Topics for midterm test

1. Fundamentals
    1. What is algorithm?
    2. How to represent algorithms (pseudocode and flowchart)
    3. Ability to devise algorithms for simple problems and represent such algorithms using pseudocode and flowcharts
    4. Understanding of problem-solving recipe described in class lectures and practiced in labs/tutorials
    5. Understanding of programming environment including code editors, compiler toolchains, and shells
    6. Understanding of redirection operators: `<` and `>`
    7. Functional understanding of machines: CPU, control unit, ALU, registers, memory, input, and output
    8. von Neumann architecture for stored-programs: both instructions and data are stored in memory and instructions are executed by CPU to transform data (stored in memory)
    9. Low-level languages: machine languages, assembly languages
    10. High-level languages; translators from high-level to low-level languages including interpreters, compilers, and assemblers
    11. Basic data types:
        1. integer: signed and unsigned, `char`, `short`, `int`, `long`, `long long` data types
        2. floating-point: `float`, `double`, `long double`
        3. `void` keyword to represent *no data*
    12. Understand difference between variables and constants
    13. Lexical conventions and tokenization: identifiers, keywords, constants, operators, tokens, whitespace
    14. Able to tokenize simple code snippet, as in how many tokens in following statement: `printf("Average value: %f\n", average);`
2. Creating a simple program
    1. source file consists of functions
    2. functions are encapsulations of algorithms
    3. understand meaning of function declarations (prototypes) and definitions
    4. role of function `main`
    5. understand purpose of header files
    6. understand purpose of preprocessor program
    7. understand preprocessor directives: `include` and `define`
    8. what is a function argument and a function parameter
    9. single- and multi-line comments
    10. variables: declaration, definition, and initialization
    11. what does compiling mean?
    12. what does linking mean?
    13. understand options of `gcc`: `-std=c11`, `-pedantic-errors`, `-Wstrict-prototypes`, `-Wall`, `-Wextra`, `-Werror`, `-c`, `-o`

14. understand various stages of compiler driver (preprocessor, compiler proper, assembler, linker)
15. understand how to use `gcc` flags to invoke only the preprocessor, compiler, assembler
16. understand diagnostic warning and error messages from the compiler
17. I/O functions in standard library
18. mathematical functions in standard library
19. linking with external libraries

3. Arithmetic and assignment expressions
    1. operators, operands
    2. meaning of unary, binary, and ternary operators
    3. arithmetic operators
        1. multiplicative operators
        2. additive operators
        3. notion of type and behavior of `/` and `%` operators
    4. expression evaluation using expression trees
    5. precedence
    6. associativity
    7. assignment and side-effects
    8. *lvalues* and *rvalues*
        1. what is *lvalue* and what is *rvalue*?
        2. why does `x = 7` work and not `7 = x`
    9. compound assignment operators
    10. `sizeof` operator
    11. implicit type conversion
    12. explicit type conversion using type cast operator `()`
    13. order of operand evaluation and sequence points
4. Conditionals
    1. relational operators (`<`, `<=`, and so on) and equality operators: `==`, `!=`
        1. integer values for true and false expression evaluations
        2. applications and examples
    2. boolean values resulting from relational expressions
    3. `_Bool` type and `<stdbool.h>` header and `bool`, `true`, and `false` macros
    4. logical operators (`&&`, `||`, `!`)
        1. order of operand evaluation and short-circuit evaluation
        2. precedence and associativity
        3. applications and examples
    5. `if` statement
        1. what is a statement in C? (`;` or `expr;` or block of statements delimited by `{` and `}`)
        2. `else` clause
        3. nested `if` statements
        4. dangling `else` problem
        5. applications and examples
    6. Conditional operator `? :`

   7. `switch` statement
      1. what is a label?
      2. `case` keyword
      3. `break` keyword
   8. Why these operators are special (hint: they are sequence operators): logical or: `||`, logical and: `&&`, conditional operator: `? :`, and comma operator `,`
5. Iteration - looping and repetitions
   1. `while` statement
      1. different problem solving techniques involving counters, sentinels
      2. introduce `getchar` and `putchar` standard library functions
      3. problem solving techniques involving text files: emulating Unix programs such as `wc`, `cp`, `cat`, ...
   2. `for` statement and significance of its three expressions
   3. rewriting a `for` statement as a `while` statement and vice-versa
   4. `do while` statement
   5. guidelines on when to use which iteration statement
   6. infinite loops
   7. Jump statements: `break`, `continue`, and `return` statements
   8. increment and decrement operators
   9. unspecified expression evaluation when using increment and decrement operators in expressions
6. Larger Programs
   1. Programs consisting of multiple source files
   2. headers files, function declarations (or prototypes), function definitions
   3. idea of compiling individual source files into object files and linking these object files plus external libraries into single executable
7. Functions
   1. Function call operator `()`
   2. Call-by-value semantics
   3. What is stack? What is a stack frame? Understanding of how functions use stack and stack frames for inter-function communication
   4. Able to trace function calls and determine arguments of each function call and return value from function call
8. Input/output
   1. Formatted input/output
      1. abstraction of I/O using streams
      2. `stdin`, `stdout`, and `stderr`
      3. `printf` function: printing integers and floating-point numbers to `stdout`
      4. `scanf` function: reading integers and floating-numbers from `stdin`, significance of `&` operator in arguments
      5. conversion specifiers used in `printf` and `scanf` functions to print common integer and floating-point types
      6. controlling width and precision of output of integers and floating-point numbers
      7. escape sequences - able to understand the meaning of this and similar escape sequences: `printf("\"\\%%\t%%\t%%\\\"\n");`
   2. Character reading functions:

1. `getchar` and `putchar`
2. Meaning of end-of-file macro `EOF` declared in `<stdio.h>`
3. Why should the return value from `getchar` be `int` and not `char` type