The task is to develop a function to handle an image file -- the function shall be able to read the image, change the interleaving and save the modified image.

We assume a simple uncompressed image files in this assignment. An image can be seen as a 2D array of pixels, the size of the image corresponds to the width and height of the array. The easiest way to save a 2D array into a file is to traverse the array in a row-major order, i.e. save the first row, then second row, ..., until the entire array is serialized into the target file. Pixels in the image do not need to be integers, instead, each pixel of the image may be formed from tuples (channels), e.g. 3 channels (RGB components), 4 channels (RGB and opacity component), or simple 1 channel (shades of gray). Next, each channel value is represented with some precision, the precision is given by the number of bits per channel value. We assume precision 1 bit, 8 bits, or 16 bits per channel, and we assume 1, 3, or 4 channels per pixel.

The above row-major serialization scheme is simple, however, there are some disadvantages. If the image is displayed and the important part of the image is located close to the bottom of the image, then all image data must be processed before the important part of the image is actually displayed. This may be significant if the image data are transported over a slow network. One possible solution is to code the image in an interleaved way. We set some fixed interleave factor, e.g. 8. First, we serialize the pixels whose x or y coordinates are multiples of 8. That is, we store pixels: `(x,y) = (0,0), (8,0), (16,0), (24,0), ..., (0,8), (8,8), (16,8), (24,8), ...`. The above pixels represent 1/64 of the image data, however, the data may be used to display a coarse version of the image. The next pass stores all pixels whose coordinates are multiples of 4 (of course, we do not duplicate the pixels we already stored in the previous pass). Thus, pixels `(x,y) = (4,0), (12,0), (20,0), (28,0), ..., (0,4), (4,4), (8,4), (12,4), (16,4), (20,4), (24,4), (28,4), ..., (4,8), (12,8), (20,8), (28,8), ...` are stored. The second pass is 3/64 of the image data, and combined with the first pass, it produces the image with doubled pixel resolution. The passes are repeated with multiples of 2 and 1, where the final image is completely stored/restored.

The example above started form interleave factor of 8. If we started with interleave factor 1, we would indeed store the image in the basic row-major order. On the contrary, if we started from a higher interleave factor, we would allow more passes and more steps in the quality of the restored image. This may be important if the image is big. The required function is supposed to convert the image files among different setups of interleave factors:

```
bool recodeImage ( const char * srcFileName,
                   const char * dstFileName,
                   int          interleave,
                   uint16_t     byteOrder );
```

`srcFileName`
: is an ASCIIZ string denoting the name of the source image file. The function is supposed to read the source image, however, it cannot modify it.

`dstFileName`

is an ASCIIZ string denoting the name of the resulting image file. The function creates/overwrites the resulting file and stores the updated image into the file.

`interleave`

is the required interleave factor of the destination file. The value must be one of the following values: 1, 2, 4, 8, 16, 32, or 64.

`byteOrder`

is either ENDIAN_LITTLE or ENDIAN_BIG (the constants are declared in the attached archive). The function creates the resulting image file using the byte order from that parameter. This parameter will be always ENDIAN_LITTLE in the mandatory tests (i.e. it matches the HW your program runs on). Both endianities will be used in the optional test. Moreover, there are few calls in the "Invalid input test" where the parameter is neither ENDIAN_LITTLE, nor ENDIAN_BIG. Clearly, these calls must result in an error.

return value

`true` to indicate success, `false` to report a failure. The following shall be considered a failure:

- file or I/O related problems (cannot read/write, file does not exist, ...),
- invalid input file format (invalid header, invalid pixel format, not enough image data, too many bytes in the file, ...),
- the interleave factor parameter is invalid,
- byteOrder parameter is neither ENDIAN_LITTLE nor ENDIAN_BIG.

The function creates the resulting file based on the source image and parameters:

- little/big endian format parameter determines multi-byte values encoding. The big/little endian format of the input and output image does not have to be the same in the optional tests, moreover, the encoding may differ from the HW your program runs on. Your implementation must explicitly handle/convert endianity to pass the optional test,
- resulting image size is determined by the source image size,
- resulting image pixel format is identical to the source image pixel format,
- resulting image interleave factor is given in the parameter,
- image data in the image file are modified such that the source and destination image would be displayed "the same".

