Piscine — Tutorial D1 AM

version **#b4173ffc5996c36bf7f9730185ebc8357500bd86**

Assistants C/Unix 2024 <assistants@tickets.assistants.epita.fr>

Copyright

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

Contents

1 Introduction to *C* 4
- 1.1 History . . . 4
- 1.2 Syntax reminders . . . 4
- 1.3 Variables and data types . . . 5
- 1.4 Operators . . . 7
- 1.5 ASCII . . . 11
- 1.6 Control structures . . . 12
- 1.7 Functions . . . 15
- 1.8 The `main` function . . . 18
- 1.9 Writing on the terminal . . . 18

2 Compilation 19
- 2.1 Some compiler options . . . 19

3 Coding Style 20

4 Clang Format 20

5 Writing on standard output 20
- 5.1 `printf(3)` . . . 20
- 5.2 Escape sequence . . . 22
- 5.3 Format output . . . 22
- 5.4 Exercise . . . 23

6 Arrays 23
- 6.1 One-dimensional arrays . . . 23
- 6.2 Size type . . . 25
- 6.3 Determining size . . . 25
- 6.4 Exercises . . . 26

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

7 Multi-dimensional arrays **27**
7.1 Declaration 27
7.2 Initialization during declaration 27
7.3 Accessing values 27
7.4 Matrix with a one-dimensional array 28

8 Preprocessor directives **29**
8.1 Macros using `#define` 29
8.2 Working with multiple files using `#include` 30

1 Introduction to *C*

1.1 History

The *C* language is linked to the design of the UNIX system by Bell labs in the 1970s. Its development was influenced by two languages:

- *BCPL*, developed in 1966 by Martin Richards
- *B*, developed in 1970 at Bell labs

The first version of the *C* compiler[1] was written in 1972 by Dennis Ritchie. From there, the language grew in popularity along the UNIX systems and numerous versions of the *C* language were created:

- In 1978: *K&R C*, an informal specification based on the book *The C Programming Language* written by Brian Kernighan and Dennis Ritchie.
- In 1989: *ANSI C* or *C89*, the first official *C* standard.
- In 1990: *ISO C* or *C90*, the same language as *C89* but published as an ISO standard.
- In 1999: *C99*, major extensions to the standard *C* language.
- In 2011: *C11*, minor new language features.
- In 2017: *C17*, minor corrections of *C11* without adding new features.

At EPITA, we will be using the *C99* standard.

The *C* programming language has been constantly ranked among the most popular programming languages since the 1980s according to the TIOBE index. Because of its dense history and low-level design, *C* is best known to be very portable, extremely efficient and a mature language. In the industry, it is widely used for: operating systems and kernel development, compilers and interpreters design, libraries, embedded systems, database management systems, ...

1.2 Syntax reminders

1.2.1 Comments

In *C* there are two types of comments: single-line comments and multi-line comments. See the examples for the syntax of each type of comment.

Examples:

```
// I am a single-line comment

/* I am a single-line comment in the multi-line style */

/*
** I am a multi-line comment authorized by the EPITA standard
*/
```

(continues on next page)

[1] The tool used to translate source code into an executable program. You will study compiler details and inner-workings at length during your ING1.

(continued from previous page)

```
/*
   I am also a multi-line comment, but not authorized by the coding style
*/
```

1.3 Variables and data types

1.3.1 Variables

A variable consists of:

- A type which is one of built in *C* types or a user defined.
- They have an identifier (a name) that must respect the following naming conventions:
  - Start with a letter or an underscore (`'_'`).
  - Consist of a sequence of letters, numbers or underscores.
  - Be different from *C* keywords.

Be careful!

Starting with `'_'` is forbidden by the coding style.

- Possibly a value.

```
int     i;
int     j = 3;
char    c = 'a';
float   f = 42.42;
```

You can then use declared variables in the program by using their identifiers.

```
int a = 1;
int b = 41;

int sum = a + b; // sum == 42
```

1.3.2 Predefined types

Basic data types of *C*

- `void`: a variable cannot have this type, which means “having no type”, this type is used for procedures (see below).
- `char`: a character (which is actually a number) coded with a single byte.
- `int`: an integer which memory space depends on the architecture of the machine (2 bytes on 16-bit architectures, 4 on 32 and 64-bit architectures).
- `float`: a floating point number with simple precision (4 bytes).

- `double`: a floating point number with double precision (8 bytes).

