Programming Paradigms
Caucasus University – Spring 2022
Assignment 1 (10 points)

– Deadline: 23:59 on Friday 30 September 2022 –

Learning Goals

This assignment covers topics from the first few lectures and the first lab. You will be building your skills with:

- editing, compiling, testing, and debugging C programs under Unix
- writing code that manipulates bits and integers using the C bitwise and arithmetic operators
- C-strings (both raw manipulation and using string library functions)

Part I
Bits and Bytes

1 Hamming (7, 4) (3 points)

Hamming (7, 4) is a simple technique to implement error correcting codes (ECC). ECC is often used to safely store data and automatically correct mistakes. (e.g. ECC RAM: `https://en.wikipedia.org/wiki/ECC_memory`)

Hamming (7,4) algorithm takes 4 bits of data, adds 3 parity bits and wraps them in 7 bits. Because of this we need 1 byte to encode 4 bit. To encode a full byte we need 2 bytes. The parity bits are calculated using data bits. Parity bits are used to detect and if possible, correct errors. Hamming (7, 4) can detect and correct 1 flipped bit. In case more than 1 bit is flipped, it can detect errors, but not correct them.

For example we have 4 bit data, that we want to encode: where $d1, d2, d3, d5$ are data bits.

index	7	6	5	4	3	2	1	0
	0	0	0	0	d1	d2	d3	d4

Hamming (7, 4) will add 3 parity bits:

index	7	6	5	4	3	2	1	0
	0	p1	p2	d1	p3	d2	d3	d4

We calculate parity bits by xor-ing data bits:

```
p1 = d1 ^ d2 ^ d4
p2 = d1 ^ d3 ^ d4
p3 = d2 ^ d3 ^ d4
```

As you can see, 3 parity bits have overlapping data bits.

Let's see how we can encode 4 bit data: 0110 We get following parity bits:

index	7	6	5	4	3 (d1)	2 (d2)	1 (d3)	0 (d4)
	0	0	0	0	0	1	1	0

```
p1 = 1
p2 = 1
p3 = 0
```

index	7	6 (p1)	5 (p2)	4 (d1)	3 (p3)	2 (d2)	1 (d3)	0 (d4)
	0	1	1	0	0	1	1	0

To decode this information, we need to recalculate parity bits. Check that they are same and then extract the data bits.

Now, let's say there was a mistake and d1 bit flipped. This will cause problems in p1 and p2 parity bits. When we recalucalte them, we'll notice that they are not the same as before. Since p1 and p2 have problems we see that d1 is overlapping with them and it needs fixing. The easiest way to calculate the index of the wrong bit is following:

- start with index 7 (the bit that we don't use)
- check the parity bit with it's corresponding data bits by xor-ing them together:

  ```
  p1 ^ d1 ^ d2 ^ d4 == 0
  p2 ^ d1 ^ d3 ^ d4 == 0
  p3 ^ d2 ^ d3 ^ d4 == 0
  ```

- and for every mistake we reduce the index, for p1 by 1, for p2 by 2 and for p3 by 4. In our example, d1 is flipped, problem is in p1 and p2 and error index will be: 7 - 1 - 2 = 4 (the bit for d1)

You can reed more about Hamming (7,4) here: `https://en.wikipedia.org/wiki/Hamming(7,4)`

Your task is to implement a hamming (7,4) data encoding/decoding algorithm:

- The function to find the index of an error:

  ```
  static uint8_t hamming74_error_idx(const uint8_t hamming_code)
  ```

- The function to encode 4 bits, it takes 4 bits of data (nibble) and returns a hamming 7,4 encoded byte:

```
uint8_t hamming74_encode(const uint8_t data)
```

- This function takes hamming 7,4 encoded byte, checks for the error, fixes the error bit if possible, and returns the decoded 4 bit data. if it fails it should return 0xFF.

```
uint8_t hamming74_decode(uint8_t hamming_code)
```

- This function takes a 1 byte data, splits it into 2 pieces, encodes each data piece using hamming 7,4 algorithm, and writes 2 bytes into the buffer.

```
void hamming74_encode_pair(const uint8_t data, uint8_t *buffer)
```

- This function reads 2 bytes from the input, decodes the data, and writes it in buf. If it fails, returns 0xFF.

```
uint8_t hamming74_decode_pair(const uint8_t *buffer)
```

Why do we return 0xFF in case of an error?
You are provided with some checks in hamming,c file. You should also write some of your own test cases.

