A1: `file(1)` — Unicode

Computer Systems 2017
Department of Computer Science
University of Copenhagen

Oleks Shturmov <oleks@oleks.info>

Due: Sunday, September 17, 23:59
Version 2 (September 12)

This is the second and last assignment on implementing a `file(1)` variant. As before, 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 3 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.

Introduction

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

In A0, we explored the ASCII-table. Originating in 1972, ASCII is an old, but sturdy standard for encoding the English alphabet, Arabic numerals (digits), basic Latin punctuation, and more—spanning just 7 bits. Being limited to the English alphabet however, it is no panacea for a multilingual world.

This assignment is about exploring some of the subsequent standards, which never-the-less all conveniently build upon ASCII:

(1) First, we take a look at ISO-8859-1, also known as `latin1`, originating in 1987. It uses 8 bits instead of 7, and encodes most Latin alphabets, including Faroese, Icelandic, and Danish.

(2) Then, we take a look at the Unicode (universal encoding) standards UTF-8 and, briefly, UTF-16.

    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, and UTF-16 should be regarded as an alternative, not an improvement.

As before, empty files should be marked as `empty`, and files for which you cannot discern the type, but are non-empty, should be marked as `data`.

Similarly, you can use `printf(1)` (when in `bash(1)`) with output redirection to generate files containing exotic characters (e.g., `æøå`).

1 API (15%)

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

1.1. Let your `file` accept one *or more* arguments.

For each 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`, consisting of the given path, followed by a colon, *some spaces*, and a guess of the file type, as specified in Section 2. Exit with `EXIT_SUCCESS`.

The number of spaces used for each given path must be such that the guesses are *aligned* in the output. For instance:

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

To achieve the above behaviour, you can use the following format, where `path` is a given path, `max_length` is the length of the longest path given, and `guess` is a file type guess for that particular `path`:

```
fprintf(stdout, "%s:%*s%s\n",
  path, max_length - strlen(path), " ", guess);
}
```

Note: This behaviour is identical to `file(1)`. However, depending on your choice of characters, standard `file(1)` may report a *superstring* of what your `file` reports for each path. For instance, if you there are no line-break characters(`\n`), `file(1)` reports:

```
$ 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, for each path.

1.2. 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), you should adhere to the alignment rules above, but otherwise behave as in A0:

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

All lines are printed to `stdout` and `file` still exits with `EXIT_SUCCESS`.

To achieve the above behaviour, we modify the `print_error` function:

```
// Assumes: errnum is a valid error number
int print_error(const char *path, int max_length, int errnum) {
  return fprintf(stdout, "%s:%*scannot determine (%s)\n",
    path, max_length - strlen(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.

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.

1.3. 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 File Types (15%)

Your `file` should recognise the following file types:

`data`

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

`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`

`Little-endian UTF-16 Unicode text`

This file type should be reported if the file is composed of UTF-16-like characters, encoded in little-endian. As a simplifying assumption, you may assume that such a file always begins with a byte-order-mark (BOM) `\xFF\xFE`.

`Big-endian UTF-16 Unicode text`

This file type should be reported if the file is composed of UTF-16-like characters, encoded in big-endian. As a simplifying assumption, you may assume that such a file always begins with a byte-order-mark (BOM) `\xFE\xFF`.

3 Theory (20%)

VER. 2

The theory-part of this assignment has to do with number formats, the precedence of operators in C, and some basic C-style arithmetic. The exercise illustrates how you can use GDB to test your C preprocessor macros.

To enable this, you need to make sure to export sufficient debugging information when compiling your program. The `CFLAGS` variable in our handed out `Makefile` now lists the debugging level `-g3` (instead of just `-g`) to enable this.

We have also added a new phony target to the handed out `Makefile`—you can now `make gdb_test`. This will build your `file` and run it with GDB. The flow of the run is controlled by the handed out `test.gdb`. Currently this file merely starts GDB and quits:

```
start
# TODO: Add debugging commands here.
q
```

Modify the handed out `test..gdb` to do the following, in the following order:

3.1. Print the hexadecimal representation of 192_{10}.

3.2. Print the binary representation of 192_{10}.

3.3. Print the binary representation of 80_{16}.

3.4. Print the decimal representation of 80_{16}.

3.5. Print the hexadecimal representation of 110_2.

3.6. Print the decimal representation of 110_2.

Declare the macros `UTF8_2B`, `UTF8_3B`, `UTF8_4B`, and `UTF8_CONT` at the top of your `file.c`, which take an argument, and yield a *positive* value if it looks like the first byte of a 2-byte, 3-byte, 4-byte, or a continuation byte of a UTF8 character, respectively. Each macro should yield 0 otherwise.

