Systems Architecture

3. Input/Output in C

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2024/2025

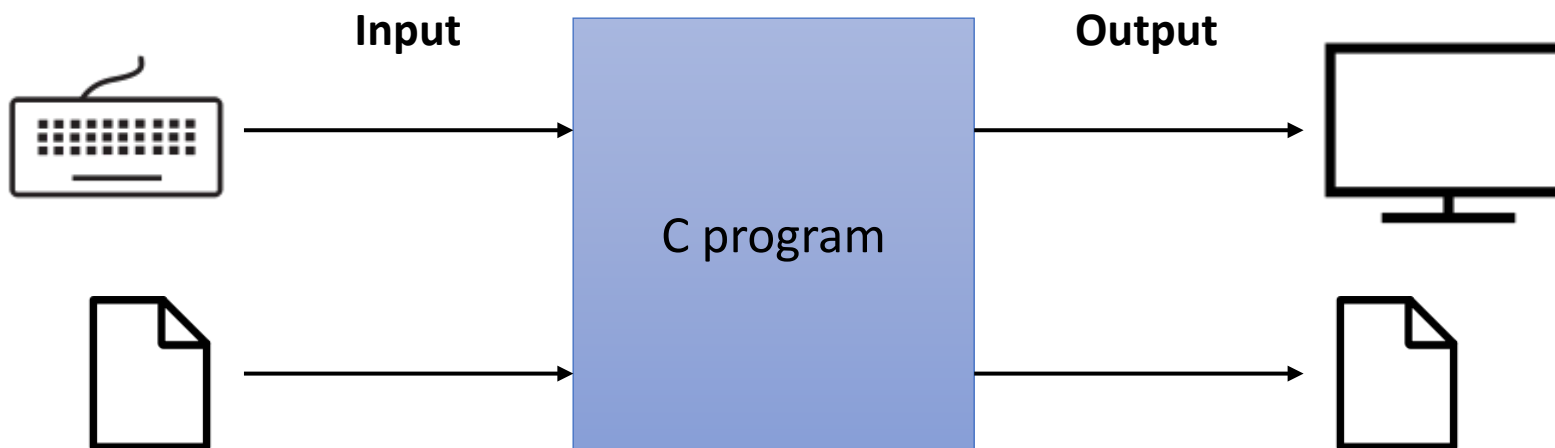uc3m | Universidad **Carlos III** de Madrid

Table of contents

1. Introduction

- In C programming:
  - **Input** means to get some data from a source into a program
  - **Output** means to put some data from a program to a target
  - Sources and targets can be devices (e.g., keyboard, screen, printer) or files (e.g. `/home/user/myfile`)
  - Devices are treated as files (and so, I/O in C deals always with **files**)

1. Introduction

- There are 3 **standard streams** in C (typically used with the shell):

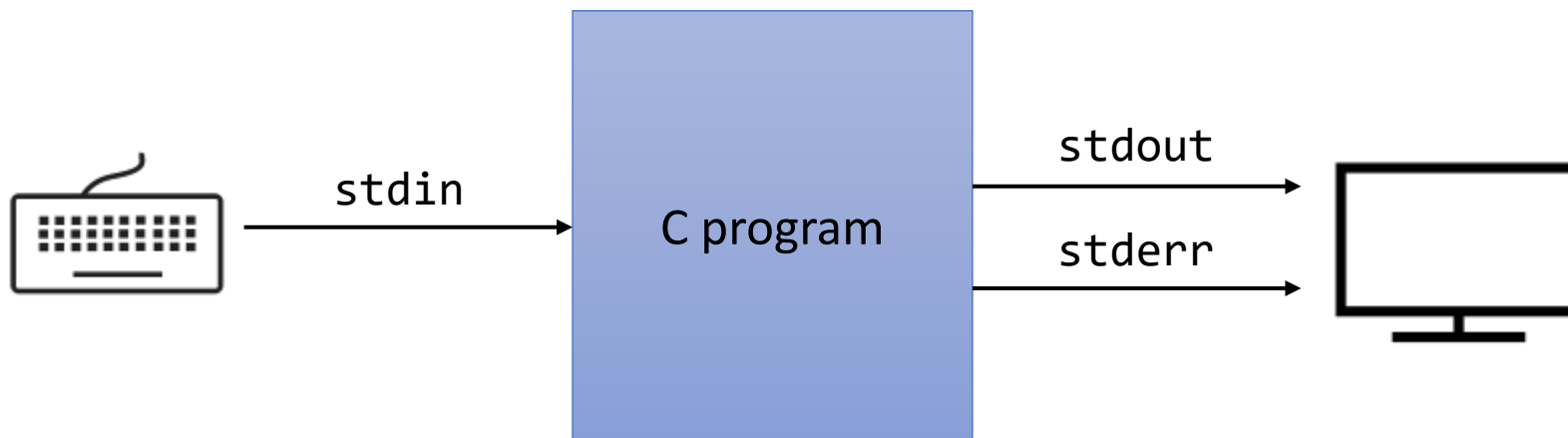| Name	File	Device	Description
Standard input	stdin	Keyboard	Default input stream (text from the keyboard typed by an user)
Standard output	stdout	Screen	Default output stream (text written to the screen to be read by an user)
Standard error	stderr	Screen	Default output stream for errors (text written to the screen to be read by an user)