It is possible to apply a number of qualifiers to these data types, the followings apply to integers:

Name	Bytes	Possible values (-2^{n-1} to $2^{n-1}-1$)
`short` (`int`)	2	-32 768 to 32 767
`int`	2 **or** 4	-2^{15} to $2^{15}-1$ **or** -2^{31} to $2^{31}-1$
`long` (`int`)	4 **or** 8	-2^{31} to $2^{31}-1$ **or** -2^{63} to $2^{63}-1$
`long long` (`int`)	8	-2^{63} to $2^{63}-1$

Note that the long qualifier depends on your architecture: on 32-bit architectures, it will be 4 bytes long, and on 64-bit architectures, it will be 8 bytes long.

Tips

A bit can have two values, 0 or 1. A byte is 8 bits long, thus having values from 0 to 255 (11111111 in binary).

For example:

```
short int shortvar;
long int counter;
```

In that case, `int` is optional.

By default, data types are *signed*, which means that variables with these types can take negative or positive values. It is also possible to use unsigned types thanks to the keyword `unsigned` (and you specify that it is signed with the `signed` keyword, but integers are signed by default, so this keyword is rarely used).

Be careful!

- `signed` and `unsigned` qualifiers only apply to integer types (`char` and `int`).
- `char` type is by default either signed or unsigned: it depends on your compiler.

Booleans

A boolean is a type that can be evaluated as either `true` or `false`. They are used in control structures.

In the beginning, there was no *boolean* type in *C* and integer types were used instead:

- 0 stated as *false*.
- Any other value stated as *true*.

Going further...

C99 standard introduced `_Bool` type that can contain the values 0 and 1. The header `stdbool.h` was also added: it defines the `bool` type, a shortcut for `_Bool` and the values `true` and `false`. You will learn more about headers later.

Typecast (implicit type conversion)

When an expression involves data of different but compatible types, one can wonder about the result's type. The *C* compiler automatically performs conversion of "inferior" types to the biggest type used in the expression.

```
int   i = 42;
int   j = 4;
float k = i / j; // k equals 10.0
```

The type of `i` and `j` variables is `int`, so the result of the division will have `int` type and will be 10. However, we want to have a `float` type as a result and so we use typecast:

```
int   i = 42;
int   j = 4;
float t = i;
float k = t / j; // k equals 10.5
```

`t` being of `float` type, the result's type becomes implicitly `float` and the value 10.5 is stored in `k`.

1.4 Operators

1.4.1 Binary operators

Arithmetic operators

For arithmetic operations, the usual operators are available:

Operation	Operator
addition	+
subtraction	-
multiplication	*
division	/
remainder	%

Be careful!

The result of a division between two integers is truncated.

Example:

```
float i = 5 / 2;    // i == 2.0
float j = 5. / 2.;  // j == 2.5, note that 5. is equivalent to 5.0
```

Comparison operators

These operators return a boolean result that is either *true* (any value different from 0) or *false* (the value 0) depending on whether equalities or inequalities are, or are not, checked:

Operation	Operator
equality	`==`
difference	`!=`
superior	`>`
superior or equal	`>=`
inferior	`<`
inferior or equal	`<=`

Logical operators

- Logical *OR* `||`:

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

  The previous expression will be true if at least one of the conditions is true, false otherwise.

- Logical *AND* `&&`:

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

  The previous expression will be true if all conditions are true, false otherwise.

The execution of conditions is **left to right**. The following conditions are only evaluated when necessary (*laziness*). For example, with two conditions separated by `&&`, if the first one returns *false*, then the second one will not be evaluated (because the result is already known: *false*). The same goes for a *true* expression on the left of a `||`, the result is obviously *true*.

Example:

```
int a = 42;
int b = 0;
(a == 1) && (b = 42);
// b equals 0, and not 42, because 'b = 42' has not been evaluated
```

Assignment Operators

- Classical assignment: =. This operator allows to assign a value to a variable. The value returned by `var = 4 + 2;` is 6 (the assigned value). This property allows you to chain assignments:

```
int i, j, k;
i = j = k = 42; // i, j and k equal 42
```

Note that the coding style requires one declaration by line.

Tips

+= is a shortcut for `a = a + b`, same goes for -=, *=, /= and %=.

```
int a = 5;
int b = 33;
a += b; // a == 38
int c += a; // does not compile because ``c`` does not exist
```

1.4.2 Unary operators

Negation

The operator -, is used to negate a numeric value. It is the same as a multiplication by -1.

```
int i = 2;
int j = -i; // j == -2
```

Increment/Decrement

In *C* you can use the ++ and the -- operators to respectively increment and decrement by 1 a variable.

