parameter the pointer that must be released.

Whenever you don't need the memory allocated by `malloc(3)` anymore, you should free it using `free(3)`.

Forgetting to free causes *memory leaks*. Those are some of the worst mistakes that can occur in a program. If a program which leaks memory runs for a long period of time (for example a server), it will completely fill the RAM and will slow the system down, or even cause it to shutdown. *Memory leaks* are also some of the hardest bugs to find. You should **always**[1] keep in mind where you will free allocated memory.

Once you call `free(3)`, your pointer still holds the address, which is not valid anymore. Dereferencing this address leads to an undefined behavior. If your pointer variable still exists after you free it (not right before the function ends), you should assign it to `NULL` to avoid confusion.

[1] **Always**

For example:

```c
int *i_ptr = NULL;

// The memory space that i_ptr points to is allocated
i_ptr = malloc(sizeof(int));

if (!i_ptr) /* malloc returned NULL, this pointer is not valid */
{
    /* handle the error case */
}

*i_ptr = 42; /* let's fill this memory with a value */

int b = *i_ptr; /* b's value is 42 */

free(i_ptr); /* memory chunk is given back */

i_ptr = NULL; /* mark this pointer as NULL to avoid keeping an invalid address */
// b's value is still 42
// Do some stuff and return
```

Be careful not to mistake a pointer and the memory's area to which it points! In the previous example, the pointer variable (i.e. `i_ptr`) and the area pointed by `i_ptr` allocated manually with `malloc(3)` are not in the same place in memory.

The man page of function `free(3)` specifies that it takes as parameter any pointer returned by `malloc(3)`, thus giving `NULL` pointer to `free(3)` is valid (but won't do anything).

12.5 Exercises: memory

12.5.1 Create an array

You want to create arrays of `int`, but with a size that is only known at runtime.

```c
int *create_array(unsigned n);
```

Return a pointer to a memory region containing `n` integers. Write a message if you cannot allocate the memory.

12.5.2 Free an array

You do not need the previously allocated array anymore.

```c
void free_array(int *arr);
```

Free the memory used by the given array. Do not do anything if `arr` is `NULL`.

12.5.3 Custom array

Implement the function `array_create`:

```
struct my_array *array_create(size_t nb_elements);
```

This function allocates a new `my_array` of size `nb_elements`. Here is the given header for this exercise:

```
/**
** \file my_array.h
*/

#ifndef MY_ARRAY_H
#define MY_ARRAY_H

#include <stddef.h>

struct my_array
{
    int *data;
    size_t size;
};

struct my_array *array_create(size_t nb_elements);

#endif /* ! MY_ARRAY_H */
```

You will have to fix the function `create`, following this example:

```
/**
** \file my_array.c
*/

#include "my_array.h"

struct my_array *array_create(size_t nb_elements)
{
    // FIXME
}
```

Here is how it can be used:

```
/**
** \file main.c
*/

#include <stdio.h>
#include <stdlib.h>

#include "my_array.h"

int main(void)
{
    struct my_array *my_array = array_create(42);

```

(continues on next page)

(continued from previous page)

```
14      void *data = my_array->data;
15
16      if (NULL == my_array)
17      {
18          printf("No hero has been created!\n");
19      }
20      else if (NULL == data)
21      {
22          printf("No pointer data has been created!\n");
23      }
24      else
25      {
26          printf("my_array has a size of %zu and my_array->data is %p.\n",
27                  my_array->size, data);
28      }
29
30      free(my_array);
31  }
```

Tips

When executing `free(my_array)`, `my_array` can be `NULL`. This is not an issue since `free` does nothing when its argument is `NULL`. You can check `man 3 free`.

Be careful!

In order to display the address of a pointer, we need to convert it to a `void *`, hence the `data` variable. For more information, check the manual for `printf(3)`.

Note that you will rarely have to manipulate generic pointers like this and this is done juste for the sake of the example.