Table of contents

2. Basic I/O functions - Standard output: `printf`

- The **printf** function writes a formatted string to the **standard output**
  - The `printf` function (and the rest of I/O functions we see) is defined in the **stdio.h** library
  - `printf` is a variadic function, and its prototype is as follows:

return type   function name   parameters

```
int printf(const char *format, ...);
```

Total number of characters printed (usually we do not use this return type)

The first parameter (mandatory) is the string to be written in the standard output

The following arguments (optional) are used to format the string with custom values

2. Basic I/O functions - Standard output: `printf`

- The string to be written in the standard output with **printf** can be formatter using *format specifiers* (the symbol **%** followed a character) to convert different types:

```c
#include <stdio.h>

int main() {
    char string[] = "Hello world";

    printf("%s\n", string);

    return 0;
}
```

```
Hello world
```

2. Basic I/O functions - Standard input: `scanf`

- The `scanf` function reads data from the standard input according to the format provided
  - The format specifiers used with `printf` are also used with `scanf` to specify different types (integers, strings, characters, etc.) to be read
  - The prototype of `scanf` is:

```
int scanf(const char *format, ...);
```

On success, the function returns the number of items of the argument list successfully read. If a reading error happens, then **EOF** is returned. EOF stands for "*End of File*" an it is a keyword in C reserved to determine the end of a file

The *varargs* parameters in `scanf` need to be pointers, because the changes made inside the function `scanf` are reflected in caller parameters

2. Basic I/O functions - Standard input: `scanf`

• Basic example using `scanf`:

```c
#include <stdio.h>

int main() {
    char str[40];

    printf("Enter a string: ");
    scanf("%s", str);

    printf("You entered: %s\n", str);

    return 0;
}
```

```
Enter a string: hello
You entered: hello
```

```
Enter a string: hello world
You entered: hello
```

In order to read a complete line, other functions (such as `fgets` or `readline`) are preferred

Fork me on GitHub

2. Basic I/O functions - Standard input: `scanf`

- Another example using `scanf`:

```c
#include <stdio.h>

int main() {
    char str[40];
    int i;

    printf("Enter a string and an integer: ");
    scanf("%s %d", str, &i);

    printf("You entered: %s %d\n", str, i);

    return 0;
}
```

```
Enter a string and an integer: hello 100
You entered: hello 100
```

Fork me on GitHub

2. Basic I/O functions - Standard input: `scanf`

- The `scanf` function work using arguments passed by reference:

```
int scanf(const char *format, ...);
```

The *varargs* parameters in `scanf` need to be pointers, because the changes made inside the function `scanf` are reflected in caller parameters

- For this reason, when we invoke `scanf` for basic types (e.g. `char`, `int`, etc.), we need to use the reference operator (&)

```
int i;
char str[40];

scanf("%d", &i);
scanf("%s", str);
```

Fork me on GitHub

2. Basic I/O functions - Standard input: `scanf`

- Another example using `scanf`:

```c
#include <stdio.h>

int main() {
    int i, res;

    printf("Enter an integer: ");
    res = scanf("%d", &i);

    if (res == EOF) {
        printf("You sent EOF\n");
    } else {
        printf("You entered: %d\n", i);
    }

    return 0;
}
```

```
Enter an integer: ^D
You sent EOF
```