2 UTF-8 (3 points)

A C char is a one-byte data type, capable of storing 28 different bit patterns. The ASCII encoding standard (man ascii) establishes a mapping for those patterns to various letters, digits and punctuation, but is limited to 256 options, so only a subset are included. Unicode is ASCII's more cosmopolitan cousin, defining a universal character set that supports glyphs from a wide variety of world languages, both modern and ancient (Egyptian hieroglyphics, the original emoji?). This large number of characters requires a different system than just a single char, however. For this part of the assignment, you will implement the most popular Unicode encoding, UTF-8, which according to recent measurements, is used by over 93

Unicode maps each character to a code point, which is a hexadecimal number. There are over 1 million different code points in the Unicode standard! A character's code point is commonly written in the format U+NNNN, which signifies the hexadecimal value 0xNNNN.

However, the Unicode standard does not specify how to best encode these code points in binary. There are a variety of different possible Unicode encodings which specify, given a code point, how to actually store it. Why multiple encodings? Each may have a different priority, whether it is compatibility with other non-Unicode encodings, the locale it is intended for, space efficiency, etc. Some questions that may have different answers depending on the encoding are: should smaller code points take up the same amount of space as larger code points? If not, how can we tell how long the encoding is? And is it possible to preserve compatibility with other encodings, such as ASCII?

One of the most popular Unicode encodings is UTF-8. UTF-8 is popular partially because it preserves backwards compatibility with ASCII - in fact, it was designed such that ASCII is a subset of UTF-8, meaning that the characters represented in the ASCII encoding have the same encoding in ASCII and UTF-8.

The UTF-8 encoding represents a code point using 1-4 bytes, depending on the size of the code point. If it falls in the range U+0000 to U+007F, its UTF-8 encoding is 1 byte long. If it falls in the range U+0080 to U+07FF, its UTF-8 encoding is two bytes long. And if it falls in the range U+0800 to U+FFFF, its UTF-8 encoding is 3 bytes long (there is a 4 byte range, but we'll be ignoring that for this assignment).

The way UTF-8 works is it splits up the binary representation of the code point across these UTF-8 encoded byte(s). Code points from U+0000 to U+007F use at most 7 bits (these are called its "significant bits"), so the UTF-8 encoding is one byte with its most significant bit0, and its remaining 7 bits as the seven significant bits from the code point. Code points from U+0080 to U+07FF have at most 11 significant bits, so the first byte of the UTF-8 encoding is a leading 110 (we'll see why later), followed by the 5 most significant code point bits. The second byte of this UTF-8 encoding is a leading 10, followed by the remaining 6 significant code point bits. Code points from U+0800 to U+FFFF have at most 16 significant bits, so the first byte of the UTF-8 encoding is a leading 1110, followed by the 4 most significant code point bits. The second byte of this UTF-8 encoding is a leading 10, followed by the 6 next most significant code point bits. The third byte of this UTF-8 encoding is a leading 10, followed by the final 6 significant code point bits.

Here is a table containing the design described above. For each byte layout, the 0 and 1 bits are fixed for all representations. The xxx bits store the binary representation of the code point.

Code Point Range	**Significant Bits**	***UTF-8 Encoded Bytes (Binary)***
U+0000 to U+007F	7	0xxxxxxx
U+0080 to U+07FF	11	110xxxxx 10xxxxxx
U+0800 to U+FFFF	16	1110xxxx 10xxxxxx 10xxxxxx

As you can see, the one-byte sequence stores 7 bits of the code point, the two-byte sequence stores 11 bits (5 bits in first byte and 6 in the other), and the three-byte stores 16 bits (divided 4, 6, and 6). The different UTF-8 byte lengths are determined based on how many bytes are needed to store all the bits of the code point (1 for up to 7 significant bits, 2 for up to 11 significant bits, and 3 for up to 16 significant bits).

In a single-byte sequence, the high-order bit is always 0, and the other 7 bits are the value of the code point itself. This is so the first 128 Unicode code points use the same single byte representation as the corresponding ASCII character!

For the multi-byte sequences, the first byte is called the leading byte, and the subsequent byte(s) are continuation bytes. The high-order bits of the leading byte indicate the number of total bytes in the sequence through the number of 1s (for instance, the leading byte of an encoding for a code point from U+0080 to U+07FF starts with 110, indicating the entire encoding is 2 bytes long, since there are two 1s). The high-order bits of a continuation byte are always 10. The bit representation of the code point is then divided across the low-order bits of the leading and continuation bytes.

Example (paraphrased from Wikipedia UTF-8 page)

Consider encoding the Euro sign, €.

1. The Unicode code point for € is U+20AC.
2. The code point is within the range U+0800 to U+FFFF and will require a three-byte sequence. There are 14 significant bits in the binary representation of 0x20AC.
3. Hex code point 20AC is binary 00100000 10101100. Two leading zero bits of padding are used to fill to 16 bits. These 16 bits will be divided across the three-byte sequence.
4. Let's calculate the leading byte. High-order bits are fixed: three 1s followed by a 0 indicate the three-byte sequence. The low-order bits store the 4 highest bits of the code point. This byte is 11100010. The code point has 12 bits remaining to be encoded.