You have to allocate a `struct my_array`, set the field `size` using the function argument and allocate the field `data` using `malloc(3)`. If the structure cannot be allocated you have to return `NULL`.

Here is what you get when you compile and execute:

```
42sh$ gcc -Wall -Wextra -Werror -pedantic -std=c99 my_array.c main.c -o my_array
42sh$ ./my_array
my_array has a size of 42 and my_array->data (should be) different of NULL.
```

13 I/O

13.1 Introduction

I/O (Input / Ouput) is the communication between an information processing system, such as a computer, and the outside world, possibly a human or another information processing system. You are used to performing I/O operations in your day-to-day life: oral communication, using your keyboard (e.g. an I/O device that performs input operations to type stuff). In computer science, we find the same concept with slight differences. In UNIX-based operating systems, we perfom I/O between **files**

or **streams** to send or receive information. You have already performed some I/O operations in your *C* curriculum through the usage of `printf(3)` or `puts(3)`.

13.2 Streams

A stream is a sequence of data elements that are available over time. You can compare this to a river where the water particles can be compared to the data elements.

In the context of I/O, streams are sequences of characters. You can think about streaming services where you can watch a movie without having to download it entirely because you are watching while the service sends data to you. This is the same process.

In computing, streams are objects from which you can read or to which you can write. You can edit or process a file thanks to them.

Streams are an abstract concept above the notion of files that are used in UNIX based systems and in the *C*/*C++* programing languages. It allows an easier and simpler way to perform I/O operations, you will know more about files later.

13.2.1 Standard streams

You will use 3 streams usually, the 3 standard streams:

- **stdout**: standard output stream
- **stderr**: standard error stream
- **stdin**: standard input stream

These three streams are the main channel of interaction between a program and its users. You use them every time you run a command without even realising. By default, when you use `printf(3)` or `puts(3)`, you write on `stdout`, the standard output.

13.2.2 FILE structure

The `FILE` structure is used to represent I/O streams in *C*. It is defined in the header `stdio.h`. You can find some documentation here. You can only manipulate and access this type of objects through pointers: `FILE *`.

A lot of the functions you will see in this tutorial takes in parameter `FILE *`.

Keep in mind that you will use `FILE *` to manipulate streams. It is perfectly fine if you do not understand all the underlying concepts of the `FILE` structure.

13.2.3 Printf

Let us learn about `printf(3)`, you already know that it is used to print strings, but how does it work?

Simply put, `printf(3)` writes on the `stdout` stream, and your terminal displays this stream. This is why you can see what `printf(3)` prints.

There is an equivalent to `printf(3)` that allows you to choose the stream in which you will write: `fprintf(3)`. Let us take a look at its prototype:

```
int fprintf(FILE *restrict stream,
            const char *restrict format, ...);
```

Going further...

The `restrict` keyword gives information on how the pointer will be used. It is useful for the compiler to optimize the code. You can read this page to learn more about this keyword.

You can see that it takes a `FILE *` as parameter.

To write on `stderr` you could do:

```
#include <stdio.h>

fprintf(stderr, "Hello World!\n");
```

The three main streams (`FILE *`): `stdin`, `stdout` and `stderr` are declared in `stdio.h`. They can be used in any operation using `streams`.

13.3 Files

We have seen what are `streams` and viewed the three main ones. However, how do we **create** new ones and edit some files?

In UNIX-based systems, everything is a file. This includes, without being limited to, files, directories, hard drives, keyboards and even printers. In order to access them, you can use the set of functions `fopen(3)`, `fclose(3)`, `fwrite(3)`, and others.