When the ++ operator (or --) is placed on the left hand side, it is called pre-increment. It means that the variable will be first incremented and then used in the expression.

On the other hand, when the ++ operator (or --) is placed on the right hand side, it is called post-increment. The variable is first used in the expression and then incremented.

```
int i = 2;
int j;
int k;

j = i++;      // j == 2 and i == 3
k = j + ++i;  // k == 6 and i == 4
```

Not

The `!` operator is used with a boolean condition. Its effect is to reverse the value of the condition:

- if `CONDITION` is *true*, then `!CONDITION` is *false*;
- if `CONDITION` is *false*, then `!CONDITION` is *true*.

1.4.3 Priorities

The following operators are given from highest to lowest priority. Their associativities are also given: left or right. This is not a list of ALL operators in C, rather the most common ones.

Category	Operators	Associativity
parentheses	`()`	Left
unary	`+ - ++ -- !`	Right
arithmetic	`* / %`	Left
arithmetic	`+ -`	Left
comparisons	`< <= > >=`	Left
comparisons	`== !=`	Left
logical	`&&`	Left
logical	`||`	Left
ternary	`?:`	Right
assignment	`= += -= *= /= %=`	Right

In programming languages, *associativity* is to be understood as *operator associativity*. When two operators are of the same precedence, in order to determine how to resolve the order of execution, we look at their respective associativity.

Left associativity indicates that operations are resolved left to right.

Right associativity indicates that operations are resolved right to left.

Example

```c
int a = 1;
int result = ! -- a == 3 / 3;
```

The following rules will be applied in this order to resolve priority issues:

- The unary operators `!` and `--` are the ones with the highest priority. As both of them have right-to-left priority, `--` will be solved before `!`.
- The arithmetic division, `/`, is now the operator with highest priority, so the next operation will be `3 / 3`.
- Finally the `==`, with the lowest priority, will be executed.

We could rewrite this whole operation as:

```c
int result = (!(--a)) == (3 / 3);
```

Tips

Associativity is not always obvious: do not hesitate to add parentheses, even if they are not required, to make some operator priorities explicit and ensure the code is easily readable.

1.5 ASCII

The *American Standard Code for Information Interchange* (abbreviated ASCII) is one of the most widely used encoding standards in the world. It was developed in the 1960s and maps 128 characters based on the English alphabet to numerical values.

Tips

You can see the ASCII table by typing `man ascii` in your terminal.

You should really take a look at the ASCII table and notice a few things:

- Characters are sorted logically, 'a' to 'z' are contiguous, as well as 'A' to 'Z' and '0' to '9'.
- The character '0' does not have the value 0.
- Some characters cannot be printed (for example `ESC` or `DEL`).

In *C*, a variable of type `char` can at least take values from 0 to 127, where each value in this range corresponds to a character following the ASCII table. The value of a `char` variable being a number, numerical operations can be performed on this variable.

```c
#include <stdio.h>

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

    if (c >= 97 && c <= 122)
        puts("'c' has become a lowercase character!");

    return 0;
}
```

File: example1.c

However, this writing is not practical at all as it is hard to read. We will prefer the following:

```c
#include <stdio.h>

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

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

    return 0;
}
```

File: example2.c

You might wonder what the ASCII value of such or such letter is. Truth is that does not matter and is even irrelevant and that you should always use the character itself when performing operations on characters.

1.6 Control structures

1.6.1 Instructions and blocks

A block regroups many instructions or expressions. It creates a **scope** where variables used in expressions can "live". It is specified by specific delimiters: `{` and `}`. Functions are a special kind of blocks. Blocks may be nested and empty.

1.6.2 If ... else

Conditions allow the program to execute different instructions based on the result of an expression.

```
if (expression)
{
    instr1;
}
else
{
    instr2;
}
```

For example:

```
if (a > b)
    a = b;
else
    a = 0;
```

Tips

You can see that there are no braces here, if your block has only one instruction, it is allowed to omit braces.

Ternary operator

This operator allows to make a test with a return value. It is a compact version of `if`.

```
condition ? exp1 : exp2
```

It reads as follow:

```
"if" condition "then" exp1 "else" exp2
```

Example:

```
int i = 42;
int j = (i == 42) ? 43 : 42; // j equals 43
```

1.6.3 While

A loop repeats its instructions while the condition is met.

```
while (condition)
{
    instr;
}
```

Tips

Braces are mandatory only if `instr` is made of several instructions.