EOF can be typed by a user in the shell using Ctrl+D in Unix-like systems

Fork me on GitHub

Table of contents

3. Other I/O functions - Handling characters: `getchar` and `putchar`

- The functions to read and write **characters** in C:

Prototype	Description
`int getchar(void);`	Read a character (only one) from the standard input and returns it as an integer
`int putchar(int c);`	Write a character (only one) to the standard output (and returns the same character)

```c
#include <stdio.h>

int main() {
    printf("Enter a character: ");
    char ch = getchar();

    printf("You entered: ");
    putchar(ch);

    return 0;
}
```

```
Enter a character: c
You entered: c
```

```
Enter a character: hello
You entered: h
```

Only a character is actually read

Fork me on GitHub

3. Other I/O functions - Handling lines: **gets** and **puts**

- Other functions to read and write **strings** are:

Prototype	Description
char *gets(char *str);	Read a string from the standard input until a terminating newline
int puts(const char *str);	Writes a string plus a newline to the standard output

```c
#include <stdio.h>
#define MAX 80

int main() {
    char str[MAX];
    printf("Enter a string: ");
    gets(str);

    printf("You entered: ");
    puts(str);

    return 0;
}
```

```
gets_and_puts.c: In function 'main':
gets_and_puts.c:7:5: warning: implicit
declaration of function 'gets'; did you mean
'fgets'? [-Wimplicit-function-declaration]
    7 |     gets(str);
      |     ^~~~
      |     fgets
/usr/bin/ld: /tmp/ccKHJUYZ.o: in function
`main':
gets_and_puts.c:(.text+0x39): warning: the
`gets' function is dangerous and should not be
used.
```

We get a warning when compiling this program since the gets function is deprecated in favor to fgets (since it protects from buffer overflow problems)

```
Enter a string: hello
You entered: hello
```

```
Enter a string: hello and bye
You entered: hello and bye
```

3. Other I/O functions - Handling lines: `fgets`

- The function **`fgets`** reads a line from the specified file and stores it into the string pointed to by a pointer
  - Prototype: `char *fgets(char *str, int n, FILE *f);`
  - It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first

> This examples reads a string from the standard input (**stdin**), and then, this string is written on the standard output

```c
#include <stdio.h>
#define MAX 80

int main() {
    char str[MAX];
    printf("Enter a string: ");
    fgets(str, MAX, stdin);

    printf("You entered: ");
    puts(str);

    return 0;
}
```

```
Enter line: Hello world!
You entered: Hello world!
```

Fork me on GitHub

3. Other I/O functions - Handling lines: `readline`

Fork me on GitHub

- Another function to read strings from a file (e.g. the input stream) is **getline**:

Prototype	Description
`int *getline(char **lineptr, size_t *n, FILE *stream);`	Reads an entire line from stream, storing the text (including the newline and a terminating null character) in a buffer

```c
#include <stdio.h>
#define MAX 80

int main() {
    char str[MAX];
    size_t bufsize = MAX;
    char *buffer = str;

    printf("Enter a string: ");
    getline(&buffer, &bufsize, stdin);

    printf("You entered: ");
    puts(str);

    return 0;
}
```

```
Enter a line: hello
6 characters were read
You entered: hello
```

getline is that is not available in some gcc compilers (e.g., in Windows)

3. Other I/O functions - Handling lines: `readline`

- **`getline`** uses a double pointer as a first argument since it dynamically allocates memory for the buffer

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

int main() {
    char *line = NULL; // Pointer to store the buffer
    size_t len = 0;    // Initial size of the buffer
    int read;          // Number of characters read

    printf("Enter a string: ");
    read = getline(&line, &len, stdin);
    printf("You entered %d characters (%ld bytes): %s\n", read, len, line);

    free(line);  // Free buffer (dynamic memory)

    return 0;
}
```

We will study dynamic memory in unit 6

Fork me on GitHub

3. Other I/O functions - Handling lines: `scanf`

- We can also use to read string lines from the user

- For that, we need a special format specifier with a regular expression (*regex*)

  - A regular expression is a sequence of characters that is used to search pattern