13.3.1 Opening and Closing

`Opening` a file is the process of allocating ressources to *view* or *edit* your file. You can view this as opening a drawer before *viewing* its content or *taking* some things to *edit* its content. If we continue the analogy, `closing` a file is like closing your drawer. You deallocate every ressource needed to maintain a file open. As you do not want to let your drawer open, you **must** close each file you have opened.

```
$42sh ls
test open.c
```

```c
/**
** \file open.c
*/

#include <stdio.h>

int main(void)
{
     FILE *file = fopen("./test", "w"); /* Open a file in writing mode */

     if (NULL == file)
     {
         puts("Could not open test file\n");
         return 1;
     }

     fprintf(file, "Seek strength\n"); /* Write something in the stream file */

     fclose(file); /* Close the stream */

     return 0;
}
```

```
$42sh gcc -Wall -Wextra -pedantic -Wvla -Werror -std=c99 open.c -o open
$42sh ./open
$42sh cat -e test
Seek strength$
```

The previous code opened the file `test`, wrote one sentence in it and closed it. We will explain each line.

It starts by opening a file using `fopen(3)` which creates a stream (`FILE *`) corresponding to the file `test` in the current directory. Its prototype is the following:

```c
FILE *fopen(const char *restrict pathname, const char *restrict mode);
```

It takes two arguments: `pathname` and `mode`. The path can be either relative or absolute. The mode defines how the file is opened. The following table presents a non-exhaustive enumeration of legal values for `mode`:

`r`	reading mode
`r+`	reading and writing mode
`w`	writing mode
`w+`	reading and writing mode
`a`	appending in writing mode
`a+`	appending in reading and writing mode

Going further...

Those options can be used together. They can also be used with the `b` option that opens the file in binary mode. For more information about the possible values for `mode`, check the man page of `fopen(3)`.

In the above example, we opened the file `test` with the mode `w`. It truncates the file or creates it if it does not exist. It gives the stream write access on the file. Then, the file is used with `fprintf(3)` and closed using `fclose(3)`.

```
int fclose(FILE * stream);
```

Be careful!

When you open a file using `fopen(3)`, you must always close it using `fclose(3)`. When you allocate resources through your code you must ensure that it is correctly deallocated. In the case of opening a file, you need to close it.

13.3.2 Practice

Goal

Create a file with permissions 755.

13.3.3 Reading

After opening a file, you may want to read it. This can be achieved through multiple functions. Each one of them has its advantages and drawbacks. You are strongly advised to read their respective man pages.

```
int fgetc(FILE *stream);
char *fgets(char *restrict s, int size, FILE *restrict stream);
size_t fread(void *restrict ptr, size_t size, size_t nmemb,
             FILE *restrict stream);
```

- `fgetc(3)` gets a `char` from `stream` at its current position.
- `fgets(3)` reads a string `s` of max length `size` from `stream`.
- `fread(3)` is equivalent to `fgets(3)` but in binary mode. Therefore, the file needs to be opened using `fopen(3)` with at least the option `b`. You need to specify the number of elements to read `nmemb` and their `size` in bytes.

```
1  /**
2  ** \file fgets_example.c
3  */
4
5  #include <stdio.h>
6
7  int main(void)
8  {
9      FILE *file = fopen("test", "r");
10
11     if (NULL == file)
12     {
13         /* Handle the error case */
```

(continues on next page)

(continued from previous page)

```c
    }

    char buf[14];

    if (NULL == fgets(buf, 14, file)) /* Fill buf with the content of file */
                                      /* with at most 14 elements */
        printf("Error occured while reading\n");

    puts(buf);

    fclose(file);
}
```

```
$42sh gcc -Wall -Wextra -pedantic -Wvla -Werror -std=c99 fgets_example.c -o fgets_example
$42sh ./fgets_example
Seek strength
```

13.3.4 Writing

The following functions allow you to write text into your `FILE *` stream.

```c
int fputc(int c, FILE *stream);
int fputs(const char *restrict s, FILE *restrict stream);
size_t fwrite(const void *restrict ptr, size_t size, size_t nmemb,
              FILE *restrict stream);
```

