A0: `file(1)` — Unicode

Computer Systems 2024-25
Department of Computer Science
University of Copenhagen

Oleks Shturmov <oleks@oleks.info> and
Michael Kirkedal Thomsen <m.kirkedal@di.ku.dk>

Due: Sunday, September 22, 16:00
Version 1.05

This is the first assignment in the course Computer Systems at DIKU. We encourage pair programming, so please form groups of 2 or 3 students. Groups cannot be larger than 3, and we strongly recommend that you do not work alone.

For this assignment you will receive a mark out of 4 points; You must attain at least half of the possible points to be admitted to the exam. For details, see the *Course description* in the course page. This assignment does not belong to a category. Resubmission is not possible.

---

1 Introduction

> *The nice thing about standards is that you have so many to choose from.*
> — Andrew S. Tanenbaum, *Computer Networks, 2nd ed.* (1988)

The purpose of this assignment is to get you started with C, and a couple command-line utilities common to a Unix-like development environment. What better way to do that, than try to implement one of these yourself?

`file(1)`[1] is a classical Unix-like command-line utility for guessing the type of a file. In a Unix-like operating system, it is often the contents, or the metadata of a file, not a filename extension, that determines the "type" of a file. Standard `file(1)` uses clever heuristics to guess the file type. In A0, we will implement a basic variant of this tool in C.

We will only discern between the following small subset of the many possible file encodings.

`empty`: An `empty` file has 0 bytes.

`ASCII text`: An `ASCII text` file only contains characters from a particular subset of the ASCII-table.

[1] For an explanation of the `(1)`-notation, see Appendix A.

`ISO-8859-1 text`: An `ISO-8859-1 text` file, also known as `latin1`, originating in 1987. It uses 8 bits instead of 7, and encodes most Latin alphabets, including Faroese, Icelandic, and Danish.

`UTF-8 text`: There are more Unicode (universal encoding) standards, but here we will only look at `UTF-8 text`. These standards aim to encompass the characters required to represent all the world's languages, and are variable-width encodings (i.e., the number of bytes per character depends on the character). UTF-8 is today the predominant universal encoding. The later UTF-16 (which we do not work with) is regarded as an alternative, not an improvement over UTF-8.

`data`: Everything else is a `data` file.

We recommend using `printf(1)` with output redirection to generate files containing exotic characters (e.g., null-bytes). See also Appendix B for more information on how to generate exotic files directly from your shell (e.g., æøå).

2 API (10%)

Write a self-contained (one-file) C program, `file.c`, which when compiled to an executable `file` has the following API:

2.1. Let your `file` accept exactly one argument. If the given argument is a path to a file that exists, and the type of that file can be determined, write one line of text to `stdout`, and let `file` exit with the exit code 0 (`EXIT_SUCCESS`).

The line of text must consist of the given path, followed by a colon, space, and a guess of the types described above.

For instance, if there exists an `ASCII text` file called `ascii` in the same directory as your `file` binary:

```
$ ./file ascii
ascii: ASCII text
$ echo $?
0
```

The shell variable `$?` refers to the exit code of the previous command.

Similarly, if there exist corresponding files `data` and `empty`:

```
$ ./file datafile
datafile: data
$ ./file emptyfile
emptyfile: empty
```

Note: So far, the behaviour is exactly identical to `file(1)`. However, depending on your choice of characters, standard `file(1)` may report a *superstring*[2] of what your `file` reports. For instance, if there are no linebreak characters (`\n`), `file(1)` reports:

[2]Note that a *superstring* denote to a string that contains the related string but may also contain more. On the other side, a *substring* can contain a smaller part of the related string.

```
$ printf "Hello, World!" > ascii
$ file ascii
ascii: ASCII text, with no line terminators
$ ./file ascii
ascii: ASCII text
```

More formally, your `file` must report a non-empty *substring* of the type that `file(1)` reports. Note that the above example uses the `printf(1)` shell command and not the `printf` C function.

2.2. If instead `file` is called with *no arguments*, it prints the usage message "`Usage: file path`" to `stderr`, and exits with `EXIT_FAILURE`:

```
$ ./file
Usage: file path
$ echo $?
1
```

Note: The usage string printed for `file(1)` is far more complicated, not least because it supports far more file types than we can hope to cover. `file(1)` however, also writes a usage message (starting with the word "Usage:") to `stderr`, and exits with the exit code 1.