```c
#include <stdio.h>
#define MAX 80

int main() {
    char buffer[MAX];

    printf("Enter line: ");
    scanf("%[^\n]", buffer);

    printf("You entered: %s\n", buffer);

    return 0;
}
```

```
Enter line: hi there
You entered: hi there
```

3. Other I/O functions - Writing strings: `sprintf`

Fork me on GitHub

- The C library function **sprintf** sends formatted output to a string (pointed by **str**)

```c
int sprintf(char *str, const char *format, ...);
```

```c
#include <stdio.h>
#define MAX 80

int main() {
    int n;
    printf("Enter your age: ");
    scanf("%d", &n);

    char str[MAX];
    sprintf(str, "You are %d years old", n);
    puts(str);

    return 0;
}
```

```
Enter your age: 20
You are 20 years old
```

3. Other I/O functions - Examples

Fork me on GitHub

- The following program asks for some character to the user:

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

int main() {
    char ch;

    for (;;) { // Infinite loop
        printf("Insert character (q to exit): ");
        scanf("%c", &ch); // equivalent to: ch = getchar();

        if (tolower(ch) == 'q') {
            puts("Goodbye!");
            break;
        }

        printf("\tYou entered: %c\n", ch);
    }

    return 0;
}
```

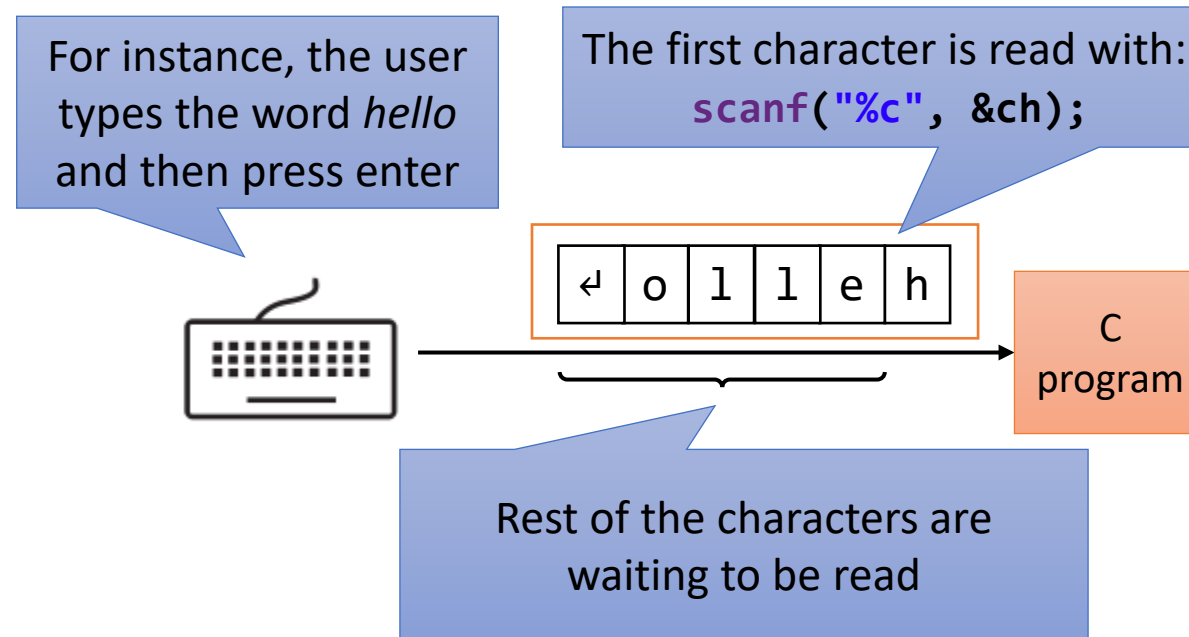> The function **tolower** converts a character to lowercase

```
Insert character (q to exit): q
Goodbye!
```

```
Insert character (q to exit): hello
        You entered: h
Insert character (q to exit):   You entered: e
Insert character (q to exit):   You entered: l
Insert character (q to exit):   You entered: l
Insert character (q to exit):   You entered: o
Insert character (q to exit):   You entered:
```