- `fputc(3)` writes char `c` to `stream` at its current position.
- `fputs(3)` dumps `s` into `stream`.
- `fwrite(3)` is equivalent to `fputs(3)` but in binary mode. Hence, you need to specify the number of elements and their size. Do not forget that the file must be opened using at least the `b` option.

```c
/**
** \file fwrite_example.c
*/

#include <stdio.h>

int main(void)
{
    FILE *file = fopen("test", "ab+");

    if (NULL == file)
    {
        /* Handle the error case */
    }

    const char *buf = "The rest will follow\n";

    fwrite(buf, sizeof(char), 21,  file); /* Write at most 21 char from buf */
                                          /* to file */
```

(continues on next page)

(continued from previous page)

```
20
21      fclose(file);
22  }
```

```
$42sh gcc -Wall -Wextra -Wvla -pedantic -Werror -std=c99 fwrite_example.c -o fwrite_example
$42sh ./fwrite_example; cat -e test
Seek strength$
The rest will follow$
```

13.3.5 Browsing through a file

For now, you have only learned to open and close a file. However, how can you write or read something at a specific place in the file? In the `FILE` structure, there is a field named `fpos` that corresponds to the actual location of the stream. When opening a file with `r` option, `fpos` is at the beginning of the file, whereas for `a` option, it is at the end.

You can also move the current position of your stream arbitraly using `fseek(3)`. You are advised to check the man page. Its prototype is:

```
int fseek(FILE *stream, long offset, int whence);
```

`whence` can be either `SEEK_SET`, `SEEK_END`, `SEEK_CUR`, which respectively correspond to the beginning, the end or the current position of the file.

```
1   /**
2   ** \file offset.c
3   */
4
5   #include <stdio.h>
6
7   int main(void)
8   {
9       FILE* file = fopen("test", "r");
10
11      if (fseek(file, 5, SEEK_SET) == -1) /* Move the current position */
12      {
13          puts("Error on fseek\n");
14      }
15
16      char buf[9];
17
18      if (NULL == fgets(buf, 9, file)) /* Fill buf with the content of file */
19      {
20          puts("Could not get content of file\n");
21      }
22
23      puts(buf);
24
25      fclose(file);
26  }
```

```
$42sh gcc -Wall -Wextra -Wvla -pedantic -Werror -std=c99 offset.c -o offset
$42sh ./offset
strength
```

13.3.6 Practice

Goal

Write a function that returns the number of lines in file `file_in`. If any error occurs, returns `-1` (for instance, if the file does not exist). Please note that an empty file is considered as having one line, and the `\n` character does not create a new line if it is at the end of the file.

You must open the file with read-only permission.

```
int count_lines(const char *file_in);
```

14 Makefile

14.1 Context

For now, in order to compile your program you have been using something like:

```
42sh$ gcc -std=c99 -Wall -Wextra -Werror -pedantic hello_world.c -o hello_world
```

We can all agree that this is inconvenient and fastidious to type[1].

When working on actual projects, you will also have more than one source file, and quickly, your simple one line command can become very long and if you want to move a file it will be complex to edit[2].

Yet another problem that you will experience soon enough: compiling a big project can take a lot of time and be boring. You do not want to recompile the whole project each time you edit a single file.

To summarize, we need something that can solve these problems:

- We do not want to type a long command.
- We want to be able to edit our compilation command easily.
- We do not want to recompile our project each time we edit one single file.
- We want other people to be able to easily compile our project.

[1] Do not bring up your `.bash_history` to us, it is equally fastidious to search the command line you want!
[2] `fc(1)` is out of the discussion as well.

14.2 What is make?

In response to those issues, the POSIX standard contains a tool used to handle project builds: `make`. The implementation of `make` that we will use this semester is GNU make which was released by the FSF as part of the GNU project. `GNU make` implements all the features defined in the POSIX standard for `make` and has some nice extensions as well.

