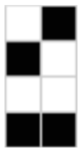
Overview

In this exercise you will be writing a simple program that converts an image from a binary format to a different format in ASCII. The input & output formats will be described, and an example of the same image in the input & output formats will be provided. The deliverables of the exercise are provided at the end of this document.

Definition of an Image

For the purposes of this exercise, an image is considered to be composed of a number of pixels organized in rows. Each pixel has a red, green & blue component, each ranging from 0 to 255 in value, so a single 8-bit byte can hold each color component, making each pixel consume 3 bytes of memory. Each row of pixels can be an arbitrary number of pixels long, so any file format must clearly delineate the rows. Every row in the image must be the same length in pixels.

An example image used below when discussing the image file formats is an image that contains 4 total pixel rows, each row being 2 pixels wide. The first 2 rows have alternating white & black pixels (i.e., in a checkerboard), followed by one all-white row and one all-black row:



Input file format

The input file is a binary image file that is a more compact form of an image than is described above, and contains all the information required to reconstruct the image in memory. The image file is encoded as a byte stream, composed of a set of nested Type-Length-Value (TLV) structures. The TLV structure is defined as:

```
[type: 1 byte] [length: 2 bytes] [value: <length> bytes]
```

Therefore, every TLV is at least 3 bytes long. Table 1 below contains the binary values of the types (in hexidecimal), the name of each type for the purposes of describing the format, and a description of what that type's values are composed of. The value portion of one TLV may be a set of TLVs itself, depending on the value composition, given below.

Type	Name	Value Composition	Comments
0x01	image-file	filename / color-table / pixel-data	An image-file contains exactly 1 filename TLV, 1 color-table TLV and 1 pixel-data TLV, in any order
0x02	filename	Text-string	String containing the image file's name, encoded as UTF-8 characters
0x03	color-table	color-mapping	A color-table contains any number of color-mappings, in any order
0x04	color-mapping	4 bytes of data: key R G B	Key is used in the pixel-group and single-pixel types, R G B are the red, green & blue values for this pixel from 0-255
0x05	pixel-data	pixel-row	There are any number of pixel-row TLVs in the pixel-data TLV
0x06	pixel-row	single-pixel | pixel-group	A pixel-row TLV contains 1 to n single-pixel or pixel-group TLVs; all rows in an image file must contain the same number of pixels
0x07	pixel-group	2 bytes of data: num key	A pixel-group represents a contiguous line of 0-255 pixels, held in <num>, of the same color referred to by <key>.
0x08	single-pixel	1 byte of data: key	A single-pixel represents a single pixel in the image of color referred to by <key>.

Table 1: Input format TLV details

We would encode the example image given in the “Definition of an Image” section using the Input File image format into the following byte stream (shown here in hexadecimal digits, separated by whitespace):

```
01 00 46 02 00 09 46 6f 6f 2e 69 6d 61 67 65 03 00 0e 04 00 04 01 ff ff ff 04 00 04 02
00 00 00 05 00 26 06 00 08 08 00 01 01 08 00 01 02 06 00 08 08 00 01 02 08 00 01 01 06
00 05 07 00 02 02 01 06 00 05 07 00 02 02 02
```

To further illustrate, this is how it would look when broken down and annotated with the TLV structure:

```
  01            00 46
[ image-file / 70 bytes long /
         02         00 09           46 6f 6f 2e 69 6d 61 67 65
       [ filename / 9 bytes long / Foo.image                    ]
         03            00 0e
       [ color-table / 14 bytes long /
                04             00 04          01       ff       ff       ff
              [ color-mapping / 4 bytes long / key = 1, R = 255, G = 255, B = 255 ]
                04             00 04          02       00     00     00
              [ color-mapping / 4 bytes long / key = 2, R = 0, G = 0, B = 0 ]
       ]
         05            00 26
       [ pixel-data / 38 bytes long /
                06          00 08
              [ pixel-row / 8 bytes long /
```

```
                      08              00 01          01
                    [ single-pixel / 1 byte long / key = 1 ]
                      08              00 01          02
                    [ single-pixel / 1 byte long / key = 2 ]
             ]
               06          00 08
             [ pixel-row / 8 bytes long /
                      08              00 01          02
                    [ single-pixel / 1 byte long / key = 2 ]
                      08              00 01          01
                    [ single-pixel / 1 byte long / key = 1 ]
             ]
               06          00 05
             [ pixel-row / 5 bytes long
                      07             00 02          02 01
                    [ pixel-group / 2 bytes long / num = 2, key = 1 ]
             ]
               06          00 05
             [ pixel-row / 5 bytes long /
                      07             00 02          02 02
                    [ pixel-group / 2 bytes long / num = 2, key = 2 ]
             ]
      ]
]
```

Output file format

The output file is a simple bitmap, held in a plain-text, ASCII file format. Each pixel row is a number of pixels wide, capped with a newline character (`\n`); each row must be the same length. This format doesn't use TLVs, instead the positions of the data are fixed, and whitespace characters are used to add structure to the file. It also eliminates the color table; each pixel row is a number of pixels separated by whitespace with their color given as 3 hexadecimal digits from 0x00 - 0xff. Additionally, the number of pixels per row & number of rows are given to allow in-memory reconstruction of the image to be optimized; the numeric values for pixels-per-row and number-of-rows are in decimal format.

The file structure is as follows, with newlines being shown as `\n` and `...` meaning that data has been skipped over:

```
<filename>\n
<number of pixels per row> <number of rows in image>\n
\n
<pixel row #1>\n
<pixel row #2>\n
...
<pixel row #n>\n
\n
```

The example image from above would be encoded in this format as:

```
Foo.image
2 4

ff ff ff 00 00 00
00 00 00 ff ff ff
ff ff ff ff ff ff
00 00 00 00 00 00
```

Exercise Deliverables

Design and implement a program that takes as input a file containing an image in the binary format as described in "Input format" above (NOT the textual form, but a binary form), and output to standard output the image in the ASCII format as described in "Output format" above. This must be written in C++, and you may not use any external project code, reference code from the web or any other libraries aside from what your compiler provides in the C++ Standard Template Library and the C Standard Library. Please include all source & any other files required to compile your code (e.g. Makefile, Xcode project file, etc.). The example must be able to be compiled on a UNIX platform with the clang or gcc compilers or on Mac OS X using Xcode; please do not submit Visual Studio projects.

Please contact the recruiter with any questions, but this file should contain enough information for you to complete the exercise. You may need to balance readability, extensibility, maintainability & efficiency, but do not sacrifice correctness for any of the other factors, as the solution MUST correctly operate on the example input & output files and examples as given in this file. Your design and implementation should be representative of your efforts at creating production-ready code for a project that could grow and evolve in the future; while the current goals of the exercise are limited, treat your solution like you would any other project that you would have ownership of!

Good luck & have fun!