A0: file(1) — Unicode

Dea Felicia (dlr211), Said Said (hdn456), Kristian Jelsted (zxw463)

September 2023

1 Introduction

This report documents our solution to the A0 assignment. In this assignment we were tasked with creating a program that has the functionality of the Unix command file(1) using C, as well as creating tests for our implementation. Our program should be able to distinguish between the file types empty, ASCII, ISO-8895, UTF-8 and Data when given a file and should be equipped to handle errors such as being given the wrong amount of arguments, being given a file that does not exist and being given a file that is not readable. This report will also include our explanation of relevant theory questions.

2 Design

We have designed our main function to first handle potential errors such as being given the wrong number of arguments or a file that does not exist or isn't readable. We will use the access() function to do this, by asking it if the files exists and if they are readable. We have done this to prevent the program from crashing.

Afterwards we check the contents of the file to determine the file type. We have decided to do this by creating a function for each of the 3 main file types. These functions checks if a file contains characters used in one of the 3 file types. We will then create another function that use the 3 type functions to distinguish between the different types and gives an output that we can use for our final print statement.
We wanted to use the approach where we create functions to the different jobs, meaning that one function only has one job. That this simplifies the process and makes it is easier to error handle, debug and potential improvement code in the future. Shortly said we tried to divide the functions such it use the single responsibility principle.

3 Implementation

We have used Enum to categorise the 5 data types we're dealing with and then created an array with matching strings for each of the file types. We did this to create a more structured code without code duplication since this allowed us to just print at the end of our program after we close the file.

To check if the contents of the text is written in ASCII we've created a function called check_ASCII(const unsigned char). This function checks if the given char is within the ranges for ASCII text, if it is the function returns 1 otherwise the function returns 0. When checking for ISO-8859-1, we use the same method but checking if the char is instead within the ranges of ISO-8859-1.

To check if the contents of the text is written in UTF-8 we've created a function called check_UTF8(const unsigned char, FILE *const file) and a helper function UTF8_sequence_length(const unsigned char). The first function UTF8_sequence_length determines the length of the sequence by looking at the headerbyte. Returns 3 if 4 bytes, returns 2 if 3 bytes, returns 1 if 2 bytes and returns 0 if neither matches which would indicate it is 1 byte. We are using the bitwise AND operator. To perform operation between the two values, the header byte value and the value of the different header sart sequence in UTF-8. Where we check if the two, three or 4 most significant bits of the "byte" are set to the different header start sequences: 0xC0, 0xE0, and 0xF0.
In the other function check_UTF8 we start off my calling the other function to determine how many bytes there could be in the UTF-8 sequence if it is a UTF-8 sequence. Meaning how many bytes according to the header byte should we read afterwards. If the sequence lenght is 0 it will return 0 since this is not UTF-8 encoded. Using ftell we store the original position. If it mananges to fail and it is not UTF-8 encoded we want to reset the file pointer to the original position before starting. Using fread we read the next byte and using bitwise XOR operator between the next byte and hexadecimal value 0x80 to check if it matches the UTF-8 encoding sequence. So if the two most signifant bits matches "10" it will result in 00 which would indicate valid continuation byte using right shift operator 6 bits to the right which isolates the result from XOR, so if the result is 0x00 it is continuation if not it is not continuation byte or UTF8 encoding. Will return 1 if it is UTF-8 and return 0 and reset the pointer if not.

