Universidad de Alcalá
Departamento de Automática

COMPUTER SCIENCE

Assignment 2.

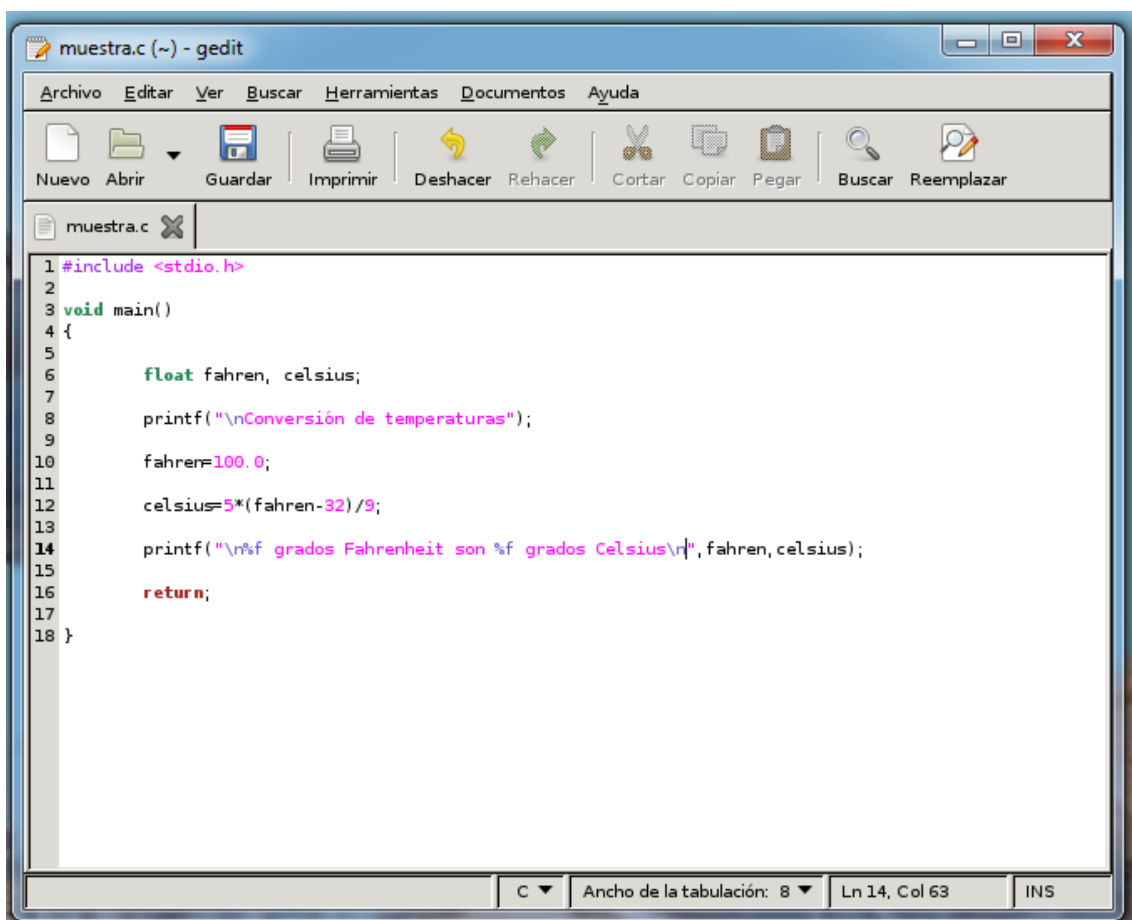
Program development.
Standard input and output in C.

GIEyAI
Fall Semester, 2022-2023
v2.7(07.09.22)

CREATION OF A PROGRAM

For some of the exercises of the upcoming coding assignments, the first thing to do is the design of the algorithm. Once this is clear, one can proceed to developing the creation of the program, which will be a mere translation into code. For the simpler exercises, the program is contained in a single source file. To create it, one can use any basic text editor, such as `vim` or `emacs`. For this assignment, we are going to use an editor with a graphical interface included in the Ubuntu distribution. It is called `gedit`.

With the editor open, one can write the program in it. While coding, it is advisable to save often to avoid loss in case of a power outage or similar reasons. The source file usually has the `.c` extension. In this case, we use `muestra.c` as a name. The following figure shows the editor with the code of a sample program.