3. Other I/O functions - Examples

- When reading consecutive characters from the standard input, we need to consider that perhaps there are more characters waiting to be read in the *input buffer*:

For instance, the user types the word *hello* and then press enter

The first character is read with:
`scanf("%c", &ch);`

↵	o	l	l	e	h

C program

Rest of the characters are waiting to be read

3. Other I/O functions - Examples

- A convenient solution to this problem is to read a complete line and then get the first character:

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

#define MAX_STR 80

int main() {
    char ch;
    char input[MAX_STR];

    for (;;) { // Infinite loop
        printf("Insert character (q to exit): ");

        fgets(input, MAX_STR, stdin); // Read a complete line from the user
        input[strlen(input) - 1] = '\0'; // Remove trailing carriage return

        printf("\tYou entered: %s (%ld characters)\n", input, strlen(input));

        ch = input[0]; // Get only the first character of the input
        if (tolower(ch) == 'q' && strlen(input) == 1) {
            puts("Goodbye!");
            break;
        }

    }

    return 0;
}
```

3. Other I/O functions - Examples

- The following program asks for some number to the user:

```c
#include <stdio.h>

int main() {
    int i;

    for (;;) { // Infinite loop
        printf("Enter an option from 1 to 7 (8 for exit): ");
        scanf("%i", &i);

        if (i > 0 && i < 8) {
            printf("You entered %d\n", i);
        } else if (i == 8) {
            puts("Goodbye!");
            break;
        } else {
            puts("Wrong option");
        }
    }

    return 0;
}
```

```
Enter an option from 1 to 7 (8 for exit): 1
You entered 1
Enter an option from 1 to 7 (8 for exit): 8
Goodbye!
```

This input doesn't match the format string and scanf leaves the invalid input in the buffer

```
Enter an option from 1 to 7 (8 for exit): hello
Wrong option
Enter an option from 1 to 7 (8 for exit): Wrong option
Enter an option from 1 to 7 (8 for exit): Wrong option
Enter an option from 1 to 7 (8 for exit): Wrong option
Enter an option from 1 to 7 (8 for exit): Wrong option
Enter an option from 1 to 7 (8 for exit): Wrong option
Enter an option from 1 to 7 (8 for exit): Wrong option
...
```

3. Other I/O functions - Examples

Fork me on GitHub

- A convenient solution to this problem is to read a line and convert it to integer using the function `atoi`

> `atoi` converts a string (1st argument) to an integer (type **int**)

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

int main() {
    int i;
    int bufsize = 80;
    char buffer[bufsize];

    for (;;) { // Infinite loop
        printf("Enter an option from 1 to 7 (8 for exit): ");
        fgets(buffer, bufsize, stdin);
        i = atoi(buffer);

        if (i > 0 && i < 8) {
            printf("You entered %d\n", i);
        } else if (i == 8) {
            puts("Goodbye!");
            break;
        } else {
            puts("Wrong option");
        }
    }

    return 0;
}
```

Table of contents

4. File access

- The typical procedure to read/write text files in C is:

  1. Declare a `FILE` pointer which represents the file in C. Internally, `FILE` is an struct that contains information about the file stream:

      ```
      FILE *fp;
      ```

  2. Open the file using `fopen` (this function will return the file descriptor)

      ```
      FILE *fopen(const char *filename, const char *mode);
      ```

  These steps can be done in the same line

  3. Perform read or write operations
  4. Close the file using `fclose`:

      ```
      int fclose(FILE *fp);
      ```

4. File access - Access modes

- The following table summarizes the access modes for **text** files:

Mode	Description	Behavior
r	Open for **reading**	If the file does not exist, `fopen()` returns NULL
w	Open for **writing**	If the file exists, its contents are overwritten. If the file does not exist, it is created
a	Open for **append** (new data is added to the end of the file)	If the file does not exist, it is created
r+	Open for both **reading** and **writing**	If the file does not exist, `fopen()` returns NULL
w+	Open for both **reading** and **writing**	If the file exists, its contents are overwritten. If the file does not exist, it is created
a+	Open for both **reading** and **appending**	If the file does not exist, it is created

For **binary** files, we obtain the same behavior using the modes: rb, wb, ab, rb+, wb+, ab+

