Piscine — Tutorial D1 AM

version #

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

Copyright

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

Contents

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

1 Angel's Crypt United

1.1 Let us enter the game

In order to start with the *C* programming language, we will create a little program together.

This program will simulate an epic game of Rock Paper Scissors between forces of good and evil, named `Angel's Crypt United`.

Tips

You have to write and test every piece of code we give you in order to fully understand this part.

First, create a file `main.c` with the following content:

```c
#include <stdio.h>

int main(void)
{
    puts("Welcome to the world of Angel's Crypt United");
}
```

For now, if you compile and run it, this will just print "Welcome to the world of Angel's Crypt United" to the standard output.

```
42sh$ gcc main.c -o acu
42sh$ ./acu
Welcome to the world of Angel's Crypt United
```

So let us explain a bit what is going on here, starting with this line:

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

It is called the `main function`, we will come back to it later. For now, keep in mind that everything between the braces will be executed when you run your program.

```c
puts("Welcome to the world of Angel's Crypt United");
```

This line will print everything that is between the double quotes.

In computer science, blocks of text that we want to use are called `strings`. In *C* a `string` is created when you put text between double quotes.

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

`puts(3)`[1] is a function that prints the given `string`. Because this function is implemented elsewhere, we need to tell the compiler where to find it. We do so by including `stdio.h` at the beginning of the file, which allows us to use `puts(3)`, among other functions that we will see later on.

You have now seen what composes a program that prints a string!

[1] short for "put string".

1.2 Let us play a game

1.2.1 Simulating an action

It is time to code the game itself.

The game will have two players: an Angel and a Demon. We will assign an integer value to each possible action:

Action	Number
Rock	1
Paper	2
Scissors	3

So first, let us simulate a move from both opponents.

```
int main(void)
{
    puts("Welcome to the world of Angel's Crypt United");

    int angel_action = 1;
    int demon_action = 1;
}
```

Here, we use variables. In *C*, a variable is a binding between a name, a type, and a value of this type.

The variables we defined are of the `int` type (short for `integer`). They both hold the value `1`.

`Variables` are used to store values that will be used later in the program execution. Their value can be retrieved, modified or compared with other values.

For now, we will let both the Angel and the Demon play Rock.

1.2.2 Our first condition

We need to know who won the game. In order to do this, we will use the `if` control structure.

The rules of Rock Paper Scissors are:

- If both actions are the same, it is a draw.
- Rock beats Scissors.
- Paper beats Rock.
- Scissors beat Paper.

How can we translate this into *C* code?

First, let us handle the case of a draw:

```
if (angel_action == demon_action)
{
    puts("It is a draw!");
}
```

The `if` clause is represented by the keyword `if` followed by a condition between parentheses and code between braces.

In programming, there is a concept of `true` and `false`. In most programming languages it can be represented by a `boolean` type. In *C* however, `true` and `false` are represented by an `int`. The value `0` is read as `false` and any other value is `true`.

Here, both `variables` must be equal for the condition to be `true`. To evaluate the condition, we use the `comparison operator` for equality ==. A comparison operator is an operator that will compare two values by returning `1` if the comparison is `true` and `0` if `false`.

So, if both values are equal, the program will print "It is a draw!".

You can check it by compiling and running the program as presented above.

If you change the value of either `angel_action` or `demon_action` and test again, the program will not print anything.

Be careful!

Do not confuse the assignment operator = that assigns a value to a variable with the comparison operator == that checks for equality.

An `if` clause may be followed by an `else` clause that will be executed if the condition is `false`.

```
if (angel_action == demon_action)
{
    puts("It is a draw!");
}
else
{
    puts("Someone won!");
}
```

If you change the value of either `angel_action` or `demon_action` and test again, the program will print "Someone won!".

1.2.3 But who won?

You can put an `if` clause within an `else` clause.

```
if (angel_action == demon_action)
{
    puts("It is a draw!");
}
else
{
    if (angel_action == 1)
    {
        if (demon_action == 3)
        {
            puts("The Angel won!");
        }
        else
```

(continues on next page)

(continued from previous page)

```
        {
            puts("The Demon won!");
        }
    }
    else
    {
        if (angel_action == 2)
        {
            if (demon_action == 1)
            {
                puts("The Angel won!");
            }
            else
            {
                puts("The Demon won!");
            }
        }
        else
        {
            if (angel_action == 3)
            {
                if (demon_action == 2)
                {
                    puts("The Angel won!");
                }
                else
                {
                    puts("The Demon won!");
                }
            }
        }
    }
```

However, this code is hard to read because of the indentation. *C* allows us to simplify this by writing `else if` on the same line.

```
if (angel_action == demon_action)
{
    puts("It is a draw!");
}
else if (angel_action == 1)
{
    if (demon_action == 3)
    {
        puts("The Angel won!");
    }
    else
    {
        puts("The Demon won!");
    }
}
else if (angel_action == 2)
{
    if (demon_action == 1)
```