In order to run it, the program must be compiled and linked first. These are two different operations, but for a program with a single file of source code, they can be carried out with a single invocation of the `gcc` program. `gcc` is an application which includes (among other things) the compiler and the linker. The way to use it is:

```
bash-ln.05$ gcc -g -o muestra muestra.c
```

(Note that everything up to, and including, the '$' is called the *prompt*, and is printed by the terminal. The command starts with 'gcc'). The -g flag tells gcc to generate debugging information (otherwise it won't be generated, and it won't be possible to debug the code later). Next, the -o name argument tells gcc what to call the output executable file. In this case, muestra. The last argument in the line is the source file to compile.

If the program has syntax errors, gcc will indicate which are the faulty lines, and will give a short description of the problem. Note that, in this case, it won't generate an executable file! If the program is error-free (at least as far as syntax is concerned), the result will be the executable file.

MANAGEMENT OF STANDARD INPUT AND OUTPUT IN C.

This section deals with the use of standard input and output in a C program. The starting point is the basic program which was created in the previous section and, with it in mind, the most relevant questions about input and output are dealt with.

Basic structure of a C program.

The basic structure of a simple C program is taken from the slides of unit 4, section 1, which is very similar to the one which was created in the previous section. It is shown below. In the code shown, line numbers are included (shown here for reference only; these numbers are **not to be included** in the source file).

```
1 /* Fahrenheit -> Celsius conversion */
2
3 #include <stdio.h>
4
5 int main()
6 {
7    int fahren, celsius;    /* integer variables */
8
9    printf("Conversion from °F to °C:\n");
10
11   /* Temperature to be converted */
12   fahren = 100;
13
14   /* Conversion */
15   celsius = 5*(fahren-32)/9;
16
17   /* Show results */
18   printf("%d °F = %d °C\n",fahren, celsius);
19   return 0;
20 }
```

The program has 2 main sections: the main function (the second section, from line 5 onwards), and everything that precedes it (the first section, lines 1 to 4). Section 1 is where library inclusion directives go (stdio.h in this example), as well as the declaration of global variables (there are none in this example). The main function contains the statements of the main program.

This assignment deals with some of the most widely used input/output functions in C. They (unsurprisingly) manage the program's standard input and output. By default, these are the keyboard and the screen, but this can be changed by redirection from the command line (recall assignment 1).

All these functions are in the `stdio.h` library so, in order to use them, it will be necessary to add the corresponding `#include` directive in the program.

In the sample program, the lines with the word `printf` (9 and 18) are performing output operations. `printf` corresponds to the name of a function which manages the task of showing in standard output (remember: by default, the screen) whatever we tell it to. The full description of the `printf` function, and of some others, is below.

Standard output.

The main function available for outputting data is `printf`. Its syntax is:

```
int printf(const char *format, …)
```

The function can show, on standard output, the data it is told to, and it formats the output according to directions that can be given to it. These directions make up the first argument of the function. It is a string of characters with interspersed control sequences. After this first argument, a number of other arguments can be included (that is what the ellipsis – '…' – represents), which are the data that must be displayed on the output. Each of them is embedded into the string of text in place of the control sequence, in the manner which it (the corresponding control sequence) determines. Each format control sequence starts with the character `'%'`, and after that comes the information about how to display the data that corresponds to it. The syntax of a format control sequence is:

```
%[flags][width][.precision][length]format
```

Each field is explained next:
`flags` (optional)
- «-» left justification
- «+» sign symbol is always shown
- «0» zero-padding to the left

`width` (optional): width of the field where the data is displayed

`precisión` (optional)
- for integers: number of digits
- for reals: number of decimal positions
- for strings: number of characters

`length` (optional) type of format to use
- «h» expects a short
- «l» for integers, `printf` expects a `long`; for `float`, no effect.
- «L» for floating point types, `printf` expects a `long double`

`format` (mandatory) tells the type of the argument
- «d» `signed int` printed in decimal
- «u» `unsigned int` printed in decimal
- «o» `unsigned int` printed in octal
- «x» `unsigned int` printed in hexadecimal (X for capitals)
- «f» `float` printed in the format `[-]ddd.ddd`

«e» `float` printed in the format `[-]d.ddde[±]ddd` (E for capitals)
«g» `double` in the most appropriate notation for its magnitude
«c» `char`
«s» `string`

According to the above, the invocation of `printf` in line 9 has no format control sequence, so it will just print on standard output the characters which make up the control string as such.

