Laboratory Assignment 2: Programming in C and standard library

Table of contents

1 Objectives

In this laboratory we are going to solve several exercises oriented to strengthen our knowledge about programming in C and the use of its standard library for basic operations on strings, input, output and files.

The student is advised to create a directory for the laboratory with one subdirectory per exercise. The instructions assume that exercise N is done in a subdirectory named exercise*N* within the common directory for the laboratory.

The file files_p2.tar.gz contains a series of files that can be used as a starting point for the development of the exercises of this practice, as well as some makefiles that can be used for the compilation of the different projects.

2 Exercises

Exercise 1: File handling with the C standard library

Analyze the code of the program `show_file.c`, which reads byte by byte the contents of a file, whose name is passed as a parameter, and displays it on the screen using functions from the standard "C" library. Answer the following questions:

- Which command should be used to generate the program executable (`show_file`) by directly invoking the `gcc` compiler (without using `make`)?
- Indicate two commands to carry out respectively the compilation of the program (object file generation) and the linking of the program independently.

Make the following modifications to the program `show_file.c`:

1. Perform byte-by-byte reading of the input file using the `fread()` function instead of `getc()`.
2. Modify also the invocation to the `putc()` function by a call to `fwrite()`.

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

int main(int argc, char* argv[]) {
    FILE* file=NULL;
    int c,ret;

    if (argc!=2) {
        fprintf(stderr,"Usage: %s <file_name>\n",argv[0]);
        exit(1);
    }

    /* Open file */
    if ((file = fopen(argv[1], "r")) == NULL)
        err(2,"The input file %s could not be opened",argv[1]);

    /* Read file byte by byte */
    while ((c = getc(file)) != EOF) {
        /* Print byte to stdout */
        ret=putc((unsigned char) c, stdout);

        if (ret==EOF){
            fclose(file);
            err(3,"putc() failed!!");
        }
    }

    fclose(file);
    return 0;
}
```

Exercise 2: Writing and reading strings in files

Implement two simple programs `write_strings.c` and `read_strings.c` that allow to write and read from a file, respectively, a set of strings with variable length terminated by `'\0'`. This terminator character must be stored in the file with the rest of the characters of each string. The following standard library functions will be used to develop the two programs: `fopen()`, `fclose()`, `fread()`, `fwrite()`, `fseek()` and `malloc()`.

The `write-strings.c` program will accept as first parameter the name of a text file where the strings passed through the command line as argument 2, argument 3, etc, will be written. If the target file exists, the program will rewrite its contents.

The `read-strings.c` program will accept as parameter the name of the text file where the strings ending in `'\0'` are stored. This program will read the strings and print them on the screen separated by a line break, as shown in the following execution example:

```
## Write strings to file
usuarioso@debian:~/exercise2$ ./write_strings out London Paris Madrid Barcelona Berlin Lisbon

## Check whether file structure is correct (null-terminated strings)
usuarioso@debian:~/exercise2$ $ xxd out
00000000: 4c6f 6e64 6f6e 0050 6172 6973 004d 6164  London.Paris.Mad
00000010: 7269 6400 4261 7263 656c 6f6e 6100 4265  rid.Barcelona.Be
00000020: 726c 696e 004c 6973 626f 6e00            rlin.Lisbon.

## Read strings from file
usuarioso@debian:~/exercise2$ ./read_strings out
London
Paris
Madrid
Barcelona
Berlin
Lisbon
```

For simplicity in implementing the `read-strings.c` program, an auxiliary function `char* loadstr(FILE* input)` has to be developed. This function reads a string ending in `'\0'` from the file whose descriptor is passed as a parameter, dynamically allocating the appropriate amount of memory for the string read and returning the string. The function will first have to find out the number of characters in the string starting from the current location of the file position pointer, reading character by character. Once the terminating character is detected, it will restore the file position pointer (moving it backwards) and finally perform a read of the entire string.

Exercise 3: Handling text and binary files with the C standard library

In this exercise, a more elaborate C program will be developed that reads and writes from regular files in both text and binary formats. For its implementation, students should use at least the following functions from the standard C library: `getopt`, `printf`, `fopen`, `fclose`, `fgets`, `fscanf`, `feof`, `fprintf`, `fread`, `fwrite` and `strsep`. The manual pages of these functions should be consulted in case of doubt.

The exercise consists of 3 parts (plus optional extensions), where different features of the program will be gradually implemented:

Section A: Develop a program `student-records.c` that reads a text file with information about different students, and prints the information read in a user-friendly format on the standard output. The student text file stores a record of 4 fields per student (unique numeric identifier, NIF, first name, and last name). To simplify the *parsing* of the file, its first line contains the number of student records, and then there are the different student records, one per line, with fields separated by ":", as in the following example:

```
4
27:67659034X:Chris:Rock
34:78675903J:Antonio:Banderas
3:58943056J:Santiago:Segura
4:6345239G:Penelope:Cruz
```

The example file stores 4 student records, where the information for the first student is as follows:

- Student ID: `27`
- NIF: `67659034X`
- First name: `Chris`
- Last name: `Rock`

To read a student text file and print its contents in user-friendly format, the `student-records` program must be invoked by specifying the `-i` (*input*) and `-p` (*print*) options simultaneously on the command line, where the `-i` option accepts a parameter indicating the path to the text file. So for example, assuming that there is a file *students-db.txt* in the current directory containing the example text shown above, running the program will produce the following output:

```
usuarioso@debian:~/exercise3$ ./student-records -i students-db.txt -p
[Entry #0]
        student_id=27
        NIF=67659034X
        first_name=Chris
        last_name=Rock
[Entry #1]
        student_id=34
        NIF=78675903J
        first_name=Antonio
        last_name=Banderas
[Entry #2]
        student_id=3
        NIF=58943056J
        first_name=Santiago
        last_name=Segura
```

```
[Entry #3]
        student_id=4
        NIF=6345239G
        first_name=Penelope
        last_name=Cruz
```

For simplicity of implementation, each record will be printed on the standard output as soon as the text line of the input file corresponding to that record is processed. In this way it will not be necessary to make use of dynamic memory. For the storage in memory of the different fields of the record, the use of the `student_t` structure, defined in the `defs.h` header file provided with the lab, is recommended.

In addition to the above options, an `-h` (*help*) option will be implemented, which prints the list of options supported by the program:

```
usuarioso@debian:~/exercise3$ ./student-records -h
Usage: ./student-records [ -h | -p | -i file ]
```

The output generated by this option should be modified as additional options are implemented in the program, corresponding to the next sections.

Section B: Extend the functionality of the `student-records` program by implementing a new `-o` option. This option will allow to generate a binary version of the student text file, and to dump it into a binary output file whose path will be passed as a parameter to the `-o` option. When reading each entry in the text file, the program will store the student information in a record represented by the following data type:

```
#define MAX_CHARS_NIF  9

typedef struct {
    int student_id;
    char NIF[MAX_CHARS_NIF+1];
    char* first_name;
    char* last_name;
} student_t;
```

The generated binary file will have the following structure. The first 4 bytes of the file (int) will store the number of student records. Then the data of each student record will be written, one after the other, storing for each of them their student ID (4 bytes integer), their NIF, their first name and last name (in this order). For all the character strings to be written in the file, the terminator character will also be stored, which is the key to be able to read the file later (section C of the exercise).

In the following example, the `xxd` command is used to display the contents of the generated file (note that this file cannot be printed successfully with `cat`, as it is not a text file):

```
## Check usage
usuarioso@debian:~/exercise3$ ./student-records -h
Usage: ./student-records [ -h | -p | -i file | -o <output_file> ]

## Generate binary file
usuarioso@debian:~/exercise2$ ./student-records -i students-db.txt -o students-db.bin
4 student records writen successfully to binary file students-db.bin

## Check file contents
usuarioso@debian:~/exercise2$ xxd students-db.bin
00000000: 0400 0000 1b00 0000 3637 3635 3930 3334  ........67659034
00000010: 5800 4368 7269 7300 526f 636b 0022 0000  X.Chris.Rock."..
00000020: 0037 3836 3735 3930 334a 0041 6e74 6f6e  .78675903J.Anton
00000030: 696f 0042 616e 6465 7261 7300 0300 0000  io.Banderas.....
00000040: 3538 3934 3330 3536 4a00 5361 6e74 6961  58943056J.Santia
00000050: 676f 0053 6567 7572 6100 0400 0000 3633  go.Segura.....63
```

