0512-1820 Fall 2024
Home Assignment **5**

Yehonatan Kohavi - kohaviy@mail.tau.ac.il

Due date: 29/01/2025

In this assignment you will get hands-on experience with Modular Programming, Bit Manipulation & File handling in the C programming language.

Submission Guidelines

- Due date is **29/01/2025**
- Submission file is: **hw5_ID1_ID2.zip**

  The zip should include the following files:

  - `error_types.h`
  - `main.c`
  - `parser.c`
  - `parser.h`
  - `encrypt.c`
  - `encrypt.h`
  - `algs.c`
  - `algs.h`
  - `file_utils.c`
  - `file_utils.h`
  - `censor.c`
  - `censor.h`

  E.g. for a pair of students with IDs 123456789 and 987654321 the zip file should be named:

  `hw5_123456789_987654321.zip`

And for example, one of the source file should be named `main.c` This is merely an example.

- Do not forget to zip all of the required files as mentioned above. No internal directories and no other irrelevant files should be included in your zip, only the **source & header files** we requested!
- Please use the standard C libraries only! No other libraries should be installed or used unless you have been specifically instructed.
- Do not break the interface! don't change the signatures of the functions in the header files!
- For any issues & questions you can use the forum in moodle, consult with your peers and also use google.

Warnings

- **Warning 1**: If your code doesn't compile you will get 0, regardless of the amount of work you've put into coding.
- **Warning 2**: Do not cheat or use any automatic code generator to complete this home work! It is for your own good. Caught cheaters will be **punished**!
- **Warning 3**: Your code would be tested automatically. Failing to comply with naming conventions will result in **points deduction**!
- **Warning 4**: Mismatches during outputs comparison will cause failing tests and therefore lead to **points deduction**!
- **Warning 5**: The submitted header files would be **overridden** with the skeleton one (except `(`algs.h)), hence do not change the prototypes (definitions & signatures)! However you are still expected to submit them!

Suggested workflow

1. Generate a new **git** repository for this home assignment / Add a new directory to your existing HW git repository.
2. **Download the attached files** from moodle into your repository.
3. **Open & read the given files** for this home assignment.
4. **Read this entire document** before writing a single line of code.
5. Write some basic **tests** to make sure your code will work (TDD).
6. Let the **coding** begin!
   Don't forget to **commit & push your progress** in git for version control & collaboration.
7. Make sure your **code compiles** in the testing environment.
8. **Add more tests** with all of the corner cases you could think.
9. Make sure your **code runs properly** and correctly, and that all of your tests pass. **Debug your code** and fix it accordingly (you might find "rubberducking" pretty useful).
10. **Re-read this document** to make sure you have not forgotten anything.
11. **Check the moodle for any updates** regarding this assignment in the Q&A forum and in the Announcements forum.
12. **Zip your code** according to the submission guidelines above.
13. **Unzip your code** and repeat steps 7 & 9 to make sure everything is OK, and that the zip file and its content comply with the naming conventions specified above.
14. **Submit the zip file** to moodle.
15. Congratulations! you have **completed** the home assignment!

Good luck!

Exercise 1 - Encryption, Decryption & Censorship

This exercise involves developing a **command-line tool** for file encryption, decryption, and content censorship. You will be provided with a set of C source and header files (skeleton) outlining the project structure. Your task is to complete the missing implementations in these files.

The program would be divided into to the following modules:

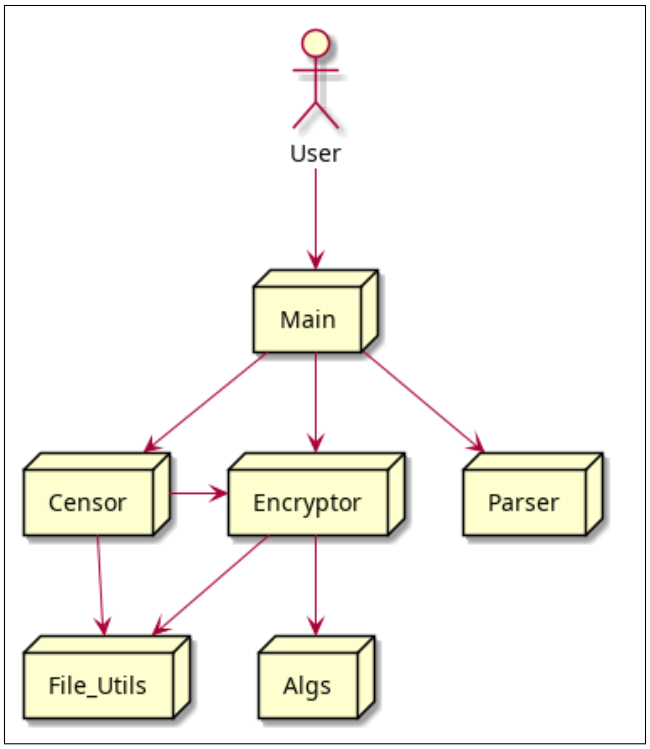

Figure 1: Components Diagram

Error Types

- **Relevant Files**: `error_types.h`
- **Purpose**: Defines an enumeration of error codes for different types of errors (e.g., argument parsing errors, file I/O errors, encryption errors).
- **Usage**: Each source file should include `error_types.h`. In your implementation you should return a suitable error code as the return value (of the functions to be completed) according to the execution sequence of the functions.
- The error-codes (the names are self-explanatory):

```
typedef enum {
      OK = 0,

      // parser errors
      ERR_NUM_ARGS,
      ERR_MISSING_ARG,
      ERR_UNKNOWN_FLAG,
      ERR_BAD_OP,
      ERR_BAD_ENC_TYPE,

      // MISC. errors
      ERR_NULL_PTR,
      ERR_MEMORY,
      ERR_FILE,
      ERR_BAD_FUNC_ARG,

} error_types;
```

For clarification:

- Parser errors: should be used as return values for error handling within the parser module.
  - `NUM_ARGS` - upon receiving different number of arguments than expected (according to the input operations).
  - `MISSING_ARG` - upon missing argument after a flag.
  - `UNKNOWN_FLAG` - upon receiving an unknown flag.
  - `BAD_OP` - upon receiving an unknown operation. Available operations are listed below.
  - `BAD_ENC_TYPE` - upon receiving an unknown encryption type. Available encryption types are listed below.
- Misc errors: these errors can be used across all of the existing modules (including parser if needed).
  - `NULL_PTR` - upon receiving a NULL pointer as argument in any of your functions.
  - `MEMORY` - in case memory allocation failed.
  - `FILE` - in case file handling failed.
  - `BAD_FUNC_ARG` - upon receiving a bad argument (unexpected) in any of your functions which makes the code sequence bad. `NULL_PTR` precedes this case!

Main

- **Relevant Files**: `main.c`
- **Purpose**:
  - The main entry point of the program.
  - Receives the command-line arguments.
  - Invokes `Parser` to parse them.
  - Invokes the execution of the chosen operation (encryption, decryption, or censorship).
- **Implementation**:
  - `Main` check if need to print help (no arguments are `"-h"` flag only).
  - `Main` calls to `Parser`'s `parse_args()` to parse the arguments and build a command variable.
  - `Main` dispatches an execution of that command in `process_operation()` via the other modules according to the input operation type.

Parser

- **Relevant Files**: `parser.h`, `parser.c`
- **Purpose**:
  - Provide `print_help()`: displaying usage information.
  - Implementation of `parse_args()` function for parsing command-line arguments.
- **Usage**:
  - Print the usage information for the command-line tool (list of available options and their descriptions).
    The flags:

```
Usage:
  -h             help: print this description,
  -p        operation: enc/dec/censor,
  -t  encryption type: 0/1/2/3,
  -i            input: file path,
  -o           output: file path,
  -b        blacklist: file path,
Please try again
```

This function is already **provided** to you.

