File Input/Output

Raul P. Pelaez

October 1, 2025

Contents

1 Introduction

Thus far, our programs have lived in a kind of walled garden, in which the only interaction with the outside world comes from printing text to the terminal. Useful programs need to be able to interact with their environment, which they can do in many ways, for instance:

- Asking the user for input from the terminal (with `scanf`)
- Reading and writing files
- Communicating with other programs or computers (pipes, sockets, etc)
- Interacting with hardware (sensors, actuators, etc)
- Using system calls to interact with the operating system (processes, memory, etc)

In future lessons, we will learn about sockets (networking). Today, we will focus on file input/output (I/O), which is one of the most common ways for programs to interact with their environment. Files are specially important in an UNIX-like environment (such as Linux, OSX or WSL) because in UNIX *everything* is a *file*: devices, processes, network connections, etc. Writing to a file is conceptually (and programatically) identical in UNIX to sending data to a printer, a network socket, or a process.

2 File I/O

C offers two APIs for file I/O: the low-level POSIX API, which is a thin wrapper around system calls, and the higher-level C standard library API, which is built on top of the POSIX API. The C standard library API is easier to use and more portable, so we will focus on it here. The POSIX API is cool, but it is not part of the C standard, so it is not guaranteed to be available on all systems (e.g. Windows). Thus, you should try to use the C standard library API whenever possible.

2.1 The IO standard library

The C standard library provides a set of functions for file I/O in the `stdio.h` header. The main data structure for file I/O is the `FILE` object, which represents an open file. You can think of a `FILE` object as a kind of "file handle" that you can use to read from or write to a file.

The main functions we will use are:

- `fopen`: opens a file and returns a pointer to a `FILE` object
- `fclose`: closes a file
- `fprintf`: writes formatted data to a file
- `fscanf`: reads formatted data from a file
- `fgetc`: reads a single character from a file
- `fgets`: reads a line of text from a file
- `fputs`: writes a line of text to a file

Advice

RTFM
Remember to read the manual pages for these functions (e.g. `man fopen` in the terminal) to understand their usage and parameters. The C standard library is very well documented, and the manual pages are a great resource for learning about the functions available to you.

Note that the functions to interact with files are similar to the functions we have used to interact with the terminal (e.g. `printf`, `scanf`). This is because the terminal is just a special kind of file (the standard input/output files, `stdin`, `stdout`, and `stderr`). Note that functions dealing with files are usually prefixed with an 'f' (for file), e.g. `fprintf`, `fscanf`, etc. Most functions are straightforward to use by translating from what you know about terminal I/O. For instance, `fprintf(file, "Hello, World!\n")` writes "Hello, World!" to the file represented by the `FILE` object `file`, just like `printf("Hello, World!\n")` writes to the terminal.

2.1.1 Opening and closing files

To open a file, you use the `fopen` function, which takes two arguments: the name of the file to open, and a string representing the mode in which to open the file. The mode can be:

- "r": open for reading (the file must exist)
- "w": open for writing (creates the file if it does not exist, or truncates it if it does)
- "a": open for appending (creates the file if it does not exist, writes are added to the end of the file)
- "r+": open for reading and writing (the file must exist)
- "w+": open for reading and writing (creates the file if it does not exist, or truncates it if it does)
- "a+": open for reading and appending (creates the file if it does not exist, writes are added to the end of the file)

Example

Here is a simple example that opens a file, writes some data to it, and then reads the data back:

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

int main() {
  // Open a file for writing
  FILE *file = fopen("example.txt", "w");
  if (file == NULL) { // Always check for errors!
    perror("Error opening file for writing");
    return EXIT_FAILURE;
  }

  // Write some data to the file
  fprintf(file, "Hello, World!\n");
  fprintf(file, "This is a test file.\n");

  // Close the file
  fclose(file);

  // Open the file for reading
  file = fopen("example.txt", "r");
  if (file == NULL) {
```

```c
    perror("Error opening file for reading");
    return EXIT_FAILURE;
  }

  // Read and print the data from the file
  char buffer[256];
  while (fgets(buffer, sizeof(buffer), file) != NULL) {
    printf("%s", buffer);
  }
  // Close the file
  fclose(file);

  // Remove the file
  // Ignore the potential error if this fails, just let the program end
  remove("example.txt");
  return EXIT_SUCCESS;
}
```

Output

```
Hello, World!
This is a test file.
```

Insight

Comments in code

Remember that in these examples, I am adding comments in a didactic manner. You will notice I sometimes write code that goes like:

- An empty line
- A redundant comment
- An actual line of code

Such as:

```c
// Close the file
fclose(file);
```

In real life this is a terrible practice. Comments should be used to explain *why* something is done, not *what* is done. The code should be self-explanatory as to what it does. Comments should be used to explain the reasoning behind a particular implementation, or to provide context that is not immediately obvious from the code itself.

In this regard, the only meaningful comment in the example above is the last one, which explains that we are ignoring the potential error from `remove`. The rest of the comments are redundant and should be removed in real code.

Beware, for ChatGPT **LOVES** to use this terrible "template" constantly.

2.2 Error handling

Error handling is very important when dealing with files, as many things can go wrong (e.g. the file does not exist, you do not have permission to read/write the file, etc). Most file I/O functions return a special value (usually `NULL` or `EOF`) to indicate an error. You should always check for these error values and handle them appropriately (e.g. by printing an error message and exiting the program). In the example above, we check the return value of `fopen` to see if it is `NULL`, which indicates that the file could not be opened. If this happens, we use the `perror` function to print an error message to the terminal, and then exit the program with a failure status. The file API also makes use of the global variable `errno`, which is set to indicate the specific error that occurred. You can use the `strerror` function to get a human-readable string describing the error.

Example

Here is an example that demonstrates error handling when opening a file:

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

int main() {
    // Try to open a non-existent file for reading
    FILE *file = fopen("non_existent_file.txt", "r");
    if (file == NULL) {
    // Print an error message
    printf("Error opening file: %s\n", strerror(errno));
    return EXIT_FAILURE;
    }
    // Close the file (this line will not be reached in this example)
    fclose(file);
    return EXIT_SUCCESS;
}
```

Output

```
Error opening file: No such file or directory
```

Other functions have different ways of indicating errors, but at this point in the course, you will be familiar any error handling scheme. Some of them are a bit convoluted. For instance, `fgets` returns `NULL` on error or end-of-file, but you cannot tell which one happened. To check for errors, you need to use the `ferror` function, and to check for end-of-file, you need to use the `feof` function.

2.3 The POSIX API

The POSIX API is a lower-level API for file I/O that is built on top of system calls. It is not part of the C standard, but it is widely available on UNIX-like systems (Linux, OSX, etc). The POSIX API is more powerful and flexible than the C standard library API, but it is also more complex and harder to use. The main functions we will use are:

- `open`: opens a file and returns a file descriptor (an integer)
- `close`: closes a file descriptor
- `read`: reads data from a file descriptor
- `write`: writes data to a file descriptor
- `lseek`: moves the file pointer to a specific position in the file
- `fcntl`: performs various operations on a file descriptor (e.g. setting flags, locking, etc)

Example

Here is a simple example that opens a file, writes some data to it, and then reads the data back using the POSIX API:

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
int main() {
  int fd = open("example_posix.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  if (fd == -1) {
    printf("Error opening file for writing: %s\n", strerror(errno));
    return EXIT_FAILURE;
  }
  const char *data = "Hello, World!\nThis is a test file using POSIX API.\n";
  ssize_t bytes_written = write(fd, data, strlen(data));
  if (bytes_written == -1) {
    printf("Error writing to file: %s\n", strerror(errno));
```

```
    close(fd);
    return EXIT_FAILURE;
  }
  if (close(fd) == -1) {
    printf("Error closing file: %s\n", strerror(errno));
    return EXIT_FAILURE;
  }
  fd = open("example_posix.txt", O_RDONLY);
  if (fd == -1) {
    printf("Error opening file for reading: %s\n", strerror(errno));
    return EXIT_FAILURE;
  }
  char buffer[256];
  ssize_t bytes_read;
  while ((bytes_read = read(fd, buffer, sizeof(buffer) - 1)) > 0) {
    buffer[bytes_read] = '\0'; // Null-terminate the string
    printf("%s", buffer);
  }
  if (bytes_read == -1) {
    printf("Error reading from file: %s\n", strerror(errno));
    close(fd);
    return EXIT_FAILURE;
  }
  if (close(fd) == -1) {
    printf("Error closing file: %s\n", strerror(errno));
    return EXIT_FAILURE;
  }
  remove("example_posix.txt");
  return EXIT_SUCCESS;
}
```

Output

