Prerequisites:

- Basic knowledge of using computer and text editor
- Knowledge how to create a program
- Knowledge how to create a function
- Knowledge of different types
- Knowledge about conditional statements
- Knowledge about loops
- Knowledge how to work with files
- Knowledge how to create a pointer

Aims:

In this laboratory student will learn how to:

- Dynamic memory allocation
- Free memory blocks

1 Theoretical background

1.1 Allocating memory

To allocate some block of memory function *malloc()* could be used. The only parameter in this function is size of block counted in bytes. Some example of a proper allocation of block of memory is given bellow:

```
double *SomeValue = (double*) malloc(sizeof(double));
```

However, it is even better to use such a memory allocation to arrays of data than one variable because additional memory is needed to handle such an operation. Allocation of array in memory could be done as follows:

```
double *ArrayValue = (double*) malloc(30 * sizeof(double));
```

Because dynamic memory allocation is done during program running, sometimes there could be not enough memory and allocation failed. In such a case your program has to be prepared to handle such a situation. To solve such an issue the following source code could be implemented:

```
double *ArrayValue = (double*) malloc(30 * sizeof(double));
if (ArrayValue)
{
  ...
  // Place here some source code which should be done if
  // allocation finished with success
  ...
}
```

Other way to allocate some block of memory is to use the `calloc()` function – a subprogram of the `malloc()` function. It is similar to the previous one, however, this time there is no multiplying. This function need two parameters: number of elements, and size of each element. Example source code is as follows:

```
double *ArrayValue = (double*) calloc(30, sizeof(double));
if (ArrayValue)
{
  ...
  // Place here some source code which should be done if
```

```
    // allocation finished with success
    ...
  }
```

The main difference is that `malloc()` do not erase allocated block of memory where `calloc()` does. Therefore, the `malloc()` function is quicker and if you initialize each part of such a block of memory it is advised to use it. In turn, if you do not initialize it, then is safer to use the `calloc()` function.

1.2 Freeing memory

For standard variables memory is freeing at the end of some block of source code denoted mostly be `}`. However, memory's block allocated earlier by some pointer is allocated also after `}` even when pointer was deleted as well. It caused to *memory leak*. To avoid it a special function `free()` must be used by programmer. As an argument you have to give a pointer's name (where this pointer is pointing to the memory block which you want to free). Simple example of such a source code is as follows:

```
int *Array;
...
Array = (int*)calloc(1234, sizeof(int));
...
free(Array);
```

Generally, **each allocation of the memory has to be paired with memory deallocation!!!**

If you do not delete pointer even the block of memory was cleared, it is wise to set such a pointer to 0. You can avoid then serious problems by mistake usage of pointers (pointing wrong part of memory used maybe by other program(s)). Some example source code could be as follows:

```
free(Array);
Array = 0;
```

2 Exercises

1. Create some pointer pointing some 50-elements array and do not initialize it! Use the `malloc()` function. Display in the terminal each element per row where the first column has element's number (it is a counter), the second column has memory address, and the third column has element's value.

2. Create some pointer pointing some 50-elements array and do not initialize it! Use the `calloc()` function. Display in the terminal each element per row where the first column has element's number (it is a counter), the second column has memory address, and the third column has element's value.

3. Create right now some pointer pointing some 50-elements array and initialize it by random float values. Display in the terminal each element per row where the first column has element's number (it is a counter), the second column has memory address, and the third column has element's value.

4. Sort all elements in array from task 3 from the smallest to the biggest one.

5. Write some infinity program with a *leaking memory* problem. Take a look carefully at memory usage and then start your program. What you can see?

6. Modify your program from task 5 to be more stable.