- **Implement** `parse_args()`:
  I. Parse command-line arguments using `argc` and `argv`.
  II. Validate the arguments for correctness (e.g., check for required arguments, valid operation types, valid encryption types, etc.).
  III. Populate the `command_t` structure with the parsed values.
  IV. There is no significance to the order of the flags.
  V. Assume input arguments are flags following by their values.
  Example for input arguments:

```
prog.exe -i in.txt -p censor -b bl.txt -o out.txt -t 1
```

File Utils

- **Relevant Files**: `file_utils.h`, `file_utils.c`
- **Purpose**: Provide utility functions for file & memory operations.
- **Implementations**:
  - `allocate_buffer()`: Allocate memory for a buffer of the specified size (dynamically).
  - `write_data_to_file()`: Write the given data (buffer) to the specified output file.
  - `load_data_from_file()`: Read the contents of the specified input file into a dynamically allocated buffer.
  - `load_array_of_words()`: Read the blacklisted words from the specified file into an array of strings. Assume each word has maximal word size 256 chars (it is provided as a define in `file_utils.c`), and each word lies in its own separate line). There are no empty lines, but the blacklist file might be empty.

Encryptor

- **Relevant Files**: `encrypt.h`, `encrypt.c`
- **Purpose**: Provide operation for encryption and decryption of **files**.
- **Implementations**:
  - `encrypt_file()`:
    I. Reads the input file.
    II. Calls the appropriate encryption function (provided in `algs.h`) based on the `enc_type` parameter.

III. Writes the encrypted data to the output file.

IV. Returns error code according to the code sequence.

- `decrypt_file()`:

I. Reads the encrypted input file.

II. Calls the appropriate decryption function (provided in `algs.h`) based on the `enc_type` parameter. The decryption is the inverse operation of the encryption.

III. Writes the decrypted data to the output file.

IV. Returns error code according to the code sequence.

Algs

- **Relevant Files**: `algs.h`, `algs.c`
- **Purpose**: Provide the implementation for encryption and decryption to **buffers** of the different algorithms.
- **Implementations**:
  1. Your task is to implement each algorithm's (encryption type) functions encryption (and decryption if not symmetrical).
  2. Each function should have the following signature:

```
int alg_name(const unsigned char *data_in,
             unsigned int size_in,
             unsigned char *data_out,
             unsigned int size_out);
```

  3. Each function:
     I. Receives an input buffer and input buffer size, output buffer and its size.
     II. Validates parameters.
     III. Performs the operation according to the algorithm.
     IV. Returns error code according to the code sequence.
  4. You should also implement `get_size_out()` function which returns the size of the output buffer to be allocated according to the encryption algorithm.
- **The different encryption algorithms are given by in `alg.h` as follows**:

```
typedef enum {
      ENC_TYPE_NONE = 0,

      ENC_TYPE_FLIP_EVEN,
      ENC_TYPE_SWAP_3,
      ENC_TYPE_ROT_AND_CENTER_5,

      ENC_TYPE_LAST
} encrypt_t;
```

0. **None**: don't encrypt, i.e. Output buffer data is identical to the input buffer data.
1. **Flip_Even**: for each byte - flip the even bits.
   Example:

   ```
   0b10110011 -> 0b00011001, i.e. 0xB3 -> 0x19
   ```

2. **Swap_3**: for each pair of neighboring (unchanged) bytes - swap the 3 LSB of the first bytes with the 3 MSB of the second byte. If not even number of bytes - don't change the last byte.
   Example:

   ```
   0b01011-010 and 0b011-10001 -> 0b01011-011 and 0b010-10001
   i.e. 0x5A, 0x71 -> 0x5B, 0x51
   ```

   NOTE: Consider 2 bytes pairs [b1, b2] & [b3, b4]. Start on the first pair [b1, b2]. Then **DO-NOT** do the perform the manipulation on [b2, b3], as b2 was already changed, hence move to the next unchanged pair [b3, b4].
3. **Rotate_and_center_5**: for each byte - rotate left by 5 bits, and center the result by splitting into 2 bytes. The 5 LSBs of the first byte is the the upper 5 MSB of the rotated byte. The 3 MSB of the second byte are the 3 LSB of the rotated byte. Padding around with 0's.
   Example:

   ```
   Byte = 0b10101111 = 0xAF
   rotated left = 0b11110101 = 0xF5
   result = 0b00011110 0b10100000 = 0x1EA0
   ```

Censor

- **Relevant Files**: `censor.h`, `censor.c`
- **Purpose**: Provide function to read a file and generate 2 output files:
  - a (un-encrypted) censored file. The file name is the given output file name.
  - an encrypted (un-censored) file. The file name should be the same as the output file with `_enc.txt` suffix.