(continues on next page)

(continued from previous page)

```
    {
        puts("The Angel won!");
    }
    else
    {
        puts("The Demon won!");
    }
}
else if (angel_action == 3)
{
    if (demon_action == 2)
    {
        puts("The Angel won!");
    }
    else
    {
        puts("The Demon won!");
    }
}
```

This is better, yet we can still go further.

```
if (angel_action == demon_action)
{
    puts("It is a draw!");
}
else if (angel_action == 1 && demon_action == 3
        || angel_action == 2 && demon_action == 1
        || angel_action == 3 && demon_action == 2)
{
    puts("The Angel won!");
}
else
{
    puts("The Demon won!");
}
```

Here, we use the `logical operators` `or` (`||`) and `and` (`&&`). You have seen the `comparison operator` `==` that compares its two arguments; `logical operators` perform operations on `booleans`, and return their result. Like `arithmetic operators` (`+`, `*`, ...), they have a precedence: `&&` operators are evaluated before `||`.

This big condition can be translated into: “Angel plays Rock and Demon plays Scissors, or Angel plays Paper and Demon plays Rock, or Angel plays Scissors and Demon plays Paper”. In a nutshell, the angel won if this condition is true.

1.2.4 Can we play now?

Now, we want to be able to actually play the game. In order to do that, the game must ask the user for input.

Add those lines above the `main` function.

```c
int is_action_valid(int action)
{
    return action >= 1 && action <= 3;
}

int read_int_from_terminal(void)
{
    int value = 0;
    scanf("%d", &value);
    return value;
}

int get_action(void)
{
    puts("Choose an action:");

    int action = read_int_from_terminal();

    while (!is_action_valid(action))
    {
        printf("'%d' is not a valid action\n", action);
        puts("Choose another action:");

        action = read_int_from_terminal();
    }

    return action;
}
```

Those three blocks are called `functions`. In *C*, a `function` is defined by a return type, a name and a, possibly empty[2], list of `parameters`. The code between the braces will be executed every time the function is called in the program.

An `argument` is a `variable` passed to a function. The behavior of a function depends on its `arguments`.

The first one is a function named `is_action_valid` that takes an integer `action` and returns whether `action` is between 1 and 3.

The second one reads an integer from the terminal and returns it.

The third one is a function named `get_action` that takes no arguments and returns a valid action. In this function lies a new control structure: the `while` loop. A loop is a control structure that will repeat a set of directives while a condition is met.

Simply put, **while** the user enters an invalid number, a new input is asked from the terminal.

Now, you just have to change the assignment of your two variables, `angel_action` and `demon_action`, by the return value of the function `get_action()`. Replace both lines by the following:

[2] If the function does not take any arguments, you **must** specify `void` between the parentheses.

```c
int angel_action = get_action();
int demon_action = get_action();
```

Because `get_action()` does not take any arguments, we leave the parentheses empty.

Going further...

If your function takes one argument, you need to put a variable of the corresponding type between the parentheses.

For example, we can call `is_action_valid()` like this:

```c
int valid = is_action_valid(2);
int not_valid = is_action_valid(6);
```

Congratulations, we just finished writing a Rock Paper Scissors!

2 Introduction to *C*

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

2.2 Syntax reminders

2.2.1 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
*/

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

2.3 Variables and data types

2.3.1 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
```

2.3.2 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`.

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`.

2.4 Operators

2.4.1 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
```

2.4.2 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*.

2.4.3 Priorities

The following operators are given from highest to lowest priority. Their associativities are also given: left or right.

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

```
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.

2.5 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;
}
```

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!");
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

2.6 Control structures

2.6.1 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.

2.6.2 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
```

2.6.3 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++;
}
```

2.6.4 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);
```

2.6.5 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:

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

2.6.6 Break, continue

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

Example:

```
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.
```

2.6.7 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.

```
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:

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

2.7 Functions

2.7.1 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`).

2.7.2 Use

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

Prototypes follow this syntax:

```
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:

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

Definition of the body:

```
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.

2.7.3 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.

```c
#include <stdio.h>

void modif(int i)
{
    i = 0;
}

int main(void)
{
    int i;

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

The previous example displays `"Not modified"`.

2.7.4 Recursion

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

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

2.7.5 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):

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

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

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`.

2.8 The `main` function

The `main` function, when the program takes no argument, 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;
}
```

2.9 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.

[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.

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.

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.

3 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` which stands for "GNU compiler collection".

To compile your project:

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

3.1 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 -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.
- `-std=c99`: allow you to use the C99 standard.
- `-pedantic`: reject programs that do not follow ISO standard.

4 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.

It is my job to make sure you do yours.