Ninth round tasks as a `ZIP file.`

Additional I/O stream handling functions¶

This section deals with few additional I/O stream handling functions.

- ftell - `long ftell(FILE *stream)` tells the current position in the stream, as bytes from the beginning of the stream. As data is being read or written from the stream, the position indicator moves forward.
- fseek - `int fseek(FILE *stream, long offset, int whence)` sets the file position indicator to the given position (**offset**), counted as bytes. If **whence** is SEEK_SET, the position is counted as distance from the beginning of file, when it is SEEK_END, the position is relative to the end of the file. Setting the position works on files, but may not work on some other types of streams (for example when accessing terminal with stdin or stdout).
- fprintf - `int fprintf(FILE *stream, const char *format, ...)` works similarly to the **printf** function, but takes one additional parameter, stream, and produces the formatted output to the given file instead of the standard output stream that is typically shown on the screen. The function returns the number of characters printed if writing was successful, or negative value if there was an error.
- fscanf - `int fscanf(FILE *stream, const char *format, ...)` is similar to the **scanf** function, except that it tries to read the input from file stream instead of standard input. The function returns the number of fields read, or EOF if there was an error or end of file was reached before the specified fields could be read.
- feof - `int feof(FILE *stream)` returns non-zero if the file is at the end (and no more reading can be done), or zero if the file is not yet at the end. **Note**: the end-of-file state is only set after an attempt to read "past" the end of file. Therefore, if you have read all content of the file, but have not tried to read any further, feof still returns 0.
- ferror - `int ferror(FILE *stream)` returns non-zero if an error has occurred in an earlier I/O operation, or zero if no error has occurred.
- fflush - `int fflush(FILE *stream)` flushes the buffered data in output stream buffer. Returns 0 on success or -1 on failure.

For standard input and standard output there are pre-defined streams **stdin** and **stdout**. For example, calling

```
fprintf(stdout, "%d\n", an_int)
```

is equivalent to calling **printf** with the same format specifiers and parameters. In addition there is a third stream that is open by default, called **stderr** that is conventionally used for printing error outputs from programs.

A text file can be read as follows. The program reads file "test.c" line by line, and shows each line on the standard output. It also uses the standard error stream for error messages.

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

int main(void) {
    FILE *f;
    char buffer[100];

    f = fopen("testfile", "r"); // open file for reading
    if (!f) {
        fprintf(stderr, "Opening file failed\n");
        return EXIT_FAILURE;
    }
    while (fgets(buffer, sizeof(buffer), f) != NULL) {
        if (fputs(buffer, stdout) == EOF) {
            fprintf(stderr, "Error writing to stdout\n");
            fclose(f);
            return EXIT_FAILURE;
        }
    }
    fclose(f);
}
```

For **binary files**, one should use **fread** and **fwrite** for reading and writing operations. These functions do not have any special treatment on NULL characters or newlines. Below is an example of binary write of an integer array of 10 numbers, followed by reading the array from disk. The example demonstrates also the use of **feof** and **ferror** indicators.

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

int main(void)
{
    int numbers[10] = { 1, 0, -2, 3, 10, 4, 3, 2, 3, 9 };
    FILE *fp = fopen("intarray", "w");
    if (!fp) {
        fprintf(stderr, "Could not open file\n");
        return EXIT_FAILURE;
    }
    size_t n = fwrite(numbers, sizeof(int), 10, fp);
    if (ferror(fp)) {
        fprintf(stderr, "Error occurred\n");
        return EXIT_FAILURE;
    }
    fprintf(stdout, "%lu items written\n", n);  // same as printf
    fclose(fp);

    // re-open file for reading, and read the integers
    fp = fopen("intarray", "r");
    int *num2 = malloc(10 * sizeof(int));
    n = fread(num2, sizeof(int), 10, fp);

    // feof indicator should not be set yet, because we did not read
    // past the end of file
    if (feof(fp)) {
        fprintf(stderr, "prematurely reached end of file\n");
        return EXIT_FAILURE;
    } else if (ferror(fp)) {
        fprintf(stderr, "error occurred\n");
        return EXIT_FAILURE;
    }
    fprintf(stdout, "%lu items read\n", n);

    // should not read anything, because we should be at the end of file
    n = fread(num2, sizeof(int), 10, fp);
    if (feof(fp)) {
        fprintf(stdout, "%lu items read, EOF indicator is set\n", n);
    }

    fclose(fp);
    free(num2);
    return EXIT_SUCCESS;
}
```

This code creates a file of 40 bytes (10 integers of 32 bits each). This is a binary file that cannot be understood by text editor, but **hexdump** shows the file content as follows:

```
$ ./a.out
10 items written
10 items read
0 items read, EOF indicator is set

$ hexdump -C intarray
00000000  01 00 00 00 00 00 00 00  fe ff ff ff 03 00 00 00  |................|
00000010  04 00 00 00 00 00 00 00  03 00 00 00 02 00 00 00  |................|
00000020  03 00 00 00 09 00 00 00                           |........|
00000028
```

Note that each integer takes four bytes, in little-endian byte order.