Now that we have functions to identify the different file types we create a function called get_file_type(file) that uses the 4 previous functions to determine the file type.
In the get_file_type we first check whether first read byte is == 0, if that's the case it would be an empty while. The we reset it using fseek and start from the beginning of the file. Creating a long variable to store the original position. If the file is not empty we predefine it as ASCII, which will change if it is not ASCII and stay the same if it is. Using a while loop where we use fread to read

one byte at a time. We use the different functions we defined earlier to check whether this byte is an ASCII, UTF-8, or ISO-8859-1. If it is none of these it will return FileType DATA.
In the main function we stared by checking if the number of arguments given to the function was correct, if it wasn't the program return EXIT FAILURE. If the correct number of arguments are given the program opens the file using fopen() and checks if the file given exists and is readable, this is done with the use of access(), that checks if the file is accessible. If the file was either non existent or non readable we gave an error message using the print_error() function we were given, to send the appropriate error message to the user. Then we run the get_file_type(file) we've created, close the file and return the result from our function. If the function has been given the right amount of arguments the the function will always return EXIT_SUCCESS, if the wrong amount of arguments are given the program will return EXIT_FAILURE

To run our program we start by making the file with the command "make file" and then we use our program by calling it along with the text document we want to check, by using the command "./file filename".

4 Tests

We were given four basic tests for the function. We have expanded these tests to test the edge cases of our program. We have created tests for an empty file, a data file, ASCII files, ISO-9885 files and UTF-8 files. We also tested to see if our program could handle being given a non readable file, by creating a file, making it non readable and then checking if the error message corresponded to the error.

For both ASCII, ISO-8859-1 and UTF-8 we tested both with and without the newline command, to make sure it worked. We also tested both shorter and longer texts for the file types. For the UTF-8 texts we tested files with 1, 2, 3 and 4 byte-sequences, to make sure all of theses worked

By testing ASCII, ISO 8859-1, UTF-8 and data we have also tested empty since empty is the first branch in the get_file_type(file) function, and will therefore cause trouble for the other tests if it hadn't worked, but we've also created a test specifically for the empty file type.

Our test coverage of our test is fine could have done more testing on the different functions to make sure there was no errors. But the test tested the different functions in our code. Made sure to get most of the cases in the file.c. Getting into all of the if statements to make sure we got the appropriate output. There could been made a few more test cases, some simple and some more extreme where our code might fail. The result was as expected but there was some issue with writing UTF test, if we didn't write new line, the original file function would give a different kind of output. But overall test coverage was fine could

been better and result was as expected.
We run all of our tests by using the command ”bash tests.sh”

5 Conclusion

We’ve created a program that mimics the functionality of the Unix command file(1) and is able to distinguish between the file types Empty, Data, ASCII, ISO-8859 and UTF-8 using access(), fopen(3), fseek(3), fread(3), and fclose(3). We’ve have implemented all functionality requested in the assignment and we’ve created tests to check for edge cases and errors that our program might have and finally we have answered the accompanying theory questions down below.

6 Theory

7.1

the Boolean expression (A ˆ B) & A = (A & ˜ B) Holds true. On the left side we are told that either A OR B should be true and A IS true, meaning it is only true when only A is true. On the right side we are told that A must be true AND B must not be true, meaning that the right side is also true if only A is true. we have created a table for it down bellow.

A	B	(A ˆ B) & A	(A & ˜ B)
0	0	0	0
1	0	1	1
0	1	0	0
1	1	0	0

7.2

- int MulEight = (x $\ll$ 3)
  To create a C expression that returns the int multiplied with 8 we use $\ll$ because this shifts the x's bits to the left which is equvilent to multipling it with 2^3

- bool isSix = (x & 6)
  To create a C expression that returns true if x = 6, we simple say that both x and 6 needs to return true.

- bool lessOrEqualZero = (x <= 0)
  To create a C expression that returns true if x ¡=0,

- bool xEqualToy = (x ˆ y)
  To create a C expression that returns a value if x and y aren't equal we use the XOR(ˆ). by saying that only y or x can be true. If x and y where equal the expression would return false since only one of them can be true and they both would return true if they were the same value.

7.3

IEEE754 represents floating-point numbers (both single precision and double precision). The advantages of having denormalised numbers in the IEEE754 floating point format, is that you can represent values much closer to zero, allowing for increased precision when dealing with extremely small values. This stops us from flat out rounding down to 0.