PISCINE — Tutorial D1

version **#v3.1.1**

ASSISTANT C/UNIX 2021 `<assistants@tickets.assistants.epita.fr>`

Copyright

Contents

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

Be careful!

At this point it is very important for every one to have access to the EPITA Discord server. If it is not done yet, take the time at the beginning of this tutorial. You will find the invite link in the newsgroup *assistant.news*. If you are having trouble with it, do not hesitate to ask an assistant for help.

1 Resources

We've made a website available, that contains information, resources, and documentation about various tools and services that you will use during this year. This website is available here:

https://assistants.epita.fr/doc/

We advise you to read through the content of the online exercises section[1] before you submit your first online exercise.

2 Numeral system

2.1 Concept

Numeral systems are ways to represent numbers with symbols. Other than having a unique representation per number or defining sets of numbers (integers, rationals...), a numeral system has a finite amount of symbols, which define a base.

For a base n we have (with a, b, c, d and e belonging to the symbols of the base):

...	n^4	n^3	n^2	n^1	n^0
...	e	d	c	b	a

In the end, the value of a number *x* represented in base n can be found:

```
x = a * n^0 + b * n^1 + c * n^2 + ...
```

We mostly use base 10, with digits (Hindu-Arabic numerals). Keep in mind that each base-n systems is just a representation. In fact, each and every number expressed in a base can be expressed in another base as well.

[1] https://assistants.epita.fr/doc/exercises.html

2.2 Binary

Only two symbols: 1 and 0. The values 1 and 0 are thus the maximal and minimal values you can get with a single digit in this numeral system. Represents perfectly the on and off states in which a transistor can be.

Let's take the example of 42. To get the binary form of 42, we try to decompose it with powers of 2.

```
42 = 32 + 8 + 2
42 = 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0
```

So 42 is 101010 in binary.

But with big numbers this method does not work well. You can use euclidean division to find the same results and with no knowledge of powers of 2:

```
42 = 2 * 21 + 0
21 = 2 * 10 + 1
10 = 2 * 5  + 0
5  = 2 * 2  + 1
2  = 2 * 1  + 0
1  = 2 * 0  + 1
```

When reading the remainders from the bottom to the top, you get 101010 which is the correct form of 42 in binary.

Now you know how to go from decimal to binary. Let's try doing the opposite. Try to understand why 1001 in binary is equal to 9 in decimal.

As you probably guessed,

```
1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 1 * 8 + 1 * 1 = 9
```

You can try computing the examples below by yourself.

```
101010                --> 42
1100 + 100 = 10000    --> 12 + 4 = 16
11111111              --> 255
100000000             --> 256
```

2.3 Octal

Eight symbols: 0 to 7. The values 7 and 0 are thus the maximal and minimal values you can get with a single digit in this numeral system.

Just like you did with the binary, you can use euclidean division to find the octal form of a number.

```
42 = 8 * 5 + 2
5  = 8 * 0 + 5
```

When reading the remainders from the bottom to the top, you get 52 which is the correct form of 42 in octal.

Now you know how to go from decimal to octal. Let's try doing the opposite. Try to understand why 112 in octal is equal to 74 in decimal.

As you probably guessed,

```
1 * 8^2 + 1 * 8^1 + 2 * 8^0 = 64 + 8 + 2 = 74
```

You can try computing the examples below by yourself.

```
14 + 4 = 20             --> 12 + 4 = 16
777                     --> 511
1000                    --> 512
```

In C, we use the prefix **0** (zero) to use octal numbers.

```c
int main(void)
{
  int x = 052;
  // here x is equal to 42
}
```

2.4 Hexadecimal

Sixteen symbols: 0 to 9 then a to f. The values f and 0 are thus the maximal and minimal values you can get with a single symbol in this numeral system.

Just like you did with the binary and the octal, you can use euclidean division to find the hexadecimal form of a number.

```
42 = 16 * 2 + 10
2  = 16 * 0 + 2
```

When reading the remainders from the bottom to the top, you get 2 and 10. The 10 becomes an 'a'. So the correct hexadecimal form of 42 in hexadecimal is 2a.

Now you know how to go from decimal to hexadecimal. Let's try doing the opposite. Try to understand why CAFE in hexadecimal is equal to 51966 in decimal.

As you probably guessed,

```
C            A            F            E
12 * 16^3 + 10 * 16^2 + 15 * 16^1 + 14 * 16^0 = 49152 + 2560 + 240 + 14 = 51966
```

You can try computing the examples below by yourself.

```
c + 4 = 10              --> 12 + 4 = 16
1ff                     --> 511
200                     --> 512
```

In C, we use the prefix **0x** to use hexadecimal numbers.

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

(continues on next page)

(continued from previous page)

```
  int x = 0x2a;
  // here x is equal to 42
}
```

3 Access control

3.1 Concept

Access control is a notion attached to every file. For each file[1], there is a list defining the individuals and their authorized actions.

In this section, we will look at the basic access control mechanism.