Task 9.2: File basics¶

Objective: Practice basic file reading.

The first simple task is to open a file and print it to the standard output. Other task is to implement a basic diff-tool that tells the first line is different within two files.

a) Print a file

Implement function `int print_file_and_count(const char *filename)` that prints the file into the standard output. Function should return the number of characters printed. If file opening fails, function should return -1.

b) Difference

Implement function `char *difference(const char* file1, const char* file2)` that compares two files. The function should return the first lines that differ in the two files, concatenated together, separated by four dashes, see example below. The returned string should be dynamically allocated. (NB! The newline character is a part of the line that comes before it.) If the files are equal, NULL is returned. Function stops immediately, if either one of the files end and returns NULL.

File 1:

```
#include <stdio.h>

int main(void) {
    printf("Hello world!\n");

    return 0;
}
```

File 2:

```
#include <stdio.h>

int main(void) {
    printf("Hello world\n");

    return 0;
}
```

Function returns:

```
        printf("Hello world!\n");
----
        printf("Hello world\n");
```


Task 9.3: Statistics¶

Implement functions to calculate the following metrics from a given file:

(a) Line count

Implement function `int line_count(const char *filename)` that calculates the number of lines in the given file, and returns the line count. If there is an error, the function should return -1. Empty file is considered to have no lines. A newline character is a part of the row which comes before the newline. If the last line of the file is not empty, it should be counted as a line even if it does not end in newline character.

(b) Word count

Implement function `int word_count(const char *filename)` that calculates the number of words in a given file and returns the word count. In this exercise we define word like this: Word is a substring that contains at least one **alphabetic** character (**isalpha** returns nonzero value in this case). Two words are separated by one or more whitespace characters (**isspace** returns nonzero value). If there is an error, the function should return -1. (Note that shell command 'wc -w' defines a "word" differently, and cannot be used to compare results with this function)

Task 9.4: Shop¶

In this task you should implement an imaginary shop's bookkeeping system in two different ways. The products of the shop are given in a *Product* struct array. In one *product* element, there exists the product's name, price and the quantity of products in stock (see the header file for struct implementation). As said before, the accounting is done by an array of these *Product* elements. **The last element of the array has a name whose first character is the terminating null character ('\0').** You should implement four functions:

(a) function *write_binary* which outputs a binary format file from given Product array with parameter name *shop*. Another parameter for this function is *filename*, the name of the desired output file. If the function succeeded, it should return 0, otherwise it should return 1.

(b) function *read_binary*, which reads a binary format file (written in section a) and outputs the read data into a *Product*-array. After reading the data the function should return a pointer to this array. **The function must allocate the memory for this array** and the reading format should be so that the first array element read from the binary file should be in the first index of the list. Also remember to mark the last element of the list as described before. If the function does not succeed, it should return a *NULL* pointer.

(c) function *write_plaintext*, which outputs a plaintext format (ie. human readable) from the given *Product* array *shop*. The file format should be following:

```
yoghurt 1.2 23
muesli 4.3 12
```

As seen the data items of the struct are separated with a space, and the array elements are separated by a newline. Because the data items are separated by a space, the product name should not have spaces in it. Into this plaintext product file the last element (with the name of null character) should not be printed at all. As in the a-case, f the function succeeded, it should return 0, otherwise it should return 1.

(d) function *read_plaintext* which reads a plaintext file (written in section c) and outputs the read data into a *Product*-array. After reading the data the function should return a pointer to this array. **The function must allocate the memory for this array** and the reading format should be so that the first array element read from the binary file should be in the first index of the list. Also remember to mark the last element of the list as described in the very beginning of this task. If the function does not succeed, it should return a *NULL* pointer.

Hints:

- It may be easier to first implement the functions in parts c and d.
- When reading plaintext data, you should use the `sscanf` function.
- When dealing with binary files, use functions fread and fwrite. Use one fread/fwrite command for one *Product*.
- Do **not** write the last "null element" into the files. However, you have to remember to add the "null element" yourself to the end of an array.
- When dealing with plaintext files, do **not** add any format specifications, i.e. read a floating point with `%f`, not with `%0.1f`.

It is encouraged to make your own test files to test the function you are implementing. Implement your functions into the file *shop.c* based on the definitions from the *aaltoshop.h*-file.

The files for this task:

- **main.c** for testing
- **aaltoshop.h** which has the needed definitions and declarations.