Problem Set 5: QR Code

Goals

1. Understand the representation of numbers in computer memory.
2. Acquire work with a two-dimensional field.
3. Create custom features as specified.
4. Learn to terminate functions using different return values at different input parameters.

QR Code

You have probably already encountered the QR code. It is a 2D field designed for visual representation of data. The QR (Quick Response) code was created in Japan in 1994 for the automotive industry. It later spread to other areas, including everyday life.

To obtain data from the QR code, it is necessary to use a scanner or have an application installed on the phone that provides this functionality.

in this assignment *Fortunately,* we will not program the entire functionality of the application that scans the QR code , but we will show you the basic principles for working with 2D fields in this code.

FIG. 1: Sample QR code, source: https://en.wikipedia.org/wiki/QR_code

The QR code consists of black squares placed in a grid, which are on a white background. This allows the code to be read for devices such as a camera or some special scanning device. We will be particularly interested in the data part of this code.

Standard modes are used to store data in the QR code, which allows you to store numbers, alphanumeric characters, binary data (bytes), kanji, but also others if extensions are used. We will work with binary data.

Your task is to program the input so that it can convert short text into bytes and then group them into blocks and vice versa.

Use a basic ASCII table to encode characters from a string to a binary.

Fields of type will be used to store bytes `bool`where `1 == true` a `0 == false`.

Blocks consist of vertically stored bytes. The bytes are stored sequentially from left to right, if the block width allows. Only then are they saved in a new "row". The index of the new "row" for the block has an *offset* of the size corresponding to the byte height (8 bits).

Example

Let's encode a string `"Ahoj!"`. The following 6 characters are required: `'A'`, `'h'`, `'o'`, `'j'`, `'!'`, `'\0'`. We encode the characters into a binary system based on their values in an ASCII table. **We also encode the terminator** , because this character also has its value in the ASCII table. The values of these characters in the ASCII table are as follows:

SIGN	Value - decimal system	Value - binary (8 bits)
`'A'`	`65`	`01000001`
`'h'`	`104`	`01101000`
`'o'`	`111`	`01101111`
`'j'`	`106`	`01101010`
`'!'`	`33`	`00100001`
`'\0'`	`0`	`00000000`

> **Note :** Under the number `0`we understand value `false`and under the number `1`we understand value `true`.

And this will be your task: Based on the ASCII value of the character, calculate what value this character has in the binary system. This creates a *byte* . So if you encode a group of characters (a string), you generate a two-dimensional field in which there is one on each line *byte* .

When we get a group of bytes (see table), we can create "blocks". Blocks have a defined number of columns and "rows" (offset * 8), and is set so that all characters fit in the blocks. Character codes (bytes) are stored in blocks vertically, first from left to right, then according to the offset we move lower and again we store bytes vertically, from left to right. If extra blocks are available, we store them in them `false`(character codes `'\0'`).

Since the number of "rows" and columns in blocks is predefined, let's say we have `2` "riadky" a `3`columns. Let's organize the characters of our chain `"Ahoj!"`in the order in which they will be stored in the blocks:

```
'A'  'h'  'o'
'j'  '!' '\0'
```

The byte of each character is stored vertically. So for `2` "riadky" (2*8) a `3`the column will be a string `"Ahoj!"`stored in blocks as follows:

```
0    0    0
1    1    1
0    1    1
0    0    0
0    1    1
0    0    1
0    0    1
1    0    1

0    0    0
1    0    0
1    1    0
0    0    0
1    0    0
0    0    0
1    0    0
0    1    0
```

Task 1: (De) Character encoding

Create a function `void encode_char(const char character, bool bits[8])`with two parameters:

- `const char character`- A character whose ASCII value is encoded from decimal to binary
- `bool bits[8]`- Field of values `true`or `false`. Its size is `8`, because each character can be encoded to `8`bits.

The function **does return** not any value, but populates the field `bits`values `true`or `false`. The field will contain an ASCII character value entry `character`in the binary system. It is true that `1 == true` a `0 == false`.

Create a function `char decode_byte(const bool bits[8])`with parameter:

- `const bool bits[8]`- Field of values `true`or `false`. Its size is `8`, because each character can be encoded to `8`bits.

The function **returns a** character that is written in the ASCII table under the same value (in decimal) as it is written in the field `bits`in the binary system.

Example of using functions

```
bool bits1[8];
encode_char('A', bits1);
for(int i = 0; i < 8; i++){
    printf("%d", bits1[i]);
}
printf("\n");
// prints: 01000001

bool bits2[8] = {0,1,0,0,0,0,0,1};
printf("%c\n", decode_byte(bits2));
// prints: A
```

Task 2: (De) String encoding

Create a function `void encode_string(const char string[], bool bytes[strlen(string)+1][8])`with two parameters:

- `const char string[]`- A string that needs to be encoded into bytes
- `bool bytes[strlen(string)+1][8]`- A two-dimensional array that contains 1 byte (8 bits) on each line

The function **does return** not any value, but populates the field `bytes`values `true`or `false`. The field will contain the entry of the ASCII values of the characters from the string `string`in the binary system **including the terminator** . It is true that `1 == true` a `0 == false`.