3.2 Ownership

For each file, it is possible to define three different sets of permissions:

- Permissions for the user owning the file. For those you created, it is you.
- Permissions for the group owning the file. Each user can be in multiple groups including one default group. Files you created are given to your default group[2].
- Permission for others, i.e. anybody that is neither the owner nor in the owning group.

It is possible to see the permissions associated with a file using the command `ls -l`. Let's take a closer look at an example:

```
42sh$ ls -l testsuite
-rwxr-xr-- 1 xavier.login students 26K Jul 29 04:32 testsuite
```

Here, the file is owned by the user *xavier.login* and the *students* group.

3.3 Permissions

Linux offers three types of permissions on a file, each of them being associated with a letter:

- **r**eading, noted '`r`'
- **w**riting, noted '`w`'
- **ex**ecuting, noted '`x`'

When the permissions of a file are represented, the three categories of users and their associated rights are displayed in this order: *owning user, owning group, others.*

For instance, when `ls` displays 'rwxr-xr–' (like in the last example) we can analyze 3 types of rights:

[1] And thus directory, since everything is a file in a UNIX system,
[2] You can see which groups you belong to using the command `groups`.

`rwx`	`r-x`	`r--`
Owner permissions	Group permissions	Others permissions

- The first three characters show the permissions of the user that owns the file
- The three middle characters show the permissions of the users that are in the file group
- The last three characters show the permissions of the users that are neither the file owner nor in the file group (others)

Thus, here the *xavier.login* user has *read*, *write* and *execute* permissions on the file. Users in the *students* group have *read* and *execute* permissions and all other users have *read* permissions only.

Tips

The '–' character means that the permission is not present.

3.4 Octal representation

Each right is associated with a numeric value:

- **r**eading, noted '`r`' is value 4
- **w**riting, noted '`w`' is value 2
- e**x**ecuting, noted '`x`' is value 1

As you may have noticed, the numeric values correspond to powers of 2:

```
value : 4 2 1
letter: r w x
```

Thus, you can write a permission on 1 byte by simply adding all the values. For instance '`r-x`' has value `4 + 0 + 1 = 5`.

Here are all the combinations of permissions and their numerical representation:

0	`---`	no permissions
1	`--x`	execute permission (for a file: right to execute, for a directory: right to list content)
2	`-w-`	write permission
3	`-wx`	write and execute permission
4	`r--`	read permission
5	`r-x`	read and execute permission
6	`rw-`	read and write permission
7	`rwx`	all three permissions

Going further...

UNIX permissions are strictly inclusive. This means if a user owns a file and is in the group of the file and the file permissions are `---rwx---` the user will not be able to do anything with this file because his owner rights overrule his group rights.

Going further...

For your information, `ls -l` does not only supply permissions information.

```
42sh$ ls -l testsuite
-rwxr-xr-- 1 xavier.login students 26K Jul 29 04:32 testsuite
```

1	`-rwxr-xr--`	Permissions
2	`1`	Number of entries physically attached to the entry
3	`xavier.login`	User owning the entry
4	`students`	Group of users owning the entry
5	`26K`	Approximated entry size
6	`Jul 29 04:32`	Last modification date
7	`testsuite`	Name of the entry

Going further...

To understand the `Number of entries physically attached to the entry`, take a look at the man page of link(2) and symlink(2) or in the glossary.

3.5 Some commands

3.5.1 `chmod`

The `chmod` command changes the permissions associated with a file. `chmod` means *CHange MODe*. This command accepts two representations of the permissions.

3.5.2 Octal representation:

As you have seen above, it is possible to specify the read-write-execute permissions as a numeric value. By concatenating the values obtained for the user, the group and the others, we get a three digits number that represents the permission in the octal base.

For example, the `rwxr-xr-x` permissions in octal representation is `755`, `rw-r-----` is `640`.

`chmod` allows this representation to specify the new permissions to be applied to a file. The syntax is straightforward, first the permissions then the file(s) to which they are applied.

For example, to give the `640` permissions to the file named `not_executable`, you will use:

```
42sh$ chmod 640 not_executable
```

3.5.3 Symbolic representation:

`chmod` also accepts a different and more explicit notation: the symbolic representation. The attribution of permissions is defined using symbols, split in two kinds:

1. Permission symbols:
    - '`r`' for the *Read* permission;
    - '`w`' for the *Write* permission;
    - '`x`' for the *eXecute* permission.
2. Group symbols:
    - '`u`' for the owning *User*;
    - '`g`' for the owning *Group*;
    - '`o`' for the *Others*.

We can specify if we want to add or remove the permissions using operators:

- '+' to *give* the permissions;
- '–' to *take* the permissions.

To modify multiple groups at the same time using the symbolic notation, it is possible to combine sequences of symbols separated by commas. For example, you can give the `640` octal mode using:

```
42sh$ chmod u+rw,u-x,g+r,g-wx,o-rwx not_executable
```

Going further...

AFS augments and refines the standard UNIX scheme for controlling access to files and directories. For example, in your afs directory, you can view the permissions of a directory with:

```
42sh$ ls -la my_directory
```

If you have time, you can check this link: http://docs.openafs.org/UserGuide/HDRWQ44.html#HDRWQ45

Going further...