```
Hello, World!
This is a test file using POSIX API.
```

It is important that you understand this API. In UNIX, there is no difference between the file descriptor you get from `open` and the one that you get from `socket` when creating a network connection. This means that you can use the same functions (`read`, `write`, etc) to read and write data to files, network connections, pipes, etc.

3 Exercises

3.1 Zipf's Law

Goal

For a not very well known reason, when a list of measured values is sorted in decreasing order, the value of the nth entry is often approximately inversely proportional to n. This is known as Zipf's law. The best known instance of Zipf's law applies to the frequency table of words in a text or corpus of natural language

$$\text{word frequency} \propto \frac{1}{\text{word rank}}$$

In other words, it is usually found that the most common word occurs approximately twice as often as the next common one, three times as often as the third most common, and so on. So, if we compute, for a given text, the frequency of each word and sort them in decreasing order, we should see a straight descending line in a log-log plot when plotting against the rank. The rank is just the position of the word in the sorted list of words.
To showcase this law, we will be investigating the book Moby Dick; Or, The Whale by Herman Melville. Take the txt containing the book from BB and put it in your current folder.

Milestone

Reading the text
Write a function that takes the filename as argument and returns an `char *` containing only the alphabetic characters in the file. Replace any non alphabetic characters with a space. You can use this as a skeleton.

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

/**
  * @brief Reads the contents of the given file into a continuous array,
  * any non-alphabetic character (numbers, symbols, ...) are replaced with a space.
  * @param fileName The name of the file to read
  * @param contents A pointer that will be allocated by the function and
  contain the contents of the book.
**/
void readBook(const char* fileName, char** contents);
```

hint

- Use the `isalpha` and `isspace` functions.
- You can read the file character by character using `fgetc` or line by line using `fgets`.
- You will need to allocate memory for the contents of the book, you can use `malloc` and `realloc` to do this. Remember to free the memory when you are done with it.

Milestone

Counting the words

Write a new function that takes the characters in the book and returns two arrays, one has the unique words in the book, and the other has the frequency of each word. It should have this signature:

```
/**
  * @brief Counts the frequency of each word in the book.
  *        The caller is responsible for freeing the allocated memory.
  * @param book A continuous array of characters containing the book.
  * @param words A pointer to an array of strings that will be allocated
                 by the function and contain the unique words in the book.
  * @param frequencies A pointer to an array of integers that will be allocated
            by the function and contain the frequency of each word in the book.
  * @param nWords A pointer to an integer that will be
            set to the number of unique words in the book.
  **/
void countWords(const char* book, char*** words, int** frequencies, int* nWords);
```

- Start by splitting the book into words, using the fact that words are separated by spaces.
- Sort the words in a way that places identical words next to each other.
- Count the number of unique words and their frequencies. You will need to allocate and use `realloc`. Use the fact that the words are sorted to make this easier.

The function should allocate the pointers provided as arguments, so the caller is responsible for freeing them.

hint

- You can use the `strtok` function to split the book into words.
- You can use the `qsort` function to sort the words.
- Use a function that compares two strings, you can use `strcmp` for this.
- `strcmp` returns 0 if the strings are equal, a negative value if the first string is less than the second, and a positive value if the first string is greater than the second. "Less" and "greater" are defined in terms of lexicographical order (dictionary order).

Insight

A map (dictionary in Python) trivializes this task, but we do not have such commodities in C.

Milestone

Clean it up

Some words will be misrepresented in your result, such as "The" and "the" being counted as different words. Modify your code to convert all words to lowercase before counting them. Additionally, there might be some spurious entries too, such as empty strings or broken words, do your best to identify and remove them.

Milestone

Plotting the results
Write a function that outputs the word frequencies to a file in the format "rank freq word" for each line. Use some external tool to plot this file in log-log scale.

hint

- The rank is just the position of the word in the sorted list of words, you can use a counter for this.
- Remember the most used word should have rank 1, not 0, for it to be visible in a log-log plot.

Advanced milestone

Plot directly from C using some external library.

hint

- Do not underestimate this one. It will give you immense experience, but it will come with a price.
- Check `PLPlot`, you can install it with `conda` and integrate it into CMake using `find_package(PLPlot REQUIRED)` and linking with `plplot`.

Milestone

The Hápax legómenon
In a corpus of text, a hapax legomenon is a word that occurs only once. Print to terminal the number of words that occur only once and some of them.

Milestone

From demo to real program
Turn your code into a real program that takes the input filename and output filename as command line arguments. The program should be used like this:

```
./zipf input.txt output.dat
```

Use the `argc` and `argv` parameters of the `main` function to get the command line arguments. You will find the input filename in `argv[1]` and the output filename in `argv[2]`. Take care of error handling, for instance, if the user does not provide enough arguments, print a usage message and exit.

Advanced milestone

Make it installable
Turn your program into a real installable program you can run from anywhere using CMake. The program should be installed to your conda environment's `bin` folder.

hint

- When configuring, call CMake with: `cmake -DCMAKE_INSTALL_PREFIX=$CONDA_PREFIX -B build`
- Add the rule `install(TARGETS zipf DESTINATION bin)` command in your CMakeLists.txt to install the program to the `bin` folder of your conda environment.
- After building, call `cmake --install build` to install the program.
- You can then run the program from anywhere (as long as the `conda` env is activated) using: `zipf input.txt output.dat`