Extend your `test.gdb` with the following. For each of the below "tests", GDB should print 1. In your report, explain why, and how you achieved this.

```
#
p "Some basic tests.."
p UTF8_CONT(128) != 0
p UTF8_2B(192) != 0
p UTF8_3B(224) != 0
p UTF8_4B(240) != 0
#
p "Testing UTF8_CONT.."
p UTF8_CONT(128 + 1) != 0
p UTF8_CONT(128 | 1) != 0
p UTF8_CONT(128 | 63) != 0
p UTF8_CONT(128 | 63) > 0
p UTF8_CONT(128 + 64) == 0
p UTF8_CONT(128 | 64) == 0
#
p "Testing UTF8_2B.."
p UTF8_2B(128 + 64) != 0
p UTF8_2B(128 | 64) != 0
p UTF8_2B(128 | 64 | 31) != 0
p UTF8_2B(128 | 64 | 31) > 0
p UTF8_2B(128 + 64 + 32) == 0
p UTF8_2B(128 | 64 | 32) == 0
#
p "Testing UTF8_3B.."
p UTF8_3B(128 + 64 + 32) != 0
p UTF8_3B(128 | 64 | 32) != 0
p UTF8_3B(128 | 64 | 32 | 15) != 0
p UTF8_3B(128 | 64 | 32 | 15) > 0
p UTF8_3B(128 + 64 + 32 + 16) == 0
p UTF8_3B(128 | 64 | 32 | 16) == 0
#
p "Testing UTF8_4B.."
p UTF8_4B(128 + 64 + 32 + 16) != 0
p UTF8_4B(128 | 64 | 32 | 16) != 0
p UTF8_4B(128 | 64 | 32 | 16 | 7) != 0
p UTF8_4B(128 | 64 | 32 | 16 | 7) > 0
p UTF8_4B(128 + 64 + 32 + 16 + 8) == 0
p UTF8_4B(128 | 64 | 32 | 16 | 8) == 0
#
p "More student tests.."
```

```
# TODO: Add more tests here.
```

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. Start by adding support for a variable number of paths given as command-line arguments. Manually test that this works.

4.3. Then, refactor your code to take alignment into account. Now, you can test with the handed out test-suite.

5 Testing (20%)

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.

Your task: make it generate more interesting 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.

5.1. Generate at least a dozen files for each of the file types that you claim to support.

5.2. Add your tests from A0 to ensure that you haven't broken your implementation for the file types `ASCII text`, `data`, and `empty`.

5.3. Add more `data` tests (to increase your code coverage, now that more file types are supported).

5.4. Extend `test.sh` to also test other elements of the API. For instance, non-existing paths.

6 Report (30%)

Alongside your solution, you should submit a short report. The report should:

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

6.2. Discuss the non-trivial parts of your implementation and your design decisions, if any.

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

The report is expected to be 2-3 pages and must not exceed 5 pages.

Handout/Submission

True to the spirit of this assignment, this file is both a PDF and a ZIP archive: it is a so-called polyglot file.

If you are reading this, congratulations! You have successfully or opened a ZIP archive in your PDF viewer (or printed one). Now let's unzip a PDF file[1]:

```
$ unzip CompSys17-A1.pdf
```

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[2].

You will now find a `src` directory containing the same `.gitignore` and `file.c` as in A0, but a modified `Makefile` and `test.sh`, as well as a new file called `test.gdb`: VER. 2

- The test-suite has been modified to test `file` with a variable number of arguments. In fact, all input files are given to `file` at once.
- The `Makefile` has a new phony target called `gdb_test`, which uses the given (fairly blank) `test.gdb` script to "test" `file` further. VER. 2

Your task is to modify `file.c`, `test.sh`, and `test.gdb`. You should hand in a ZIP archive containing a `src` directory, and the files `src/file.c`, `src/Makefile`, `src/test.sh`, and `src/test.gdb` (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:

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

[1] PDF files are not ZIP archives in general; some formats are (e.g., DOCX); this is a hack.
[2] These are, in fact, the same file.