Example:

```
int i = 0;

while (i < 100)
{
    i++;
}
```

1.6.4 Do ... while

The condition is checked only after the first run of the loop. Hence, `instr` is always executed at least once.

```
do {
    instr;
} while (condition);
```

Example:

```
int i = 0;

do {
    i++;
} while (i < 100);
```

1.6.5 For

Prefer the more compact `for` loop syntax when you need to repeat the same instructions a known amount of times.

```
for (assignation; condition; increment)
{
    instr;
}
```

Example:

```c
for (int i = 0; i < 10; i++)
{
    // do something 10 times
}
```

1.6.6 Break, continue

- `break`: exits the current loop.
- `continue`: skips the current iteration of a loop and goes directly to the next iteration.

Example:

```c
for (int i = 0; i < 10; i++)
{
    if (i == 2 || i == 4)
        continue;
    else if (i == 6)
        break;
    puts("I am looping!");
}

// The text "I am looping!" will only be printed 4 times.
```

1.6.7 Switch

The `switch` statement allows to execute instructions depending on the evaluation of an expression. It is more elegant than a series of `if` ... `else` when dealing with a large amount of possible values for one expression.

```c
switch (expression)
{
case value:
    instr1;
    break;
/* ... */
default:
    instrn;
}
```

Detail:

- `value` is a numerical **constant** or an enumeration value.
- `expression` must have integer or enumeration type.

It is important to put a `break` at the end of all cases, else the code of the other instructions will also be executed until the first `break`. The `default` case is optional. It is used to perform an action if none of the previous values match.

Example:

```c
switch (a)
{
case 1:
    b++;
    break;
case 2:
    b--;
    break;
default:
    b = 0;
};
```

1.7 Functions

1.7.1 Definition

A function can be defined as a reusable and customizable piece of source code, that may return a result. In *C*, there is barely any difference between functions and procedures. Procedures can be seen as functions that do not have a return value (`void`).

1.7.2 Use

A function is made of a *prototype* and a *body*.

Prototypes follow this syntax:

```c
type my_func(type1 var1, ...);
```

- `type` is the return type of the function (void in case of a procedure).
- `my_func` is the name of the function (or *symbol*) and follows the same rules as variables' name.
- `(type1 var1, ...)` is the list of parameters passed to the function.

If the function has no parameter, you have to put the `void` keyword instead of the parameters list:

```c
type my_func2(void);
```

Definition of the body:

```c
type my_func(type1 var1, type2 var2...)
{
    /* code ... */
    return val;
}
```

The execution of the `return` instruction stops the execution of the function. If the function's return type is not `void`, `return` is mandatory, otherwise it will cause undefined behaviors. If the return type is `void` and that `return` is present, its only use is to end the function's execution (`return;`).

Be careful!

When a function has no parameter, forgetting the `void` keyword can lead to bugs.

Notice the difference between `type my_func(void)` and `type my_func()`:

- The `type my_func(void)` syntax indicates that the function is taking **no arguments**.
- The `type my_func()` means that the function is taking an unspecified number of arguments (zero or more). You must avoid using this syntax.

When a function takes arguments, declare them; if it takes no arguments, use `void`.

Here is an example showing the risk of forgetting the `void` keyword.

```
int foo()
{
    if (foo(42))
        return 42;
    else
        return foo(0);
}
```

If you test this code, you will realize that it compiles and runs causing undefined behavior. However, if you use `int foo(void)` it will generate a compilation error.

1.7.3 Function call

In order to use a function, you need to *call* it, using this syntax:

```
my_fct(arg1, ...)
```

Arguments can either be variables or literal values.

Example:

```
int sum(int a, int b)
{
    return a + b;
}

int a = 43;
int c = sum(a, 5);
```

Tips

If you want to call a function that does not take any argument, just leave the parentheses empty.

Arguments of a function are always passed **by copy**, which implies that their modification **will not** have an impact outside the function.

```
#include <stdio.h>

void modif(int i)
```

(continues on next page)

(continued from previous page)

```
{
    i = 0;
}

int main(void)
{
    int i;

    i = 42;
    modif(i);
    if (i == 42)
        puts("Not modified");
    else
        puts("Modified");
    return 0;
}
```

File: example3.c

The previous example displays `"Not modified"`.