2.3. If we provide a path to a non-existing file, or some other I/O error occurs (e.g., someone rips out the USB-stick the file is on), behave as follows:

```
$ ./file vulapyk
vulapyk: cannot determine (No such file or directory)
$ ./file hemmelig_fil
hemmelig_fil: cannot determine (Permission denied)
```

In both cases, `file` must still print to `stdout` and exit with `EXIT_SUCCESS`.

To achieve the above behaviour, we provide the following function:

```
// Assumes: errnum is a valid error number
int print_error(char *path, int errnum) {
  return fprintf(stdout, "%s: cannot determine (%s)\n",
    path, strerror(errnum));
}
```

This utilizes the standard `strerror` function to print a standard string for a corresponding error number, as set by the standard I/O functions.

In particular, if an error occurs during `fopen(3)`, `fseek(3)`, `fread(3)`, or `fclose(3)`, the standard C library will set the so-called `errno` variable. This variable becomes available if you include the `errno.h` library. The `string.h` library then provides the function `strerror(3)`, which returns a string representation for an otherwise integer error code.

Note: `file(1)` behaves in a similar way: it exits with `0`, but writes rather different messages to `stdout`. `file(1)` does not regard invalid paths and I/O errors as errors on its part. This is a questionable design decision.

3 File Types (15%)

Your `file` should recognise the following file types:

`empty`

This file type must be reported if the file contains no bytes.

`ASCII text`

This file type must be reported if all bytes belong to the following set:

$$\{\texttt{0x07}, \texttt{0x08}, \ldots \texttt{0x0D}\} \cup \{\texttt{0x1B}\} \cup \{\texttt{0x20}, \texttt{0x21}, \ldots, \texttt{0x7E}\}$$

`ISO-8859 text`

This file type should be reported if the file composed of ISO-8859-1-like bytes. These include all ASCII-like bytes (see above), and also decimal values 160–255. The decimal values 128–159 are not part of ISO-8859-1, and their appearance might indicate that the file really is a UTF-8-encoded text file. (Why?)

`UTF-8 Unicode text`

This file type should be reported if the file is composed of UTF-8-like characters. UTF-8 is a variable-length encoding where each subsequent byte of a character begins with a designated bit-sequence. The following table summarises the encoding:

Number of Bytes	Byte 1	Byte 2	Byte 3	Byte 4
1	`0xxxxxxx`			
2	`110xxxxx`	`10xxxxxx`		
3	`1110xxxx`	`10xxxxxx`	`10xxxxxx`	
4	`11110xxx`	`10xxxxxx`	`10xxxxxx`	`10xxxxxx`

`data`

This file type must be reported in case no other type matches.

4 Hints

4.1. Create some files to test `file(1)` and your `file` with first. (See Testing below.) Explore the behaviour of `file(1)`.

4.2. Discern between `empty` and `data` first. Handle the cases when your `file` is given too few (or too many?) command-line arguments. Only then proceed to also handle `ASCII text`. Write tests as you go along.

4.3. KISS! Keep it simple, stupid! Don't try to solve all problems at the same time. Look though the file and discern one file type at the time. We are not looking for an efficient solution.

4.4. You might want to use an enum for all the supported file types and a constant array of user-friendly strings for each element of the enum. This way, your function for identifying the file type can use the enum, and the printing of the file type can happen at a later stage.

```
enum file_type {
  DATA,
  EMPTY,
  ASCII,
  ...
};

const char * const FILE_TYPE_STRINGS[] = {
  "data",
  "empty",
  "ASCII text",
  ...
};
```

4.5. Our reference `file.c` begins by including the following libraries, for the following reasons:

```
#include <stdio.h>  // fprintf, stdout, stderr.
#include <stdlib.h> // exit, EXIT_FAILURE, EXIT_SUCCESS.
#include <string.h> // strerror.
#include <errno.h>  // errno.
```

4.6. To create a non-readable file, you can use `chmod`:

```
$ printf "hemmelighed" > hemmelig_fil
$ chmod -r hemmelig_fil
```

The file is still writable, and you can also still delete it.