  For example if the given output file name is `out.txt` you should generate `out.txt` as the censored file, and `out_enc.txt` as the encrypted file.
- **Implementations**: Implement `censor_and_encrypt()`:

  I. Encrypt the data using the specified encryption method (uncensored).

  II. Read the blacklist file to obtain a list of words to be censored.

  III. Read the input file. Keep in mind a char buffer is not a string (remember to add the '\0').

  IV. Censor - replace the blacklisted words in the input data with a asterisks, e.g. for `word = it`

```
input txt =
"It is what it is! This iteration cannot make sence,
sitting under the tree might. IS IT CLEAR?"

Censored txt =
"** is what ** is! This iteration cannot make sence,
sitting under the tree might. IS ** CLEAR?"
```

  V. Dump the encrypted and censored data to the output file.

  VI. Returns error code according to the code sequence.

  As you can see if the word is a substring you shouldn't censor it. Only if its nested near the following special characters.

```
const char *special_chars = "\\\n\r\t ‘!()[]{}<>,.:;\"-_=+/?";
```

Instructions

For ease of development and collaboration we will separate the development into a few steps:

1. Understand the Project Structure:
    - Re-read this document and carefully review the provided files and their descriptions. Make sure you understand the overall program flow and the responsibilities of each module.
    - Refer to the `error_types.h` header for appropriate error codes to return in your functions.
2. Implement the Functions:
    - Do not break the interface! Do not change the header files nor any of the given prototypes/decelerations (the typedefs or functions signatures).
    - You should add your code only in the source files under the `// TODO`
    - You can & should add static functions (which are not externed in the API via the header files).
    - Always check parameters of the externed functions.
    - Write clean and well-documented code. Use meaningful variable names and add comments to explain the logic of your code.
    - Do not forget to return the correct error codes according to your function execution sequence.
3. **Suggested** Development Process:
   You should implement functions as you go and not all at once.
    I. Refer to `main.c`, start by implementing the main function, before invoking any operation.
    II. Proceed with the implementation of `parser`. Make sure it works. Unit test everything, every possible argument according to the error codes.
    III. Now you can complete the `Main` remaining functions.
    IV. Proceed by implementing `encrypt`. For this you need to implement some functions in `file_utils`. Implement only the functions you need. You also need an encryption algorithm for a buffer, so start with type none - a naive implementation (buf_out data = buf_in data), and to test your `encrypt` code.
    V. Continue developing the rest of the encryption algorithms (and their corresponding decryption).
    VI. Finally implement `censor`, and complete the remaining `file_utils` functions.

4. Test Thoroughly: Test your implementation with various input files, command-line arguments, and encryption types.
    - Check the parser works correctly and handles all of the possible inputs (and errors as listed in the `error_types` header).
    - Check for correct encryption and decryption results against the supplied examples.
    - Verify that the censorship functionality works as expected.
    - Test with different input file sizes.
    - Try to think of all of the corner cases.
5. Handle Errors Gracefully: Implement proper error handling in all functions.
    - Check for potential errors such as invalid arguments, file I/O errors, memory allocation failures, and return appropriate error codes. Do not forget to free allocated memory and exit gracefully.
    - The return values - the error code. Do-not use the `exit()` function.
    - You should check your code for memory leaks with any tool you desire. I suggest you use `valgrind` as follows:

```
valgrind --leak-check=full --show-leak-kinds=all \
         --track-origins=yes main.exe
```

Appendix: Sequence Diagrams Suggestions

Those are mere suggestions!

