Week 7

Exercise 1: Step analysis, file processing in multiple files

Write a program that reads and analyzes the step data given in a file. The program reads step data from a given file, parses the number of steps to an array, and calculates the sum of the steps. The program consists of three programs: the main program, a file containing functions for reading the files and printing, and a header file.

The main program file contains only the main program, which calls the subroutines and finally prints to number of steps. The "input/output" module files `io.c` and `io.h` contains the following functions:

- int readStepsList(int list[], int * size);
- void printList(int list[], int size);

Recall that in C, arrays are effectively passed by reference by default. The subroutine **readStepsList** reads the number of steps per line from the file `stepdata.txt` and saves them in an array `list`. The reference argument `size` stells the number of observations, that is, the size otf the array `list`. As a return value, the subroutine returns the sum of the steps.

The **printList** subroutine gets the array `list` and its `size`. It prints the elements in `list`.

You can assume that `list` is a array with the maximum size of 50;

Finally, write the main program `program.c`, which calls your functions yielding the example run below.

In CodeGrade, you can upload multiple files. In this exercise, you must upload **three files**: `program.c, io.c and io.h`. Be sure to use these names, otherwise CodeGrade does not compile your work correctly.

The file `stepdata.txt` can be found in Moodle. The file is also available in CodeGrade, and you should **not** upload it.

Add a newline to the end of each printed line.

Example run:

```
Steps in the list: 5004 12054 3000 24002 2020 11548 7824 15829
14882 6831

Sum of steps is 102994
```

Exercise 2: A menu-based program for managing a list in multiple files

This task is similar to the L6T4 task. This week your task is to implement linked list using **header node** as explained in the lecture. However, in this exercise you do not need to care about the last pointer. You only have `head` pointer to the header element of the list. Recall that the header element does not contain no data

Divide the program into 3 files, one of which is `program.c` for the main program and for the menu. The other file is `list.c`, which contains all the subroutines that handle the linked list. Your task is to write the following functions:

- node *createList() – creates an empty list with a header node and returns a pointer to that header node.
- void addItem(node *head, int index, int `x`) – adds a new node to the list at position i. Note that he element after header is in the position 1. The new node contains integer `x`. If i is bigger than the number of elements, then the new node is added to the end.
- void removeItem(node *head, int index) – The node in position i is removed. If i ≤ 0, the functions prints a warning and returns to the menu. If i is bigger than the number of elements, then the last element is removed. If the list is empty, the program displays: `"Can't remove item from empty list!!\n"`
- void printList(node *head) – print the elements of the list.

The third file is `list.h`, which defines the node as a structure and declares all the subroutines in `list.c`. For a structure definition that is to be used across more than one source file, you should put the definition in a header file.

You should create an empty list in the beginning of the main program. The beginning of the main should look like this:

```
node *head = NULL;
head = createList();
int key = menu();
```

Add a newline to the end of each printed line**.**

Example run:

```
1)Add item to the list
2)Remove item from the list
3)Print list
0)Exit
1
Please enter index of the item:
1
Please enter the Number:
6
1)Add item to the list
2)Remove item from the list
3)Print list
```

```
0)Exit
1
Please enter index of the item:
1
Please enter the Number:
7
1)Add item to the list
2)Remove item from the list
3)Print list
0)Exit
1
Please enter index of the item:
3
Please enter the Number:
5
1)Add item to the list
2)Remove item from the list
3)Print list
0)Exit
3
Items in the list are: 7 6 5
1)Add item to the list
2)Remove item from the list
3)Print list
0)Exit
2
Please enter index of the item:
2
1)Add item to the list
2)Remove item from the list
3)Print list
0)Exit
3
Items in the list are: 7 5
1)Add item to the list
2)Remove item from the list
3)Print list
0)Exit
1
Please enter index of the item:
2
Please enter the Number:
9
1)Add item to the list
2)Remove item from the list
3)Print list
0)Exit
3
Items in the list are: 7 9 5
1)Add item to the list
2)Remove item from the list
```

```
3)Print list
0)Exit
0
```

Exercise 3: Lottery, dynamic multi-file memory allocation – Extra task

Note that this exercise is an extra task. By solving this task you may earn an extra point.

In Finnish Lotto, there are 40 possible numbers from the range 1-40. Each week, 7 numbers and one "extra number" is drawn.

Create a lottery program that ask the size of the set and the amount of numbers to be drawn. As the number of numbers to be generated is selected during program execution, dynamic memory management is required.

The program consists of modules with a header file and a source code file. The first module includes memory management and the second the functions related to the drawing of numbers.

The first module (files memory.c and memory.h) contains the following function:

- int * allocateMemory (int);

It allocates a memory of given size and returns a pointer to the allocated area.

The second module (files io.c and io.h) contains the following subroutines:

- void generateNumbers (int list[], int size, int max);
- int checkNumber (int list[], int size, int num);
- void printNumbers (int list[], int size);

The **generateNumbers** function takes the array into which the generated numbers are stored. The value `max` tells the size of the "universe", that is, the possible numbers are between 1 to `max`. The integer size tells how many numbers are drawn.

The function `rand()` returns an integer value between 0 and RAND_MAX. You should use the formula

```
num = (rand() % max) + 1;
```

to "draw" the numbers. Note that `rand() % max` returns a value between 0 and max - 1, so by adding +1, we get a number from a desired range: 1 to max.

The **checkNumber** subroutine takes an array, the number to be checked and size of the array. The function checks that the number `num` is not already in the array `list` and returns 1 if the number is already in the list, and 0 otherwise. This is to make sure that we are not drawing the same number twice.

The **printNumbers** subroutine gets the array `list` and the size of `list`. It prints the array.

The main program `program.c` uses your functions. Ask the user for the *size* of the set and how many numbers are drawn. Then use the memory allocation module and allocate memory for the drawn numbers. Run the draw and print the drawn numbers on the screen.

The `srand()` function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand()`. Write the command srand(1) to the beginning of your main function so that the series of numbers generated in this way corresponds to the series of CodeGrade.

You need to upload all the files: `io.c, io.h, memory.c, memory.h, program.c` to CodeGrade

Example run:

```
Enter the size of the set:
40
How many numbers are drawn:
7
These numbers are drawn:
24 7 18 36 34 16 27
```