`make` is a tool used to simplify the task of building programs composed of many source files. It can be configured through a file named `Makefile`, `makefile`, or `GNUmakefile`. When ran, `make` will first search for a `GNUmakefile`, then `makefile`, and finally `Makefile` if it did not find the previous ones. However, it is recommended in the official GNU/Make documentation to call your makefile `Makefile`, and we expect you to follow this convention for your submissions.

`make` parses rules and variables from the `Makefile` which are mainly used to build the project. `make` will only re-build things that need to be re-built by comparing modification dates of targets with their dependencies.

A `Makefile` typically starts with a few variable definitions, followed by a set of target entries. Each variable from the `Makefile` is used with `$(VARIABLE)` and can be declared at the top of the file as:

```
VARIABLE = VALUE
```

Finally, each rule abides by the following syntax:

```
target: dependency1 dependency2 ...
    command
    ...
```

A target is usually the name of the file generated by the Makefile rule. The most common examples of targets are executables or object files. It can also be the name of an action to carry out, for example `all`, `clean`, ...

A dependency (or "prerequisite") is a file that needs to exist in order to create the target. A target can also be a dependency for another target. When evaluating a rule, `make` will analyze its dependencies and if one of them is a target of another rule, `make` will evaluate it too before the one it depends on.

A command (or "recipe") is an action that `make` carries out. Be careful, each command is preceded by a **tabulation** (`\t`). Otherwise, your `Makefile` will not work.

14.3 Invocation

14.3.1 Basics

Let us start easy. First, you need a `hello_world.c` like the following one:

```c
#include <stdio.h>

int main(void)
{
    puts("Hello World !");
    return 0;
}
```

And a `Makefile`:

```
CC=gcc
CFLAGS=-std=c99 -Wall -Wextra -Wvla -Werror -pedantic

exec: hello_world
	./hello_world

hello_world: hello_world.c
	$(CC) $(CFLAGS) -o hello_world hello_world.c
```

Be careful!

Remember that the `command` part of a `rule` **must** start with a tabulation. If it does not, `make` will raise the following error:

```
Makefile:5: *** missing separator.  Stop.
```

Now try:

```
42sh$ make exec
gcc -std=c99 -Wall -Wextra -Wvla -Werror -pedantic -o hello_world hello_world.c
./hello_world
```

So what is going on here?

First you have two `variables`:

- `CC`, for the *C* Compiler
- `CFLAGS`, for the *C* Flags

Going further...

Some variables are predefined by `make`, like `CC` and `CFLAGS` for example. They are generally used for the *C* compiler and *C* flags respectively.

Then you have two `rules`. Those are the core of a `Makefile`. Let us look at the first one.

First we have the target `exec` that will execute a binary called `hello_world` using `bash`.

In order to execute our binary, we need it. We say that our target `exec` depends on the existence of the binary `hello_world`. Thus, `hello_world` is a **dependency** of the target `exec`.

But the binary `hello_world` does not exist! This is why we have the second rule! If a dependency does not exist, `make` will search for a rule that can produce it and execute it before the first.

The second rule follows the same principle. This time the target is `hello_world` and that is the file produced by this rule. This rule has as dependency `hello_world.c`.

Now that our `rule` specifies the target it produces and what it depends on, we need to specify how it should be produced. Remember our `variables`? We will need them now. In order to expand[3] a Makefile `variable`, you need to put it between parentheses[4] with a dollar before it.

[3] `expansion` is the concept of replacing a variable by its value before interpreting the line. Remember your shell courses. When you needed to access a variable's value you also used this syntax.

[4] You might see braces instead of parentheses, they work the same. It is as you prefer.

So if we summarize this rule, it will produce the target `hello_world` that depends on the file `hello_world.c` using the following command line:

```
gcc -o hello_world hello_world.c
```

Going further...

The default `rule` called when typing `make` is the first one in the file. Thus, here, typing `make` or `make exec` will have exactly the same behavior.