5. Next, let's calculate the first continuation byte. High-order bits are fixed 10, low-order bits store the next 6 bits of the code point. This byte is 10000010. The code point has 6 bits remaining to be encoded.

6. Finally, let's calculate the last continuation byte. High-order bits are fixed 10, low-order bits store the last 6 bits of the code point. This byte is 10101100.

7. The three-byte sequence this results in is 11100010 10000010 10101100, which can be more concisely written in hexadecimal, as e2 82 ac. The underscores indicate where the bits of the code point were distributed across the encoded bytes.

You task is to write the to_utf8 function that takes a Unicode code point and constructs its sequence of UTF-8 encoded bytes.

```
int to_utf8(unsigned short code_point, unsigned char utf8_bytes[])
```

The to_utf8 function has two parameters; an unsigned short code_point and a byte array utf8_bytes. The function constructs the UTF-8 representation for the given code point and writes the sequence of encoded bytes into the array utf8_bytes. The first byte (e.g. leading byte) should go at index 0, and the remaining bytes (if any, e.g. continuation bytes) should follow at index 1, etc. The utf8_bytes array is provided by the client and is guaranteed to be large enough to hold a full 3 bytes, although only 1 or 2 may be needed. While we will talk about C arrays in more depth later, it turns out that if you pass an array as a parameter, modifying the elements of that array parameter will modify the elements of the original array passed in, so the function above can pass back the byte(s) it creates. The function returns the number of bytes written to the array (either 1, 2, or 3).

You will implement this within the provided utf8 program, which takes one or more code points and displays the UTF-8 hex encoding and corresponding character glyph for each code point using your to_utf8 function. Here is a sample run:

```
$ utf8 0x41 0x20ac
U+0041   Hex: 41         Character: A
U+20AC   Hex: e2 82 ac   Character: €
```

This task is excellent practice with constructing bitmasks and applying bitwise ops and standard techniques to extract/rearrange/pack bits.

Part II
C Strings

3 scan_token (4 points)

With the goal of making an improved function that performs the same type of tokenization as strtok without its awkward design, you are to write the function scan_token. The required prototype for scan_token is:

```
bool scan_token(const char **p_input, const char *delimiters,
                char buf[], size_t buflen)
```

The function scans the input string to determine the extent of the token using the delimiters as separators and then writes the token characters to buf, making sure to terminate with a null char. The function returns **true** if a token was written to buf, and false otherwise.

Specific details of the function's operation:

- Your implementation of scan_token should take the same general approach as strtok, meaning it can (and should!) use the handy < *string.h* > functions such as strspn and strcspn, but it should not replicate the bad parts of its design, which is to say no static variables, no weird use of the input argument to pass information across a sequence of calls, and should not destroy the input string.
- The function separates the input into tokens in the same way that strtok does: it scans the input string to find the first character not contained in delimiters. This is the beginning of the token. It scans from there to nd the first character contained in delimiters. This delimiter (or the end of the string if no delimiter was found) marks the end of the token.
- Note that the parameter p_input is a char ** . This is a pointer argument that is being passed by reference. The client passes a pointer to the pointer to the first char of the input string. The function will update the pointer held by p_input to point to the next character following the token that was just scanned.
- buf is a fixed-length array to store the token and buflen is the length of the buffer. scan_token should not write past the end of buf. If a token does not fit in buf, the function should write *buflen - 1* characters into buf, write a null byte in the last slot, and the pointer held by p_input should be updated to point to the next character following the buflen - 1 characters in the token. In other words, the next token scanned will start at the first character that would have overflowed buf.

Consider this sample use of scan_token:

```c
const char *input = "super-duper-awesome-magnificent";
char buf[10];
const char *remaining = input;

while (scan_token(&remaining, "-", buf, sizeof(buf))) {
    printf("Next token: %s\n", buf);
}
```

Running the above code produces this output:

```
Next token: super
Next token: duper
Next token: awesome
Next token: magnifice
Next token: nt
```

Write your implementation of scan_token in the scan_token.c file. You can test it using our provided tokenize.c program:

```
$ ./tokenize " -" "hello I am a C-string"
Tokenized: { "hello" "I" "am" "a" "C" "string" }
```

Submitting

Once you're finished working, zip all source code and CMakelists.txt files together and upload them on teams.

Grading

Correctness: 80% Your code should compile without any errors or warnings. We will test your programms on different inputs and edge cases. For a full score your code should pass all tests (some exercises might already have predefined tests).

Code Quality: 20% We expect your code to be clean and readable. We will look for descriptive names, defined constants (not magic numbers!), and consistent layout. Be sure to use the most clear and direct C syntax and constructs available to you.