PROG C — td1

version **#0.0.1-dirty**

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

Copyright

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

Contents

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

1 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 content of 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, you can 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. The best way to understand it is simply to compile it and run it. You have used *Python* before. There is an important difference between *C* and *Python*: **compilation**.

As opposed to *Python* which is an interpreted language, *C* is a **compiled** language, which means that 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, compilation will work. 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 match.

When defining a variable, we have to provide its type before its name. Initialising variable is not mandatory. Nevertheless, it 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:

```c
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:

```
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 include code that is declared elsewhere. This is typically useful to use functions that are not declared locally.

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.

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 `"I can C shell."` 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. In order to tell **gcc** that the declaration of the function is actually in another file, you need to tell it where to 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 is the preprocessor 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 an 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`.

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