If you run `make hello_world` without updating any file, and after running `make` or `make hello_world` once, you will have a message like this:

```
42sh$ make hello_world
make: 'hello_world' is up to date.
```

If a target already exists, `make` will rebuild it if its dependencies do not exist or if they have been updated since the last build.

Going further...

You can force `make` to rebuild all targets with the option `-B`.

If you modify the `hello_world.c` and retry `make hello_world`, it will rebuild everything.

Another interesting feature of `make` is the `-n` option, which asks `make` to print the commands it would run without actually executing them. This is known as a **dry run**.

```
42sh$ cat Makefile
all:
    echo toto
42sh$ make
echo toto
toto
42sh$ make -n
echo toto
```

Tips

Generally the first rule in a Makefile is named `all`. Note that this is only a convention and you are free to not respect it.

15 Bitwise Operations

In *C* and many other languages, you can operate on bits. There are six bitwise operators:

Operator	Description
&	Performs bitwise AND on each bit of two operands.
|	Performs bitwise OR on each bit of two operands.
^	Performs bitwise XOR on each bit of two operands.
~	Performs bitwise NOT, inverting each bit of the operand.
<<	Left shift operator. Shifts bits of the first operand left by the number of positions specified by the second operand.
>>	Right shift operator. Shifts bits of the first operand right by the number of positions specified by the second operand.

Do not worry if you do not yet understand these operators. Each one of them will get its own section. Go slow and read carefully, these operators can be a little hard to properly understand. Do not hesitate to ask questions!

15.1 AND operator

The *AND* (`&`) operator compares each bit of the first operand to the corresponding bit of the second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the result bit is set to 0.

Let us look at what is called a truth table. The `A` and `B` columns represent all possible combinations of values for these two bits. The `A & B` column shows the result of applying the `&` operator to the corresponding values of `A` and `B`.

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

```
unsigned char a = 1;
unsigned char b = 0;
// this will be 0
unsigned char a_and_b = a | b;
```

15.2 OR operator

The *OR* (`|`) operator compares each bit of the first operand to the corresponding bit of the second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the result bit is set to 0.

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

```
unsigned char a = 1;
unsigned char b = 0;
// this will be 1
unsigned char a_or_b = a | b;
```

15.3 XOR operator

The *XOR* (^) operator compares each bit of the first operand to the corresponding bit of the second operand. If the bits are different, the corresponding result bit is set to 1. Otherwise, the result bit is set to 0.

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

```
unsigned char a = 1;
unsigned char b = 0;
// this will be 1
unsigned char a_xor_b = a ^ b;
// this will be 0
unsigned char a_xor_1 = a ^ 1;
```

15.4 NOT operator

The *NOT* (~) operator is the only **unary** operator on this list, meaning this operator only needs one operand.

A	~A
0	1
1	0

```
unsigned char a = 1;
// this will be 0
unsigned char not_a = ~a;
```

15.5 SHIFT LEFT operator

Shifts all bits of the first operand to the left by the number of positions specified by the second operand. Bits shifted out of the left side are discarded, and 0s are shifted into the right side. This operation effectively multiplies the number by 2 for each shift position.

```
unsigned char a = 1;
// this will be 2
// 00000001 << 1 = 000000010
unsigned char b = a << 1;
// this will be 4
// 00000001 << 2 = 000000100
unsigned char c = a << 2;
```

15.6 SHIFT RIGHT operator

Shifts all bits of the first operand to the right by the number of positions specified by the second operand. Bits shifted out of the right side are discarded, and 0s are shifted into the left side. This operation effectively divides the number by 2 for each shift position.

```
unsigned char a = 4;
// this will be 2
// 00000100 >> 1 = 000000010
unsigned char b = a >> 1;
// this will be 1
// 00000100 >> 2 = 000000001
unsigned char c = a >> 2;
```

Seek strength. The rest will follow.