To make the file readable again:

```
$ chmod +r hemmelig_fil
```

5 Testing (25%)

The assignment has been carefully designed so that you can test your implementation by comparing its observable behaviour with that of the standard `file(1)` utility.

To this end, we hand out a shell-script `test.sh`. Run it as follows:

```
$ bash test.sh
```

This script will create a directory `test_files`, and fill it with sample files.

The script will then compare the expected output (generated by `file(1)`, filtered to remove anything following `ASCII text`), and the `actual` output (generated by your `file` utility). If the files differ, the script will fail.

The comparison is done using the standard `diff(1)` utility. You should be familiar with diff's from Software Development. See also Appendix C.

Your task: make it generate more interesting sample files, such that you get a covering unit testing. That is:

- Generate at least a dozen relevant files for testing of each relevant text encoding.
- Generate at least a dozen files for testing `data`.
- Is the `empty` file type already tested?

Consider and document how well your test covers the input problem. What is tested well and what is hard to text?

6 Report (30%)

Alongside your solution (implemented code and tests), you should submit a short report; must not exceed 5 pages of textual content excluding the theoretical part[3].

The report *must*:

- Have a title that states the assignment solved on the first page
- Clearly state the names and KUids of all group members on the first page

The report emphshould:

6.1. Shortly introduce the problem.

6.2. Discuss your approach to it and your design decisions.

6.3. Disambiguate any ambiguities you might have found in the assignment.

6.4. Present the non-trivial parts of your implementation.

6.5. Describe how to compile your code and run your tests to reproduce your test results.

6.6. Discuss the coverage of your test and what the outcome were.

6.7. Conclude on the outcome of the entire solution.

6.8. Answers to the theoretical questions

See the given report example for inspiration and details.

7 Theory (20%)

In addition (exceeding the above mentioned 5 pages) to the report over programming task, you should also hand-in answers to the following questions. For each, write a couple of lines elaborating on your answer.

[3]These 5 pages is a strict upper limit. Quality over size; less can easily be a very good report.

7.1 Boolean logic

Show if the following Boolean expression is correct or wrong:

$$(A \hat{} B) \,\&\, A = (A \,\&\, \tilde{} B)\,.$$

7.2 Bit-wise logical operators and representation

Answer the following questions using only bit-wise logical operators (i.e. `&`, `|`, `^`, `>>`, `<<`, and `~`) or logical operators (i.e. `&&`, `||`, `^`, and `!`) in C. Explain your choice.

- Given an integer value `int x`, give a C expression that returns the value multiplied by 8.
- Given an integer value `int x`, give a C expression that returns true if the value is equal to 6.
- Given an integer value `int x`, give a C expression that returns true if the value is less than or equal to 0.
- Given integer values `int x` and `int y`, give a C expression that returns true if the value of `x` and `y` are different.

7.3 Floating point representation

- Explain the advantages of having denormalised numbers in the IEEE 754 floating point format.

7.4 Language levels

The following two questions are more discussion based than the previous. Each should probably have a paragraph discussing the underlying concepts, and your interpretation of them. These are the sorts of questions that I love to ask in exams so be sure to practice.

- If everything runs in machine code anyway, why do we program in higher level languages like C?
- If higher level languages like C are so good, why don't we invent a CPU that runs uses that as its instruction set instead?

8 Learning Goals and Evaluation

The assignments will be graded with points from 0 to 4 and it is not possible to re-hand-in any of the assignments. Note, that to qualify for the exam you are required to achieve at only 50 % of the total number of points in the assignments. Thus, you are not expected to make everything of the assignment.

Always be aware that you show what you have learned in the course. For example, this is not a course on algorithms, so we are not looking for an efficient implementation. The learning goals of this assignment is to:

- implement relatively simple programs in C (use different loops and conditionals);
- work with files (open, read, close, etc.);
- test and evaluate a program;
- work with bit-wise operators;
- understand encoding of data (here the file formats);
- document your work in a report.

Here is an outline of what is expected for this assignment:

Good Implement `empty`, ASCII, and data. Make a handful of tests that shows how the program works. Make a nice report (2-3 pages) that documents and evaluates the implementation (see reports example).