1.7.4 Recursion

It is possible for a function to be *recursive*. The following example returns the sum of numbers from `0` to `i`.

```
int recurse(int i)
{
    if (i)
        return i + recurse(i - 1);
    return 0;
}
```

1.7.5 Forward declaration

Sometimes, it is necessary to use a function before its definition (before its code). In this case, it is enough to write the function's prototype above the location where we want to make the function call, outside of any block. This is the same as declaring the function (to declare that the function exists) without defining it (implementing its body). Hence, the compiler will know that the function exists but that its implementation will be given later.

Example (note the `;` at the end of the prototype):

```
int my_fct(int arg1, float arg2);

int my_fct2(int arg1)
{
    return my_fct(arg1, 0.3);
}
```

(continues on next page)

(continued from previous page)

```c
int my_fct(int arg1, float arg2)
{
    // returns something
}
```

Without the forward declaration, the compiler would tell you it does not know the function `my_fct`.

1.8 The `main` function

The `main` function is the *entry point* of your program. That is to say when your program starts, it executes the `main` function. When the program takes no argument, it will have the following prototype:

```c
int main(void)
```

You will see the prototype of the `main` function with parameters in a couple of days.

The value returned by the `main` function will be the return value of the program. This is why you must remember to give this value with the `return` keyword.

```c
int main(void)
{
    return 42;
}
```

1.9 Writing on the terminal

A program can display information on the terminal using several functions from the standard library. You will discover them progressively but here are the basic ones: `putchar(3)` and `puts(3)`[2].

Prototypes:

```c
int putchar(int c);
int puts(const char *s);
```

Detailed parameters:

- `c`: the ASCII value of the character you want to display.
- `s`: the string that you want to display.

For more information, we invite you to look at the *man* pages of `putchar(3)` and `puts(3)`. Remember that `puts(3)` will add a `\n` at the end of your string.

Tips

These functions are declared in the `stdio.h` header. You must include it in your files by writing `#include <stdio.h>` at the top of the file.

[2] When presenting a concept, you can see a number between parentheses. This number is the section of the man page where it is described[3].
[3] You should go and check the `man(1)` upon seeing those.

Be careful!

The *man* contains prototypes and includes of libC's functions. They are located in the third section. Always look at the *man* pages before calling an assistant.

2 Compilation

To be able to execute a program, you have to translate it into your machine's language. To do so, we use compilers. The one we will mostly use is gcc, the C compiler of the GNU Compiler Collection (GCC with capitals)

To compile your project:

```
42sh$ gcc file.c
```

2.1 Some compiler options

Warnings will not stop compilation, but it is **strongly** advised to take them into consideration because they highlight instructions used in a suspicious way.

When you compile a program with `gcc`, you should specify at least the compiling options used to grade you.

Unless explicitly stated otherwise, the ACU will always check your programs with those flags:

```
-Wextra -Wall -Werror -Wvla -std=c99 -pedantic
```

Here are some useful options (for more see `man gcc`):

- `-o`: to specify the output file's name.
- `-Wall`: display *warnings* in specific cases.
- `-Wextra`: display *warnings* in specific cases, different from those given by `-Wall`.
- `-Werror`: *warnings* are considered as errors.
- `-Wvla`: display a *warning* when an array is declared with a variable size
- `-std=c99`: allow you to use the C99 standard.
- `-pedantic`: reject programs that do not follow ISO standard.

3 Coding Style

In the **Documents** section of the intranet, you will find the EPITA *Coding Style*. You have to read, understand and know everything that is written on this document. If you fail to apply the rules described here, you will lose points.

4 Clang Format

Clang-Format is a code formatter. This means you can run it on a source file and it will output the exact same code, but formatted according to a set of rules and formatting options defined in a configuration file: the *.clang-format* file. This tool can and will be used to check that your code respects EPITA's *Coding Style* on a formatting level, you should use it as well.

A *.clang-format* file is provided on the intranet in the **Documents** section.

5 Writing on standard output

If you want to write to the standard output, several functions can realize these operations and you already used some of them previously.

As you could see, `putchar(3)` and `puts(3)` just take a simple character or a constant string as argument. How about if we want to customize the output with better formatting, or if we want to print a float? This is what `printf(3)` was made for.

Going further...

`printf(1)` also exists. Beware of that when looking up the man page. Also, note that `printf(1)` is very similar `printf(3)` when it comes to how you use it (except it is, of course, in shell). It may be useful to you when writing shell scripts.

5.1 `printf(3)`

Prototype:

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

Printf stands for print format, and is a *variadic* function. A variadic function accepts a variable number of arguments. `printf(3)` arguments are a format string followed by the needed variables or constants.