```
00000060: 3435 3233 3947 0050 656e 656c 6f70 6500  45239G.Penelope.
00000070: 4372 757a 00                             Cruz.
```

Section C: Implement a new `-b` (*binary*) option in the program that allows you to print the contents of an existing binary student file using the same output format as specified in Section A of the exercise. By specifying the `-b` option (instead of `-p`) on the command line, the program will assume that the format of the input file will be binary rather than text.

The following example illustrates the functionality to be implemented in this section:

```
## Check usage
usuarioso@debian:~/exercise3$ ./student-records -h
Usage: ./student-records [ -h | -p | -i file | -o <output_file> | -b ]

## Dump contents of binary file
usuarioso@debian:~/exercise2$ ./student-records -i students-db.bin -b
[Entry #0]
        student_id=27
        NIF=67659034X
        first_name=Chris
        last_name=Rock
[Entry #1]
        student_id=34
        NIF=78675903J
        first_name=Antonio
        last_name=Banderas
[Entry #2]
        student_id=3
        NIF=58943056J
        first_name=Santiago
        last_name=Segura
[Entry #3]
        student_id=4
        NIF=6345239G
        first_name=Penelope
        last_name=Cruz
```

Hint: It is recommended to reuse the `loadstr()` function implemented in exercise 2.

Optional parts

Subsection 1: The mandatory sections of the practice assume that the program processes the records read from the input file one by one, either by writing them on the standard output (options `-p` and `-b`) or by dumping each record read into the output file (option `-o`) as it is read. In this optional part we propose to refactor the `student_records.c` code program in such a way that the records from the input file–whether in binary or text format–are all read consecutively and stored in a vector of records of type `student_t`. To do so, 2 auxiliary functions have to be defined:

- `student_t* read_student_text_file(FILE* students, int* nr_entries)`

  This function reads all the information from a text file of student records already opened, and returns both the number of records in the file (return parameter `nr_entries`), and the array of student records (return value of the function). The memory of the returned array must be reserved with `malloc()` within the function itself.

- `student_t* read_student_binary_file(FILE* students, int* nr_entries)`

  Same as the `read_student_text_file()` function but reading a binary file of student records.

To complete this optional part, the rest of the implemented functions have to be rewritten to use the new functionality provided by these functions. So for example, the function that previously read from the input text file, and printed the records one by one to the standard output, must now read all the records at once using `read_student_text_file()`, and then print the records stored in the returned vector using a loop.

Subsection 2: Extend the functionality of the `student_records.c` program with a new `-a` option to add extra student records to the end of an existing file, either in binary or text format. The new records will be specified in text mode on the command line, with fields separated by `:`, and where the records will be separated by spaces. The program will automatically infer the format of the existing file based on its extension (“.txt”: text; “.bin”: binary).

Example:

```
## Check usage
usuarioso@debian:~/exercise3$ ./student-records -h
Usage: ./student-records [ -h | -p | -i file | -o <output_file> | -b | -a ]

## Add two new records
usuarioso@debian:~/exercise3$ ./student-records -i students-db.txt -a \
> 23:43159076B:Michael:Jordan 30:34651129G:Stephen:Curry
2 records written succesfully to existing text file

## Check contents
usuarioso@debian:~/exercise3$ cat students-db.txt
6
27:67659034X:Chris:Rock
34:78675903J:Antonio:Banderas
3:58943056J:Santiago:Segura
4:6345239G:Penelope:Cruz
23:43159076B:Michael:Jordan
30:34651129G:Stephen:Curry
```

Note that in this case the -a option does not accept any arguments, but the records are provided as extra arguments on the command line (without associated option). To process these arguments, the global variable `optind` of `getopt()` has to be used, which at the end of the option processing stores the index of the first extra argument provided. In the command example shown above, `optind` will be worth 4 at the end of the option processing loop, since the first extra argument constitutes the *token* number 4 on the command line. Thus the C expression `&argv[optind]` can be used to access the extra argument vector, having the following contents in the example: `{"23:43159076B:Michael:Jordan", "30:34651129G:Stephen:Curry", NULL}`