Very good Implement all file types, write good tests, and document the work in the report. Solve the theoretical tasks. Some significant errors are accepted.

Expert Implement everything but with only some minor errors.

9 Handout/Submission

Alongside this PDF file there is Zip archive containing a framework for the assignment.

```
$ unzip src.zip
```

You might have to download the 'unzip' command-line utility. Another option is to temporarily rename this file to something ending in '.zip', use your standard unzipping utility, and then rename the file back to PDF. You can also download the ZIP archive uploaded alongside this PDF file[4].

You will now find a `src` directory containing the following:

`.gitignore`
: A suitable `.gitignore` file, should you choose to use Git.

`file.c`
: A skeleton file for your `file.c`.

`Makefile`
: The file that configures your `make` program.

  To compile the code with all the appropriate compiler flags, navigate your shell to the `src` directory, and simply execute `make`.

[4] These are, in fact, the same file.

`test.sh`

A shell script for testing your implementation.

Your task is to modify `file.c` and `test.sh`. You should hand in a ZIP archive containing a `src` directory, and the files `src/file.c`, `src/Makefile`, and `src/test.sh` (no ZIP bomb, no sample files, no auxiliary editor files, etc.).

To `make` this a breeze, we have configured the `Makefile` such that you can simply execute the following shell command to get a `../src.zip`:

```
$ make ../src.zip
```

Alongside a `src.zip` submit a `report.pdf`, and a `group.txt`. `group.txt` must list the KU-ids of your group members, one per line, and do so using *only* characters from the following set written using *standard alphabetical encoding*:

$$\{\texttt{0x0A}\} \cup \{\texttt{0x30}, \texttt{0x31}, \ldots, \texttt{0x39}\} \cup \{\texttt{0x61}, \texttt{0x62}, \ldots, \texttt{0x7A}\}$$

Please, make sure that the file is UTF-8 encoded with UNIX line endings. An example of this is:

```
dlb838
mdt733
agy623
```

A `man(1)`

When we write `file(1)`, instead of just `file`, we mean that you can use the Unix-like command-line utility `man(1)` to look up the documentation of the command. For instance, try typing this at a Unix-like command line:

```
$ man 1 file
```

`man`-style documentation is organized into sections, and `file(1)` indicates that the documentation of `file` is in section 1. Often, `man` can guess the section you mean, and you can get away with things like this:

```
$ man file
```

`man`-style documentation is used not only for programs, but also other aspects of a Unix-like system. For instance, you will find a description of the file system hierarchy under `hier(7)`. Beware however, not all software developers keep their documentation up to date. You should.

Note: `man(1)` typically uses the `less(1)` to show the documentation, showing some basic controls at the bottom of the screen: To quit, type q. To get help, type h. Further, to scroll in the documentation using the ↑, ↓, PgUp, PgDn, Home, End.

B `printf(1)`

`printf(1)` prints a line of text to the standard output. You can use `printf` in conjunction with output redirection to create all sorts of files.

For instance, to create an ASCII file called `ascii`:

```
$ printf "Hello, World!\n" > ascii
$ file ascii
ascii: ASCII text
```

To create what will be regarded as a data file by `file(1)`, it suffices to insert an extra null-character somewhere in the file. Since the null-character is not printable (and hence, not writable), we will need an escape sequence (similar to standard C `printf(3)`):

```
$ printf "Hello,\x00World!\n" > data
$ file data
data: data
```

C `diff(1)`

`diff(1)` can compare files line by line. If you are familiar with the revision control system, git, you should be familiar with the format that `diff(1)` outputs with the command-line argument `-u` (for "unified context").

```
$ printf "Hello,\nWorld!" > x
$ printf "Hello,\nBrave New World!" > y
$ diff -u x y
--- x 1970-01-01 00:00:00.000000000 +0200
+++ y 1970-01-01 00:00:00.000000000 +0200
@@ -1 +1 @@
-Hello,\nWorld!
+Hello,\nBrave New World!
```

The exit code of `diff(1)` is non-zero if the files differ.

Note: `diff(1)` and `patch(1)` are rudimentary in a Unix-like development environment, although sometimes hidden behind high-level tools like `git(1)`.