Example usage:

```
#include <stdio.h>

int main(void)
{
    int day = 12;
```

(continues on next page)

(continued from previous page)

```
    char month[] = "September";
    printf("Today is the %dth of %s %d.\n", day, month, 2000);

    return 0;
}
```

File: example4.c

Here, `%d` and `%s` are what we call *format tags*. A format tag is always composed of the character `%` and a *specifier* (here `d`). When `printf` is called, it replaces the format tag "`%d`", "`%s`" and "`%d`" by the values given in order: `day`, `month` and `2000`.

```
42sh$ gcc -Wextra -Wall -Werror -Wvla -std=c99 -pedantic -o my_hello_word my_hello_word.c
42sh$ ./my_hello_world
Today is the 14th of September 2000.
```

Here is the list of the most common used specifiers:

Specifier	Output
c	Character
d	Decimal integer
s	String
f	Decimal floating point
x	Hexadecimal
u	Unsigned decimal integer
zu	`size_t`

Note that `printf(3)` does not make any difference between `float` and `double`, `float` are always converted to `double`, so you can use `%f` for both. `char` and `short` are converted to `int`.

```
#include <stdio.h>

int main(void)
{
    char a = 'a';
    printf("%c\n", a); // a
    printf("%d\n", a); // 97
    printf("%f\n", a); // 0.000000

    return 0;
}
```

File: example5.c

Why is the last output `0.000000`? Because you tell `printf(3)` to read an integer as a float, and as said by the **man 3 printf**, it is an *undefined behavior*... If you compile with `-Wall`, `gcc` will warn you about it.

5.2 Escape sequence

The `\n` used in the `printf` statements is called an escape sequence. In this case it represents a newline character. After printing something to the screen you usually want to print something on the next line. If there is no `\n` then another `printf` command will print the string on the same line.

Commonly used escape sequences are:

Escape sequence	Output
\n	Newline
\t	Tabulation
\\	Backslash
\"	Double quote
\r	Carriage return

Tips

The carriage return escape sequence can be useful to *re-write* on the same line by resetting the cursor position at the beginning of the line.

5.3 Format output

You can also use `printf(3)` to have a pretty output formatting. We will not detail it here, but you can find lots of information about it in the man page, here is an example:

```c
#include<stdio.h>

int main(void)
{
    int a = 15;
    int b = a / 2;

    printf("%d\n", b);
    printf("%3d\n", b); // Padding with (3 - 'nb_of_digits_of_b') spaces
    printf("%03d\n", b); // Padding with '0's

    float c = 15.3;
    float d = c / 3;

    printf("%.2f\n", d); // 2 characters after point
    printf("%5.2f\n", d); // Padding until 5 characters, 2 characters after point

    return 0;
}
```

File: example6.c

```
42sh$ gcc -Wextra -Wall -Werror -Wvla -std=c99 -pedantic -o my_printer my_printer.c
42sh$ ./my_printer
7
  7
```

(continues on next page)

(continued from previous page)

```
007
5.10
 5.10
```

5.4 Exercise

5.4.1 Print hexa

Goal

Write a function that prints on the standard output his argument in hexadecimal.

```
void print_hexa(int number);
```

Example :

```
int main(void)
{
    print_hexa(42);
    print_hexa(1024);
    print_hexa(0);

    return 0;
}
```

```
42sh$ gcc -Wextra -Wall -Werror -Wvla -std=c99 -pedantic -o hexa hexa.c
42sh$ ./hexa
0x0000002a
0x00000400
0x00000000
```

6 Arrays

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

6.1 One-dimensional arrays

6.1.1 Declaration

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

With `N` being a positive integer setting the size of the array, which is the total number of items that can be stored in the array.

6.1.2 Initialization

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

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

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

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
};
```

It is also possible not to specify the size of an array, **only** if you initialize it during its declaration. The size will then be determined based on the number of values:

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

Here, `arr` is an array of size 5.

6.1.3 Accessing values

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

```
arr[index]
```

- `index` can go from `0` to `N - 1` (`N` being the size of the array).
- `index` can 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[4 + 1];      // Undefined behaviour
```

Going further...

The expression *undefined behaviour* means that this action (in this case, accessing a value out of range of an array) is not specified by the language, and therefore the compiler implements it arbitrarily. The execution may continue, potentially leaving your program in an erroneous state.