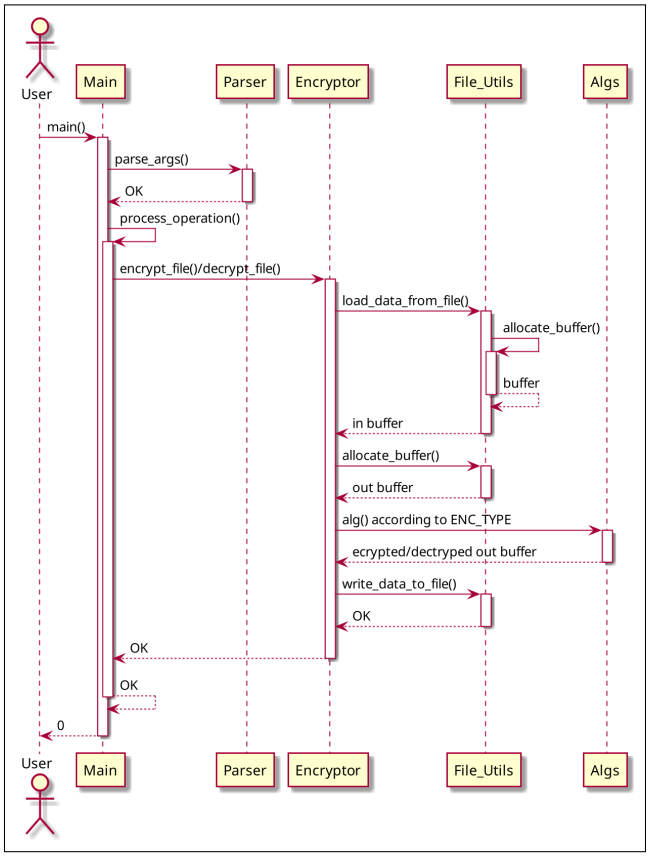

Figure 2: Encrypt/Decrypt Sequence Diagram

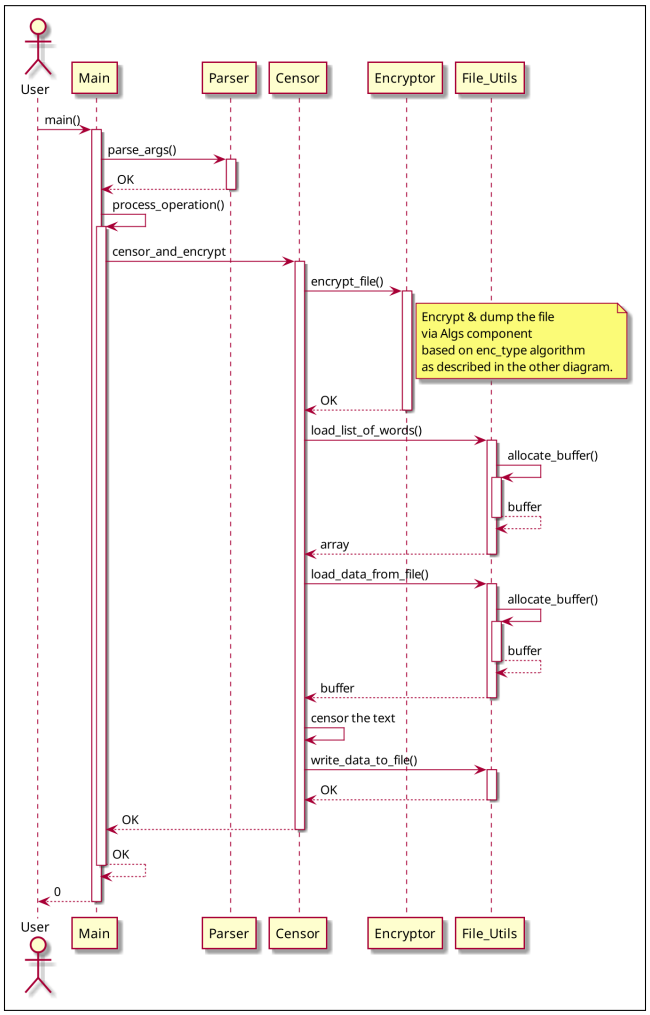

Figure 3: Censor Sequence Diagram