Piscine — Tutorial D1 PM

version **#**

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

Copyright

Contents

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

1 Writing on standard output

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

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

Going further...

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

1.1 `printf(3)`

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

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

1.2 Escape sequence

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

Commonly used escape sequences are:

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

Tips

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

1.3 Format output

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

```
#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
```

1.4 Exercise

1.4.1 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.

2 Arrays

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

2.1 One-dimensional arrays

2.1.1 Declaration

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

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

2.1.2 Initialization

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

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

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

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

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

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

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

Here, arr is an array of size 5.

2.1.3 Accessing values

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

```
arr[index]
```

- `index` can go from `0` to `N - 1` (`N` being the size of the array).
- `index` can be an arithmetic expression.

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

a = arr[2];         // OK
a = arr[3 + 1];     // OK
a = arr[4 + 1];     // Undefined behaviour
```

Going further...

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

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

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

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

Be careful!

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

2.2 Size type

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

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

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

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

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

2.3 Determining size

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

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

Here are some examples.

Going further...

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

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

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

2.4 Exercises

2.4.1 Maximum

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

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

[2] As opposed to a dynamic array where its size is not known at compile time. You will learn about them soon.
[3] Meaning you, the programmer, do not need to know the type of the elements contained in the array to know the number of elements. You could change the type of the elements in the array without changing this line and still get the correct size!

2.4.2 Vice-maximum

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

Prototype:

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

3 Multi-dimensional arrays

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

3.1 Declaration

You can declare two-dimensional arrays this way:

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

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

Tips

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

3.2 Initialization during declaration

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

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

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

The size is deduced by the compiler.

3.3 Accessing values

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

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

Tips

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

3.4 Matrix with a one-dimensional array

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

Going further...

A matrix is simply a two-dimensional array.

The following matrix `M`:

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

can be represented by the following one-dimensional array:

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

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

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

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

Be careful!

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

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

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

And `arr[5] == 6`.

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

4 Preprocessor directives

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

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

4.1 Macros using `#define`

4.1.1 Use case

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

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

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

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

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

```
#define ARR_SIZE 5

int arr[ARR_SIZE] = { 0 };

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

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

4.1.2 Definition

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

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

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

Note that you can define macros on multiple lines.

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

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

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

Tips

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

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

4.2 Working with multiple files using `#include`

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

4.2.1 Use case

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

```c
// main.c

void my_awesome_function(void);

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

```c
// my_awesome_file.c

#include <stdio.h>

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

And to compile your files:

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

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

```c
// main.c

#include "my_awesome_file.h"

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

```c
// my_awesome_file.h

#ifndef MY_AWESOME_FILE_H
#define MY_AWESOME_FILE_H

void my_awesome_function(void);

#endif
```

The compilation step has not changed:

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

4.2.2 Definition

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

A header file will always have the following structure:

```c
#ifndef MY_HEADER_FILENAME_H
#define MY_HEADER_FILENAME_H

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

#endif
```

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

Be careful!

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

Custom header files are included using "", unlike *C* standard library headers which use <>.

```c
#include <stdio.h>

#include "my_header_filename.h"

// ...
```

It is my job to make sure you do yours.