WORK PACKAGE 2: C PROGRAMMING

EXERCISE 1: ROBOT

Implement a control program for a robot. The program asks for the robot's starting position (x, y coordinates, ranging between 0-99) and **for a string of characters 'm' and 't'**, where **m** stands for move one step in current direction and **t** for turn of direction as below.

move: means that the robot takes one step in the current direction.
turn: means that the robot turns 90 degrees clockwise.

The start direction is always north for the robot.

The program performs instructions from strings provided by the user, one by one. When all instructions have been executed the robot stops and the program prints out the new position of the robot. The program then asks for new starting position and the string of characters. The process repeats until the program encounters a dedicated end character (you can define it).

Implement the functions move() and turn() as two void functions and use a pointer parameters as arguments so that the function can update the robot position which is a variable in the main function (calling function).

Use `enum` and a record of type `ROBOT` as below for the robot's position and direction.

```
enum DIRECTION {N,O,S,W};

typedef struct {
        int xpos;
        int ypos;
        enum DIRECTION dir;
} ROBOT;
```

EXERCISE 2: LINKED LIST

a) Write a function that creates a linked list with a NUMBER of records of type REGTYPE (see below). The value of the variable data is given a random number between 0 and 100.

Function declaration : `REGTYPE * random_list (void);`

Test the function from the main() program.

b) Extend the program with a function with the function declaration:

```
REGTYPE * add_first (REGTYPE * temp, int data);
```

That adds a new record to the first position of the list and assign the field *numbers* the value of *data*. The function must return a pointer to the new first entry in the list. Extend main() so that this function is tested.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//#### Constants #####
#define MAX 5

// ##### typedefs          ####
typedef struct q{
        int number;
        struct q *next;
        struct q *prev;
} REGTYPE;

// ##### Function declarations   #####

REGTYPE* random_list(void);
REGTYPE* add_first(REGTYPE* temp, int data);


//###### Main program #######
int main(int argc, char *argv[])
{
    int nr=0;

    REGTYPE *act_post , *head=NULL;

    srand(time(0));  //   Random seed
    head=random_list();
    act_post=head;
    while( act_post!=NULL){
           printf("\n Post nr %d : %d" , nr++, act_post->number);
           act_post=act_post->next;
    }



     // --- Free the allocated memory  ---

      while((act_post=head)!=NULL){
         head=act_post->next;
         free(act_post);
      }



  return 0;
}

// ====     End of main   ======================================

REGTYPE* random_list(void ){
    int nr, i=0;
    REGTYPE *top, *old, *item;

    return(top);
}

//============================================================

REGTYPE* add_first(REGTYPE* temp, int data){


}
```

EXERCISE 3: BINARY FILES

You should write a program to manage a database of persons. The database should be stored as a binary file (this is important – binary, not text file!). The structure of the program is easiest to understand by reading the description and the program skeleton below.

From the **main program** you should be able to choose between these options:

```
1 Create a new and delete the old file.
2 Add a new person to the file.
3 Search for a person in the file.
4 Print out all in the file.
5 Exit the program.
```

After entered the choice the program executes the task and returns to the menu for new choices.

1. Create a new and delete the old file.
   Program creates a new file with the specified filename (hardcoded, fixed) and writes a first dummy record to the file and then close it.

2. Add a new person to the file.
   Gives an opportunity to put in one new person to a temp record and then add this record in the end of the file.

3. Search for a person in the file.
   Gives you an opportunity to search for all persons with either a specified first name or family name (by choice). The program prints out all persons with that name.

4. Print out all in file.
   Prints out the whole list

5. Exit the program.
   Just exits the program.

```
#include <stdlib.h>
#include <stdio.h>

// -----typedefs -------
typedef struct {
        char firstname[20];
        char famname[20];
        char pers_number[13]; // yyyymmddnnnc
} PERSON;

// Function declaration (to be extend)
PERSON input_record(void);                  // Reads a person's record.
void write_new_file(PERSON *inrecord);      // Creates a file and
                                            // writes the first record
void printfile(void);                       // Prints out all persons in the file
void search_by_firstname(char *name);       //    Prints    out    the    person    if
                                            // in list
void append_file(PERSON *inrecord);         // appends a new person to the file

int main(void){
    PERSON ppost;

    return(0);
}
```

You should make the program fail-safe, in particular:

- Checking if the file exists
- Checking if the list is empty

EXERCISE 4: BITPACKING

Write two programs – one that pack bits into a byte and the other one which unpacks into a string instructions. You need to store 4 different values in a byte. The values are:

Name	Range	Bits	Info
engine_on	0..1	1	Is engine on or off . This is bit no 7 (MSB == Most Significant Bit, the one that carries the value 128 in decimal)
gear_pos	0..4	3	The position of the gear. Three bits means that we can have 2^3 == 7 positions
key_pos	0..2	2	Position of the key – 0 == stop, 1 == on, 2 == engine starter on
brake1	0..1	1	Position of the front brakes
brake2	0..1	1	Position of the rear brakes. Bit no 0 (LSB)

We should store them in a byte like this:

```
[engine_on]     [gear_pos]      [key_pos]       [brake1]        [brake2]
 1 bit          3 bits           2 bits         1 bit           1 bit
```

The first program, ***code.c***, takes 5 arguments (fewer or more than 5 arguments should be treated as an error).

The arguments should correspond to the values/variables above.

Example for a start of the program from command line:

```
code   1 2 2 1 1
```

The above should be treated as:

```
Name             Value
------------------------
engine_on        1       Bit no 7
gear_pos         2
key_pos          2
brake1           1
brake2           1       Bit no 0
```

The program should pack these values together in a byte (unsigned char) and print it out to the console in hexadecimal form. For this example, it should be 'AB' corresponding to bits '10101011. After printing out the code to the console, the program should exit. The program should be fail-safe, i.e. if it finds anything wrong (too many/few arguments, faulty input values) your program should print out an error message and exit.

The second program, ***decode.c***, takes 1 argument (one argument, hexadecimal number) and prints out the bit positions for the engine, gear, etc. In the example of 'AB' (again, make it fail-safe). The argument should correspond to the decoded byte, e.g. as printed by `code.c`. If your program finds anything wrong (too many/few arguments, faulty input values) your program should print out an error message and exit.

The program should unpack the bytes according to the specification above. Print out the result as below:

```
Name            Value
-----------------------------
engine_on
```

```
gear_pos
key_pos
brake1
brake2
```

For example, start program in the console window :

```
decode AB
```

and the result should be:

```
Name           Value
-----------------------------
engine_on      1
gear_pos       2
key_pos        2
brake1         1
brake2         1
```