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};
```

Be careful!

The first index of an array is **zero** not one.

6.2 Size type

You already know all the basic types in *C*. But other types are defined in headers that you can include in your code. For example, you can use the type `size_t` by adding the following line before your code:

```
#include <stddef.h>
```

As you should know, the `int` type is limited. It has a maximal value[1]. The `size_t` type is always the same size as the type that controls memory addresses. Therefore, on our architecture (`x86_64`), it is twice as big as an `int`, which means the maximum value is much higher. Besides, it is unsigned (i.e. cannot be negative).

As its name suggests, `size_t` is designed to manipulate size values. It is a better choice than `int` when you want to manipulate array indices and sizes because you are sure that, even if your array spans across all your memory, the `size_t` type is big enough to contain the index of the last element.

6.3 Determining size

In *C*, the `sizeof` keyword can be used to determine how many bytes of memory are necessary to store a specific type. The return value of the `sizeof` keyword is always a `size_t`.

You can use the `sizeof` keyword with an array to get the size of the array. However, be careful. It works in the same scope as the array declaration, but, if the array was, for example, given as parameter of a function, `sizeof` inside the scope of the function would not work because the array was cast to a *pointer* and `sizeof` returns the size of the pointer type. You will see pointers in depth later, so just keep in mind that `sizeof` works with arrays but only in the scope of the array declaration.

Here are some examples.

[1] https://en.wikipedia.org/wiki/C_data_types#Basic_types

Going further...

The `const` keyword in *C* is a reserved keyword, part of the type declaration and used to declare constants, e.g. non-alterable-after-declaration data.

```
const int vec[] = {1, 2, 3, 4, 5};
sizeof(vec);                  // 20
sizeof(int);                  // 4
sizeof(vec) / sizeof(int);    // 5
sizeof(vec) / sizeof(vec[0]) // 5
```

The last example shows you how to get a static[2] array's size in a way that is type agnostic[3].

6.4 Exercises

6.4.1 Maximum

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);
```

Tips

INT_MIN can be found in the `<limits.h>` header. However, you should use it sparingly

Be careful!

The inclusion of `<limits.h>` fails on the PIE with pedantic. Therefore, you must compile this exercise without the flag `-pedantic`.

6.4.2 Vice-maximum

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);
```

[2] As opposed to a dynamic array where its size is not known at compile time. You will learn about them soon.

[3] Meaning you, the programmer, do not need to know the type of the elements contained in the array to know the number of elements. You could change the type of the elements in the array without changing this line and still get the correct size!

7 Multi-dimensional arrays

We will take two-dimensional arrays as an example, which are arrays of arrays, but you might as well have `N`-dimensional arrays.

7.1 Declaration

You can declare two-dimensional arrays this way:

```
type var[A][B];
```

`var` is an array of size `A` containing arrays of size `B`. The first dimension is of size `A` and the second of size `B`. Arrays of the last dimension contain `B` elements of type `type`.

Tips

If it helps, you can see a two-dimensional array as a matrix.

7.2 Initialization during declaration

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

The dimension of the **external array** can be omitted:

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

The size is deduced by the compiler.

7.3 Accessing values

```
arr[i][j]
```

For instance, in the previous example `arr[0][2]` is `3` and `arr[1][2]` is `6`.

Tips

For a matrix, previous examples would refer to the first row, third column and second row, third column. Do not forget that, in *C*, the index starts from 0!

7.4 Matrix with a one-dimensional array

Now that you have learned about multi-dimensional arrays, you have to know that it is possible to *emulate* them with a one-dimensional array.

Going further...

A matrix is simply a two-dimensional array.

The following matrix `M`:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

can be represented by the following one-dimensional array:

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

A simple *formula* exists to convert two-dimensional coordinates in one-dimension coordinates (i.e. the index):

```
Index = Row * Width + Column
```

For example, if we wanted to get the index of '6', which is located on the second row and the third column:

Be careful!

Arrays are zero-indexed, so all matrix coordinates have to be decremented by one. As such, if we want the value at M_{23}, we have:

```
Row = (2 - 1)
Column = (3 - 1)
```

```
Index = 1 * 3 + 2
Index = 5
```

And `arr[5] == 6`.

We have just shown you how to emulate a two-dimensional array with a one-dimensional array, but it is also possible to emulate dimensions higher than two although the formula to access elements inside it will be different.

8 Preprocessor directives

Before transforming your source code into a binary executable, your compiler expands special *preprocessor directives*. These preprocessor instructions all start with a `'#'`. You have already used at least one of them: `#include <stdio.h>` to use functions provided by the *C* standard library to do input and output manipulation.

We will explain the full compilation toolchain in detail in a later tutorial, however you need to understand the basics of two important directives: `#define` and `#include`.

8.1 Macros using `#define`

8.1.1 Use case

You will often encounter situations where you use an array's size at multiple places in your code. Here is an example:

```
int arr[5] = { 0 }; // { 0, 0, 0, 0, 0 }

for (size_t i = 0; i < 5; i++)
    arr[i] = i
```

Now imagine you want your array to be of a different size. You would have to change the size in the declaration, but also in the `for` loop. This is tedious, and it is relatively easy to forget to change the size everywhere it is used, especially when you are dealing with source files composed of a few hundreds or thousands lines of code.

The number `5` here is what we call a *magic value*: it has no name associated with it, and it is unclear what its purpose is. Such values make the code more difficult to understand and maintain, and you **want** your code to be understandable and maintainable (and it also helps assistants to be in a better mood when they try to help you). In order to avoid using magic numbers[1] that do not require a variable (because the array size is a constant here), you can use a *macro* like this:

```
#define ARR_SIZE 5

int arr[ARR_SIZE] = { 0 };

for (size_t i = 0; i < ARR_SIZE; i++)
    arr[i] = i;
```

[1] If the value is a number, the term *magic number* can also be used.

8.1.2 Definition

Macros are a simple way to implement text replacement. The following syntax allows you to define a basic macro with no parameters:

```
#define MACRO_NAME replacement-text
```

Whitespace characters before and after the `replacement-text` are not part of the replacement text.

Note that you can define macros on multiple lines.

```
#define MACRO_NAME This is a \
    very long text
```

As a macro is just text replacement, the whitespace between `'a'` and `'\'`, and between the newline and `'very'` are part of the replacement text.

You should try to avoid defining macros inside functions. A nice place to put macros is at the top of the file, below includes.

Tips

By convention macros are named in uppercase. It helps differentiate them from variables and function names.

Be aware that you can do a **lot** more with macros! As always, you will learn more about them later this week.

8.2 Working with multiple files using `#include`

You already know how to include *C* standard library headers, so let us explore how to create your own headers, why you would need to do this, and how to use them with the `#include` directive.

8.2.1 Use case

During exercises and projects, you will have multiple source files to split your code and especially functions into different sections. To use a function declared in another file, you must add the function prototype before using the function to let the compiler know it is defined somewhere else. For example:

```c
// main.c

void my_awesome_function(void);

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

```c
// my_awesome_file.c

#include <stdio.h>

void my_awesome_function(void)
{
    puts("Hello from my_awesome_file.c");
}
```

And to compile your files:

```
42sh$ gcc -Wextra -Wall -Werror -Wvla -std=c99 -pedantic main.c my_awesome_file.c -o myproject
42sh$ ./myproject
Hello from my_awesome_file.c
```

This can quickly become tedious and repetitive to write in each files many function prototypes at the top. To solve this problem, we use **header files** where we put all our prototypes, so we can easily include those files every time we need a specific function.

```c
// main.c

#include "my_awesome_file.h"

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

```c
// my_awesome_file.h

#ifndef MY_AWESOME_FILE_H
#define MY_AWESOME_FILE_H

void my_awesome_function(void);

#endif
```

The compilation step has not changed:

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

42sh$ ./myproject
Hello from my_awesome_file.c
```

8.2.2 Definition

Header files are files ending with a `.h` suffix, which contain various definitions that you might have to use in multiple source files. For now, you only know about function prototypes and macros to put in your header files.

A header file will always have the following structure:

```
#ifndef MY_HEADER_FILENAME_H
#define MY_HEADER_FILENAME_H

// My macros definitions
// My function prototypes
// ...

#endif
```

This format with `#ifndef`/`#define`/`#endif` is what we call **include guards**. There are necessary to prevent infinite inclusions (for example: `file1.h`, which includes `file2.h`, which includes `file1.h`, and so on). The `#ifndef` directive means "if not declared" and extends until the `#endif` directive. You can thus read the guards as: if the macro `MY_HEADER_FILENAME_H` has not been declared, declare it and include everything from the header file, otherwise include nothing. This way, next time the file is included, the macro will be defined and therefore the compiler will not include it again.

Be careful!

You **must** always protect your header files with guards.

Custom header files are included using "", unlike *C* standard library headers which use <>. Headers using <> are looked for in a standard list of system directories (`/usr/include`, ...), while headers using "" are looked for in the directory that the file is in, then in the same standard list.

```
#include <stdio.h>

#include "my_header_filename.h"

// ...
```

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