The image file format is very simple. The file starts with a fixed size header that is followed by the image data. The header has the following structure:

```
offset   size         description
+0       2B           endianity (0x4949 little endian, 0x4d4d big endian)
+2       2B           image width
+4       2B           image height
+6       2B           pixel format
+8       ??           image data (pixels)
```

- Little/big endian identifier is the first field in the header. The field describes the byte order of two byte values stored in the file. The actual values (0x4949 and 0x4d4d) are symmetric, thus the field can be correctly read by any platform. This field will be set to little endian in the mandatory tests (little endian is the platform your program runs on). However, the value may be either little endian or big endian in the optional tests. (Actually, there are few invalid inputs in the mandatory tests where the endianity is set to a random value. Obviously, such inputs must be rejected).

- Width and height fields denote the dimensions of the image. The values must be non-zero.
- Pixel format describes the coding of individual pixels. The value is composed from individual bits:

```
    bit  15              8 7 6 5  4 3 2  1 0
         0 0    ...      0 I I I  B B B  C C
```

Each pixel may consist of several channels. Bits 1 and 0 denote the number of channels per pixel:

```
    00 - 1 channel: black/white, or shades of gray
    01 - invalid combination
    10 - 3 channels = RGB
    11 - 4 channels = RGBA
```

Each channel value is coded in the given number of bits:

```
    000 -  1 bit per channel
    001 -  invalid combination
    010 -  invalid combination
    011 -  8 bits per channel
    100 - 16 bits per channel
    101 \
    110  | invalid combination
    111 /
```

Interleave factor is determined by bits 7 to 5:

```
    000 -  1 (not interleaved)
    001 -  2
    010 -  4
    011 -  8
    100 - 16
    101 - 32
    110 - 64
    111 - invalid combination
```

Bits 8-15 are not used, they must be set to zero. A few examples:

```
   0x006c = 0b01101100 - interleave factor 8, 1 channel per pixel, 8 bits per
channel
   0x000e = 0b00001110 - interleave factor 1, 3 channels per pixel, 8 bits
per channel
   0x00b3 = 0b10110011 - interleave factor 32, 4 channels per pixel, 16 bits
per channel
```

Notes:

- Pay special attention to the file I/O. The testing environment tests your implementation, it tries nonexistent files, unreadable files, files with invalid contents, ...
- Do not assume anything about file names. There are no explicit restrictions on file names. The important thing is: can the file be opened/read/written.
- You may use either C or C++ file interface, the choice is free.
- There is a set of input and corresponding result images in the attached archive. Further, there is a source file with a sample test `main` that calls your implementation to convert the sample input files and compares the results with the reference. You may use the example source file as a basis of your implementation. There are parts of shared code, the code is placed in conditional compile blocks. If these conditional compile blocks are preserved, the source can be submitted to Progtest.
- Use bit operation (&) and bit shifts (<< and >>) to handle pixel format field in the header.
- Little/big endian applies to the fields in the header, moreover, it applies to the 16 bit values in the image data (if the actual pixel format is 16 bits per channel).
- Do not develop one huge function. Instead, divide the problem into subproblems, use functions/classes to implement the subproblems. A reasonable solution reads the source into memory, decodes the pixels, modifies the pixels, and writes the contents back into the destination file. There is enough memory to load the entire image.
- There are fixed width fields in the header (e.g. 2 byte integers). Do not use `int` type to handle these values, the width of `int` data type is not guaranteed. Instead, use fixed width data types declared in `cstdint` header, such as `int16_t` and `uint16_t`.
- Mandatory tests use 8 bits / channel format, moreover, only little endian is used (i.e. it matches the processor architecture). Your implementation passes all mandatory tests if:
  - it handles 8 bits / channel + little endian, and
  - it reports an error for all other bits/channel and endian combinations.
- The test "16 bit + endianity" is an optional test. If your implementation does not pass the test, there will be a certain malus to the overall result. The fact it is optional means that your implementation may be awarded non-zero points even if your implementation completely fails the optional test. (If it was a mandatory test, then a failure to pass the test would automatically result in 0 points.)
- The last test is focused on 1 bit / channel. It is a bonus test, i.e. you will be awarded extra points over the nominal 100% if your program passes this test. The processing of 1 bit values means that the bytes read from the input file must be decomposed into individual bits. Similarly, the bits written to the output file must be combined into bytes. The bits in the bytes are to be stored in the LSB to MSB order (least significant bit to most significant bit). If the total number of bits in the image is not a multiple of 8, then the last byte is to be padded with zero bits.
- It is recommended to start with the basic variant (8 bits/channel + little endian). Once it works, extend the solution to work with the other bit depths. There is a hidden benefit of this

continuous development - you have to design function/method interfaces to be extensible. You may need to re-design the function several times, however, the experience is invaluable.