Report on the Reference Solution to the A0 CompSys Assignment

Troels Henriksen (athas@sigill.dk)

September 18, 2019

Commented by Michael Kirkedal Thomsen. Do not copy-paste from this without explicit reference.

1 Introduction

This report documents the reference solution of the A0 assignment. It is intended not just to explain the reference solution, but also as an example of a report with good structure and content. You may use it as inspiration for reports you write later during this and other courses. However, do not follow it slavishly: different problems call for different solutions, and different people have different style. If you prefer another style, then stick to it. An idiosyncratic report executed well is superior to a idiomatic report executed poorly.

The task was to implement a small clone of the Unix tool `file(1)`. I have not found reason to stray from the functional requirements outlined in the report, and it is to my knowledge implemented completely. For the pedantic, Section 4 notes some theoretical deficiencies.

Start with a short descriptive introduction. You are writing for someone else, so the introduction must give the reader an idea of what to expect in the rest of the report.
Always consider who you are writing for. For reports at DIKU you are always writing to explaing your work for other students at your level. Thus, a student should be able to read first the assignment text and then your report to get the full idea of a solution.

Also tell the reader where to find things; use specific reference where relevant. You are writing a technical report (not fiction), so no need to surprise the reader with plot twists.

2 Implementation

The core of the program is `file_stream`, a simple trinary state machine that loops through each byte in the file, starting from the first. The state indicates the current idea of the file type: `EMPTY`, `DATA`, or `ASCII`, modeled with a C enum `file_type`. Initially the state is `EMPTY`. If the next byte is an ASCII character (as defined by the assignment text and implemented as `is_ascii()`), then the state transitions to `ASCII`. Otherwise, the state becomes `DATA` and the loop ends. The final state then indicates the file type. The advantage of this scheme is that we stop as soon as we are certain of the file type.

Much of the program is concerned with error checking. All functions that return an error code are checked, and any errors propagated. If another error occurs while printing the file type or an error message, then the program immediately terminates with `EXIT_FAILURE`.

Unless some background knowledge is needed (only if you use things not taught on the course), start by explaining your work. This should *not* be a direct description of our assignment text; the reader knows this. Instead describe how you read the problem and how you have decided to solve it. For these reports this should also aid understanding your code, without talking about specifics. Thus, after reading this, I can more easily read your code.

3 Testing

I test by extending the handed-out `test.sh` script, which compares the results of my `file` with the system-installed `file(1)`, on a corpus of sample files. Interesting cases include:

- An empty file.
- A file that is ASCII until the last byte.
- A file that is ASCII except the first byte.
- ASCII files that contain some of the special characters required in the text (edge cases).

Tests are to show what your implementation can and cannot do. Like any other scientific experiment tests must be reproducible. Start by explaining the test methodology and how the reader can run them.

Explain the different test cases that you have; with more tests you can group them together. This must also show the reader test coverage. For A0 it is very easy to see based on the below examples, but for larger test suites you would need to argument this explicitly.

3.1 Test Limitations

The automated tests only cover the *happy path*, not any error cases. I have tested trivial cases (e.g. incorrect number of command line parameters) by hand, but some cases (e.g. I/O error in the middle of a file) are nontrivial to test at all.

End by describing what your tests cannot show. This shows that you understand limitations of the methodology and coverage.

Here we only have positive tests, but in some cases negative tests can be used to show what your implementation cannot do.

4 Limitations and Potential Problems

If an error occurs while reading bytes from the file, `file_path` will return without closing the file with `fclose()`. However, as the program will terminate soon after, leaking this `FILE` object is not harmful. This could potentially be a problem if `file_path` were to one day be used in another setting.

Explain what your implementation cannot and what problems there might be. This should of course be in relation to the assignment text. E.g. writing that the implementation cannot solve the *halting problem* is not useful.

5 Conclusions

I have implemented a version of `file` that can check for empty file and ASCII encoding. All parts of the API specified in the assignment text are supported. I judge the implementation to be of adequate quality; I for example do not use magic numbers. The tests verify correct operation only in the happy path.

End with a conclusion that (1) explains to what extend you have solved the problem and (2) sums up highlights of the report. This is your final chance to have the readers attention and point to the important parts of your report.