Assignment 2: Implementing **fscanf**

Goal

- develop an implementation of `fscanf` that supports
  - strings: sequence of characters ends in whitespace (' ') or newline (`'\n'`)
  - chars
  - ints: sequence of digits
  - variable number of parameters, passed using a linked list or array
- how scanf works
  - the format string contains place-holders (that start with a '%') and regular characters
  - the characters of the format string are parsed
  - regular characters from the format string must match a character from the input
  - when it finds a '%' character it
    - reads the next character and determines what the placeholder is for: int, string, or char
    - it verifies the type of the next parameter from the list; if there is a type mismatch, it returns the number of matched parameters
    - it reads characters from the input for as long as they match the type of the parameter (see above for the definition of the types)
    - the characters read are converted to the respective data type and stored in the parameter
    - advances to the next parameter and increments the number of matched characters
    - the last character that was read (and that did not match the type of the parameter) is pushed back into the input buffer, for example
      - when reading a whitespace while parsing a string
      - when reading a non-digit while parsing an int
  - advances in the format string
- since this is a file the input ends
  - in a newline
  - or at EOF
- the function should take as parameters:
  - the file, must have been opened previously
  - the format string, that should have the same syntax as the `stdio` implementation of `scanf`
    - no fancy stuff, just %d, %s, and %c
  - a singly linked list where list nodes contain
    - a pointer to the data: the address should be valid, which means it must be allocated prior to calling `scanf`; `scanf` will not allocate memory itself
    - an indication of the data type
    - the `next` pointer
  - it is also possible instead of using a linked list to use an array of structures, that comes at a lower grade
- the function should return the number of matched parameters
- test the function in a complete program
  - you are given a file containing student records
  - each student record contains
    - name
    - initial (single char)

        - surname
        - year (int)
        - course name (e.g. "KCOMP")
        - group (single char, e.g. 'A')
        - average (int e.g. 75)
    - the fields of a student record are written in the file in order, using the format string:
        - `"%s (%c) %s %d %s %c %d"`
    - the path to the file should be passed as a command line parameter
    - open the file
    - your program must use the `scanf` implementation to read the student records to a student database
        - this can be implemented as a linked list for the maximum grade
        - using a static array of pointers to student records, for lower grade
        - using a static array of student records, for lowest grade
    - print all the student entries from the database.
    - close the file.

Notes

- you are not allowed to use the `stdio fscanf` in any part of your scanf implementation.
- when you are reading integers, don't forget to initialise your parameter's value to 0 before starting to form the number out of the digits
- when you are reading strings don't forget to finalise the string with the '\0' character at the end
- `getchar` is used to take characters out of the input buffer for processing; you can use `ungetc(ch, stdin)` to put the character `ch` back in the buffer so it can be processed later
- when the format string is all processed you should clear the input buffer by reading all remaining characters until you get to the newline
- check for the EOF when parsing the input; if EOF is found then scanf should exit, returning the matched elements up to that point.

Grading

General 10/100

- structure to hold scanf parameter 5
- scanf parameters 5
    - using linked list 5/5
    - using array 2/5

Scanf implementation 30/100

- function prototype 3
- parsing format string 25
    - don't skip characters in the input 2/25
    - don't skip characters in the format string 2/25
    - handle EOF correctly 2/25
    - parsing non-placeholder characters 2/25
    - parsing placeholders 9/25
        - parsing int 4/9
        - parsing string 4/9
        - parsing char 1/9
    - storing values read into parameters 4/25
    - advancing to the next parameter 2/25
    - error check the parameters that should hold the values read 2/25
- return the correct value 2

Main program 60/100

- student database 20
  - structure to hold student details 5/20
  - option 15/20
    - using linked list 15/15
      - memory allocation for student record 3/15
      - memory allocation for list node 3/15
      - initialising the list node 4/15
      - adding nodes to the list 5/15
    - using array of pointers 10/15
      - declare the array correctly 5/10
      - memory allocation for student record 5/10
    - using array of structs 5/15
- calling the scanf implementation 25
  - create the parameter list 10/25
    - using linked list 8/10
      - allocate memory for list nodes 4/8
      - add nodes to the list 4/8
    - using array 4/8
    - correct number of parameters 2/10
  - for each parameter set the right values for its type and the address to where the data should be stored 15/25
    - parameter type must be correct and correspond to the format string placeholder 2/15
    - address must be correctly allocated in 5/15
      - location 2/5
      - size 3/5
      - either statically or dynamically.
    - values read must eventually be copied into the correct database entry 8/15
- reading from the file 15
  - get the file path from the command line arguments 3
    - error checking 1/3
  - declare the file variable 1
  - open the file 3
    - error checking 1/3
  - close the file 1
  - reading entries 7
    - read fixed (predefined) number of entries 3/7
    - read unlimited number of entries, using error checking on scanf to determine when to stop reading 7/7

Learning outcomes

- structures
- linked lists
- pointers
- memory allocation
- switch statements
- enums
- string processing
- state machines