Re: Assignment A1
CompSys

B. Alix, K. A. Madsen, C. S. Hansen

3. marts 2019

1 Introduction

In this following assignment, we will explore some of the subsequent standards, which build on ASCII. We will take a look at `ISO-8859-1` also known as `latin1`, which uses 8 bits and encodes most Latin alphabets, including Faroese, Icelandic and Danish. We will also take a look at Unicode standards as UTF-8 and UTF-16. Where UTF-8 today is the predominant universal encoding, while UTF-16 is an alternative, not an improvement.

2 Compiling and running the code

1. Unzip src.zip.
2. Navigate to an open the src directory.
3. Use command: `make` to compile the programs.
4. Use command: printf "content" > name_of_file, to create files.
5. Use command: `./file ascii data empty` for the program to guess the file types of the files you just created.
6. Use command: `bash test.sh` to run ./file and file(1) comparison-tests.
7. Use command: `gdb file -x test.gdb` to run the gbd test script.

3 Implementation

The implementation of `file.c` has its purpose to determine different kinds of file types of on or more given files. When running the program, it will first find out if any arguments have been given. If none has been given, it will print out a usage message, while if some argument(s) are found it will find the longest argument, and use it to compute the number of spaces that the final output uses. It will then determine the type of the amount of arguments given.

3.1 Empty files

To check if any given argument(s) are empty files, our `CheckForEmpty` function will take care of this. It uses `fseek()` and `ftell()` to calculate the size of the file(s) and if `fseek()` is unable to move the pointer, then that means that the length of the file is zero and the file will be set to be empty.

3.2 ASCII files

The function `CheckForAscii` checks if a file contains any characters that are not to be considered ASCII characters. I.e., these characters will lie outside of the following set: $\{0x07, 0x08, ..., 0x0D\} \cup \{0x1B\} \cup \{0x20, 0x21, ..., 0x7E\}$, if the file consist of non-ASCII characters. The file will then be considered as a non-ASCII, otherwise the file will be considered as ASCII.

3.3 ISO-8859 files

The `CheckForISO` function works similarly to the `CheckForAscii` function. Compared to the ASCII function, the ISO checker includes all the ASCII-like bytes, and also decimal values in 160-255.

3.4 UTF-8 files

To check if a given file(s) is of type UTF-8 Unicode text, we have the function `CheckForUTF8`. It can determine if a file is encoded in UTF-8 Unicode text by using the designed bit-sequence, which can be seen on page 4 in assignment 1. However, the different number of bytes has different patterns of bytes in UTF-8 accordingly to table (on page 4). Therefore, we have chosen to implement four if-statements to cover the four numbers of bytes (which can be seen on page 4 in the assignment).

3.5 UTF-16 files

To check if a given file(s) is of type Little-endian UTF-16 Unicode text or Big-endian UTF-16 Unicode text, we have the `CheckForUTF16` function. It works in similarly way as the `CheckForUTF8` function, as it also uses a characteristic (of UTF-16) approach to determine whether Big-endian or Little-endian is used. Where if the file begins with a byte-order-mark (BOM): `x\FF\xFE`, it is Little-endian, while if the file begins with a byte-order-mark (BOM): `x\FE\xFF`, it is Big-endian.

4 Type checker

To make such that our program can differentiate between the different file types, we have made a function called: `CheckForType`. This detects and returns the corresponding type number to a print function located in the our main function, with the help of type checkers. However, as we use alot

of if-statements for this function, the running time of our program becomes quite high.

The function works by, we first check if the file is empty, since if it is empty it is not necessary to run through the rest. Then we check for ASCII file type, as the ISO-8859 file type has a bit bigger range. We then assume that a file cannot not have both a UTF-8 and UTF-16 encoding. We then check for UTF-16 and the UTF-8. Lastly we have an else statement, which takes care of the data type that return the corresponding type number: zero.