4. File access - Functions

- The following functions are used to **read** and **write** text from/to files:

Prototype	Description	
`int fgetc(FILE *fp);`	Reads and returns a single character at a time from a file. It returns EOF (end of file) when there are no more characters	read
`char *fgets(char *buf, int max, FILE *fp);`	Reads a line from the file. It stops when either (n-1) characters are read, the newline character is read, or EOF is reached	
`int fscanf(FILE *fp, const char *format,...);`	Reads formatted input from a file (same as `scanf` but from a file)	
`int fputc(int ch, FILE *fp);`	Writes a single character into a file	write
`int fputs(const char *str, FILE *fp);`	Writes a text line into a file	
`int fprintf(FILE *fp, const char *format, ...);`	Write formatted text from a file (same as `printf` but from a file)	

Fork me on GitHub

4. File access - Write text file

- Basic example for writing a text file:

The function `exit` terminates the program returning a given exit code (1 this example)

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

int main() {
    FILE *fp = fopen("file.txt", "w");
    if (fp == NULL) {
        fputs("Error opening file", stderr);
        exit(1);
    }

    // Write a line to the file
    fputs("I am writing into the file", fp);

    int i;
    printf("Enter integer: ");
    scanf("%d", &i);

    // Write another line to the file
    fprintf(fp, "You entered: %d\n", i);

    fclose(fp);

    return 0;
}
```

4. File access - Read text file

- Basic example for reading a text file line by line:

This example uses the macros **EXIT_FAILURE** (value 1) and **EXIT_SUCCESS** (value 0) defined in the standard library **stdlib.h** for the exit code

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 255

int main() {
    FILE *fp = fopen("file.txt", "r");
    if (fp == NULL) {
        fputs("Error opening file", stderr);
        exit(EXIT_FAILURE);
    }

    char buffer[MAX];
    while (fgets(buffer, MAX, fp) != NULL) {
        printf("%s", buffer);
    }

    fclose(fp);

    return EXIT_SUCCESS;
}
```

4. File access - Read formatted text file

- Basic example for reading a formatted text file:

fscanf.c

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

int main() {
    FILE *fp = fopen("data.txt", "r");
    if (fp == NULL) {
        fputs("Error opening file", stderr);
        exit(1);
    }

    char name[80];
    int age;

    while (fscanf(fp, "%s is %d years old\n", name, &age) != EOF) {
        printf("Name: %s -- Age: %d\n", name, age);
    }

    fclose(fp);

    return 0;
}
```

data.txt

```
Alice is 24 years old
Bob is 31 years old
Charles is 12 years old
```

Fork me on GitHub

4. File access - End of file: `feof`

- The function `feof` tests the EOF for the given stream

- Its prototype is as follows:

```c
int feof(FILE *fp);
```

It returns a non-zero value (i.e., *true*) when EOF is reached, else zero (i.e., *false*) is returned

4. File access - End of file: **feof**

- Basic example using feof:

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 255

int main() {
    FILE *fp = fopen("data.txt", "r");
    if (fp == NULL) {
        fputs("Error opening file", stderr);
        exit(1);
    }

    char buffer[MAX];
    while (!feof(fp)) {
        fgets(buffer, sizeof(buffer), fp);
        printf("%s", buffer);
    }

    fclose(fp);

    return 0;
}
```

The problem is the **feof** function only returns true after we tried to read past the end of the file

```
Alice is 24 years old
Bob is 31 years old
Charles is 12 years old
Charles is 12 years old
```

Possible solutions:
1. Avoid **feof** (use other conditions to check EOF)
2. Double check the output of **feof** (see example)

Fork me on GitHub

Table of contents

5. Takeaways

- In C programming, there are 3 **standard streams** (typically used in conjunction with the **shell**):
    1. Standard input (`stdin`): Messages typed from the **keyboard**
    2. Standard output (`stdout`): Messages displayed on the **screen**
    3. Standard error (`stderr`): Error messages displayed on the **screen**

- Input/Output (I/O) functions (defined in `stdio.h`) are used to:
    - To **read** data from files (or devices, also treated as files) using **input** functions (such as `scanf`, to read text from the standard input)
    - To **write** data to files (or devices) using **output** functions (such as `printf`, to write text to the standard output)