Create a function `void decode_bytes(const int rows, bool bytes[rows][8], char string[rows])`with three parameters:

- `const int rows`- The number of rows in the field `bytes`and the number of characters in the string `string` **including terminator**
- `bool bytes[rows][8]`- A two-dimensional array that contains 1 byte (8 bits) on each line
- `char string[rows]`- The string that needs to be created by decoding the field data `bytes`

The function **returns** no value, but populates the string `string`characters by decoding the data in the field `bytes`. Pole `bytes`contains 1 byte (8 bits) with values on each line `true`or `false`, which expresses the ASCII value of characters in the binary system, **including the terminator** . It is true that `1 == true` a `0 == false`.

Example of using functions

```c
char* text = "Hello, how are you?";
const int len = strlen(text);
bool bytes1[len+1][8];
encode_string(text, bytes1);
for(int j = 0; j <= len; j++){
    printf("%c: ", text[j]);
    for(int i = 0; i < 8; i++){
        printf("%d", bytes1[j][i]);
    }
    printf("\n");
}
// prints:
// H: 01001000
// e: 01100101
// l: 01101100
// l: 01101100
// o: 01101111
// ,: 00101100
//  : 00100000
// h: 01101000
// o: 01101111
// w: 01110111
//  : 00100000
// a: 01100001
// r: 01110010
// e: 01100101
//  : 00100000
// y: 01111001
// o: 01101111
// u: 01110101
// ?: 00111111
// : 00000000

bool bytes2[7][8] = {
    {0,1,0,0,1,0,0,0},
    {0,1,1,0,0,1,0,1},
    {0,1,1,0,1,1,0,0},
    {0,1,1,0,1,1,0,0},
    {0,1,1,0,1,1,1,1},
    {0,0,1,0,0,0,0,1},
    {0,0,0,0,0,0,0,0}
};
char string[7];
decode_bytes(7, bytes2, string);
printf("%s\n", string);
// prints: Hello!
```

Task 3: (De) Coding blocks

Create a function `void bytes_to_blocks(const int cols, const int offset, bool blocks[offset*8][cols], const int rows, bool bytes[rows][8])`with parameters:

- `const int cols`- Number of columns for blocks
- `const int offset`- Number of row groups for blocks (adjusts the number of rows for blocks)
- `bool blocks[offset*8][cols]`- Two-dimensional array for blocks with a precisely defined number of columns and rows
- `const int rows`- Number of lines (string length including terminator)
- `bool bytes[rows][8]`- A two-dimensional array with code bytes for string characters

The function **does return** not any value, but populates the field `blocks`byte blocks that contain the codes of each character of the string. It is true that `1 == true` a `0 == false`.

Create a function `void blocks_to_bytes(const int cols, const int offset, bool blocks[offset*8][cols], const int rows, bool bytes[rows][8])`with parameters:

- `const int cols`- Number of columns for blocks
- `const int offset`- Number of row groups for blocks (adjusts the number of rows for blocks)
- `bool blocks[offset*8][cols]`- Two-dimensional array for blocks with a precisely defined number of columns and rows
- `const int rows`- Number of lines (string length including terminator)
- `bool bytes[rows][8]`- A two-dimensional array with code bytes for string characters

The function **does return** not any value, but populates the field `bytes`codes of individual characters of the string. It is true that `1 == true` a `0 == false`.

Example of using functions

```
int length = 4+1, cols = 3, offset = 2;
bool bytes1[4+1][8] = {
    {0,1,0,0,0,0,0,1},
    {0,1,1,0,1,0,0,0},
    {0,1,1,0,1,1,1,1},
    {0,1,1,0,1,0,1,0},
    {0,0,0,0,0,0,0,0}
};
bool blocks1[offset*8][cols];
bytes_to_blocks(cols, offset, blocks1, length, bytes1);
for(int j = 0; j < offset*8; j++){
    for(int i = 0; i < cols; i++){
        printf("%d ", (blocks1[j][i] == true) ? 1 : 0);
    }
    printf("\n");
    if(j % 8 == 7){
        printf("\n");
    }
}
// prints:
// 0 0 0
// 1 1 1
// 0 1 1
// 0 0 0
// 0 1 1
// 0 0 1
// 0 0 1
// 1 0 1
//
// 0 0 0
// 1 0 0
// 1 0 0
// 0 0 0
// 1 0 0
// 0 0 0
// 1 0 0
// 0 0 0
```

```c
bool blocks2[2*8][3] = {
    {0,0,0},
    {1,1,1},
    {0,1,1},
    {0,0,0},
    {0,1,1},
    {0,0,1},
    {0,0,1},
    {1,0,1},
    {0,0,0},
    {1,0,0},
    {1,0,0},
    {0,0,0},
    {1,0,0},
    {0,0,0},
    {1,0,0},
    {0,0,0}
};
bool bytes2[length][8];
blocks_to_bytes(3, 2, blocks2, length, bytes2);
for(int j = 0; j < length; j++){
    for(int i = 0; i < 8; i++){
        printf("%d", bytes2[j][i]);
    }
    printf("\n");
}
// prints:
// 01000001
// 01101000
// 01101111
// 01101010
// 00000000
```