5 Theory

To make each of the ”tests” in the test.gdb file print 1, we have extended the file with the use of macro. For our macro function, we have made use of bit masking to compare number of bits to a designated bit-sequence. To achieve this, we have made use of the ternary conditional operator `?:`, as this also makes the macros simple and easy to read. The ternary conditional operator (syntax): `condition ? true : false` evaluates a boolean expression, and returns the result of evaluating one of two expressions, depending on whether the Boolean expression evaluates to true or false.

6 Decompilation of C program

6.1 Program 1

This program takes a long (64 bit int), and test it against if it is negative. If it negative, then it will return the value, else it will negate it and return the negated value. The code can be found in the src directory, named as `program1.c`.

6.2 Program 2

This program takes a long and a pointer to an array of longs. It start by checking if each long stored, is starting from a location that is identical to a given input. If it is, it will remove it and sub-steps will backwards break the stored memory. It continues until a zero is found, and will then return how many occurrences of the input long were removed in total. Since we found that this program can null terminate, we considered that this program might be designed for strings, as it stops at the end of the string. However, we chose to rewrite the program as we saw it should

be implemented. The code can be found in the src directory, named as `program2.c`.

7 Testings

7.1 test.sh

For the `test.sh`, we test our implementation by comparing its observable behaviour (of ./file) with that of the standard file(1) utility, where if the files differ, the script will fail. When running the shell-script `test.sh`, it processes the file types that our program supports. For the tests, we have made at least 12 tests for each file type, with an exception to UTF-16 and UTF-8. As we found, it was not sufficient to just write different kinds of characters to various files, and therefore we are also using `iconv(1)` to encode the files. This ensures that any given file(s) (arguments) has the right character encoding for the tests.

The way we handle the UTF-16 Big-endian and Little-endian character encoding, we first encode the test file to UTF-16, leaving the program to determine what endian version is being used. Then, we examine the encoding of the UTF-16 file and finally we use `iconv(1)` to convert from the encoding that the machine defined, to the encoding that our tests requires.

7.2 test.gdb

In our test script `test.gdb`, it shows different kinds of basic use scenarios of gbd. In the start of the test we print number representations of e.g. hexadecimal representation of 192_{10} and binary representation of 192_{10}. E.g. to represent hexadecimal representation of 192_{10}, we use p for print and x to print in hexadecimal: $p/x192$. Next, we define some macros for respectively UTF8_2B (2-byte), UTF8_3B (3-byte), UTF8_4B (4-byte), and UTF8_CONT (continuation-byte), such that we are able to run the tests that we have been given, as well as some we have created. We define these macros inside the test-script, as this is necessary because gdb cannot by itself access those there exists within `file.c`.

8 Discussion

8.1 Non-trivial parts of the implementation

The non-trivial part of the implementation was clearly the implementation of the UTF-8 Unicode text checker, since we firstly had to use a lot of time

to figure out how we should interpret the table provided to us in the assignment at page 4. But we ended up interpreting it as we should convert the binary number of bytes to hexadecimals and use those to check for, if the file fit into any of the number of bytes, by checking its designated bit-sequence.

However, we also did have a lot of trouble how we should implement the main function. I.e. how we should stick everything together, but we found it was most easy to create a type checker function beforehand, to take care of choosing the right type for the specific files. In the main function, we handle if the program finds a file, and if it does it should print the corresponding file type, else it should spit out an error message.

8.2 Ambiguous formulations

It took quite long for us how to understand and how to implement section 1.2 in the assignment description. It was not very precise.

We found it hard to figure out whether or not we should answer the following question in the report. In the assignment text about ISO-8859 text, where it is described that: "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?)".

We found it hard to understand what is meant by: "no ZIP bomb, no sample files, no auxiliary, editor files, etc.", in the Handout/Submission section. We think it was as well badly explained, that to test if our program worked, we had to create the an ascii, data and an empty file in the source directory (to test for `./file ascii data empty`).

9 Conclusion

For this assignment, we have done our best to implement the program `file.c`, the test files `test.gdb` and `test.sh`, while as well the decompilation of C programs `program1.c` and `program2.c`. We have also answered questions in the report. In doing so, we have gained a greater understanding of different kinds of file types and how to implement a program, that can recognize these types from given inputs. We have also learned about the usage of gdb, running scripts and testing our programs.