For those who want to go further, you might also want to take a look at the `chown` man page to see how to change the owner of a file.

4 Angel's Crypt United

4.1 Let's enter the game

In order to start with this 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's 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

4.2 Let's play a game

4.2.1 Simulating an action

Now, let's create the game.

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's simulate a move from both opponents.

```c
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.

4.2.2 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's handle the case of a draw:

```c
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 won’t 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.

```c
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!”.

4.2.3 But who won?

You can put an if clause within an else clause.

```c
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.

4.2.4 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!

5 Introduction to C

5.1 The birth of C

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

- BCPL by M. Richards (1967)
- B, developed in 1970 at Bell labs

The first version of the compiler was written, in 1972, by Dennis M. Ritchie. In 1975, Alan Snyder writes and presents the PCC (***P**ortable **C** **C**ompiler*), that will become the most popular compiler of its time. The language was normalized by IEEE in 1987, and by ISO two years later (ISO/IEC 9899).

5.2 Syntax reminders

5.2.1 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:

```c
// 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)

(continued from previous page)

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

5.3 Variables and data types

5.3.1 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 */
```

5.3.2 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 `char` and other integer types
- `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 should still use the integer type, in order to express a boolean value.

5.3.3 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. But 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`.

5.4 Operators

5.4.1 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 aren't, 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 -=, *=, /=, %=

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

5.4.2 Unary operators

Negation

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

```c
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.

```c
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*.

5.4.3 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

Associativity is to be understood, in programming languages, 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 hightest 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:

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

Tips

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

5.5 ASCII

5.5.1 ASCII

In **C** a variable of type `char` can take values from `0` to `255`. Each value in this range corresponds to a character, following the ASCII table

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:

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

The value of a `char` variable being a number, numerical operations can be performed on this variable.

```c
#include <stdio.h>

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

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

    return 0;
}
```

But this writing is not practical at all as it is hard to read. Thus, we will prefer the following.

```c
#include <stdio.h>

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

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

    return 0;
}
```

5.6 Control structures

5.6.1 Instructions and blocks

A block regroups many instructions or expressions. It creates a context 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.

5.6.2 If

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

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 */
```

5.6.3 While

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

Tips

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

Example:

```
int i = 0;

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

5.6.4 Do ... while

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

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

Example:

```
int i = 0;

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

5.6.5 For

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

Example:

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

5.6.6 Break, continue

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

5.6.7 Switch

```
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;
};
```

5.7 Functions

5.7.1 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`).

5.7.2 Use

A function is made of a *prototype* and a *body*.
Prototype's 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 generate 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 behaviour. But, if you use `int foo(void)` it will generate a compilation error.

5.7.3 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:

```c
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.

5.7.4 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;
}
```

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
```

(continues on next page)

(continued from previous page)

```
        puts("Modified");
    return 0;
}
```

The previous example displays `"Not modified"`.

5.7.5 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):

```
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`.

5.8 The main function

The `main` function, when the program takes no argument, will have the following prototype:

```
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.

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

5.9 Writing on the terminal

Sometimes it can be useful that a program displays information on the terminal. Several functions can achieve these operations: you will discover several of them progressively but here are some basic ones: `putchar(3)` and `puts(3)`[1].

Prototypes:

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

Detailed parameters:

- `c`: it's the ASCII value of the character you want to display.
- `s`: represents 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.

Be careful!

Note: 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.

5.10 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
```

5.10.1 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`, it is strongly advised to use at least the compiling options used to grade you.

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

[1] When presenting a concept, you can see a number between parentheses. This number is the section of the manpage where it is described[2].

[2] And you should go and check the `man(1)` uppon seeing those.

```
-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

5.11 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.

6 Writing on standard output

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

As you could see, `putchar(3)` and `puts(3)` just take a simple character or a const 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.

Tips

Note, that `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.

6.1 printf

Prototype:

```c
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:

```c
#include <stdio.h>

int main(void)
{
    int day = 12;
    char month[] = "September";
    printf("Today is the %dth of %s %d.\n", day, month, 2000);

    return 0;
}
```

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 -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) doesn't make any difference between float and double, float are always converted to double, so you can use %f for both. Note that char, short, and enumerations are converted to int.

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

    return 0;
}
```

Why is the last output 0.000000 ? Because you tell printf 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.

6.2 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

Going further...

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

6.3 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 manpage, 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;
}
```

```
42sh$ gcc -Wextra -Wall -Werror -std=c99 -pedantic -o my_printer my_printer.c
42sh$ ./my_printer
7
  7
007
5.10
 5.10
```

6.4 Exercise

6.4.1 Print hexa

Write a program that prints on the standard output the first argument number in hexadecimal. You can use the `atoi` function.

Example :

```
42sh$ gcc -Wextra -Wall -Werror -std=c99 -pedantic -o hexa hexa.c
42sh$ ./hexa 42
0x0000002a
42sh$ ./hexa 1024
0x00000400
42sh$ ./hexa 0
0x00000000
```

Going further...

argc and *argv* main variables will be explained more precisely later on, do not worry about them for now.

The only way out is through