The call in line 18, on the contrary, has two format control sequences. Specifically, the sequence `"%d"` appears twice. They are two simple sequences where only the format is specified, that is, the type of the data to display. In this case, we have two integers (hence `d`). The variables shown are the ones included in the function call next to the control sequence, in the order that they appear: `fahren` (for the first control sequence) and `celsius` (for the second one). The values of both variables are embedded in the output string in the positions indicated by the format control sequence `"%d"` (each variable in its corresponding position).

Format control sequences allow us to specify, up to a very high level of detail, how we want to display the data. Here follow some examples with more complex format control sequences (you can try them on a program of your own and see what happens; use the example above as a template and modify it as you see fit):

Examples:
For an `int` variable, `data`, printed in 8 positions, with zero-padding on the left, up to 6 digits, in `signed int` format:

```
printf("%8.6d",data);
```

To print `data`, an `unsigned int`, with zero-padding of up to 8 positions, left justified, in a space of 12 positions:

```
printf("%-12.8u",data);
```

To print `data`, an `int`, in hexadecimal (2's complement):

```
printf("%x",data);
```

The following example prints a string of characters created as a variable in the program. A string of characters is a collection of characters, set one next to the other, so that, together, they make up the message.

```
char string[11]="one string";

printf("%s",string);
```

In the example, variable `string` is declared as `char`, the type which corresponds to characters. The number in square brackets (11) is the maximum number of characters that the string can hold. This number must be, at least, one more than the number of desired characters (in this case, 10, the number of characters in "`one string`"). With this declaration, a call can be made to function

printf to tell it to print the characters to show, simply using the name of the set, which, in this case, is string.

A string of characters can be embedded in the format control sequence:

```
char string[20]="some message";
printf("%s: \nEl dato es: %-12.8u",cadena,dato);
```

In this case, the characters in the string are printed to standard output, and the content of variable dato is embedded in the place marked by the escape sequence ("%-12.8u").

printf allows a high level of detail on how the output is shown. For simpler, more specific tasks, there are other easier functions:

int puts(const char *): to output a simple character string.

int putchar(int): to output a single character.

Standard input.

Similarly to the printf function, there exists an analogous input function, scanf. The functionality is equally extensive. The syntax is:

```
int scanf(const char *format, …)
```

In this case, the format string, which is also a control sequence, is not sent to the output, but specifies instead what the input is expected to be. The function ignores any whitespace characters (blanks, tabs and carriage returns) it finds before the next character. A whitespace character in the format string is taken to mean any amount of them found in the standard input (even none). Just as with printf, in scanf format control sequences are allowed, in this case with the following syntax:

```
%[*][width][length] format
```

The meaning of the different fields is:

*: The data is read, but not assigned to any variable.

width (optional): Number of characters to read (the rest are ignored).

length (optional): modifies the storage expected from the variable which will contain the data:

h	unsigned short int
hh	unsigned char
l	unsigned long int
ll	unsigned long long int
L	long double

format (mandatory). Specifies the data type. Accepted values are the same as for the analogous field in printf.

Examples:
To read a 6-digit integer:

```
scanf("%6d",&data);
```

To read a string of characters:

```
char string[20];
scanf("%20s",string);
```

To read two ints:

```
scanf("%d %d",&data1,&data2);
```

NOTE: In this last example, when working with a keyboard, the user will have to type first the first `int`, then an arbitrary number of whitespace characters, and then the second `int`. To make it more flexible, it is more advisable to read first the first `int` and, on a second call, the second `int`:

```
scanf("\n%d",&data1);
scanf("\n%d",&data2);
```

Notice the newline characters in the format string (`"\n"`). They are meant to remove any carriage returns left over from the previous call to `scanf`.

Just as with output, there are easier-to-use functions for the input:

`char *gets(char *)`: reads a string of text.
`int getchar(void)`: reads a single character.

EXERCISES.

1. Write a program that will do the following: it will have a text string variable, with an initial value (of the programmer's choice), and it will print that string on the screen.
2. Write a program that reads in a text string from the keyboard (storing it in a variable) and then shows that string on the screen (it is possible to create an empty string variable just declaring it without an initial value).
3. Write a program that reads in two signed integer numbers from the keyboard and shows their addition on the screen. Note: it is advisable to read the numbers one by one, each with its own call to the input function, instead of both in a single call.
4. Modify the previous program to work with unsigned numbers. Note that the program can't avoid it if the user inputs a signed number. The user must know what it is doing when using the program.
5. Modify the previous program to work with floating point numbers.
6. Write a program that reads an unsigned integer from the keyboard and shows its conversion to hexadecimal on the screen.
7. Modify the previous program to read a signed integer.