NPRG041 – 2023/24 Winter – Labs MS – **Small Assignment C01**

Subsets

The objective of this first homework assignment is to implement a simple application that finds and prints all subsets of a given input set of elements. To simplify our situation as much as possible, we will assume that the elements of this set are just letters of the English alphabet. We also introduce this input set as such using an ordinary local variable directly within the body of the `main` function, in the form of a traditional C-style array `const char items[] = { 'A', 'B', 'C', ' D'}`.

It is obvious that we will somehow need to work with the size of this input array. And since we want to create universally usable and well-structured code, we certainly do not want to hardwire it anywhere. If we introduced this size as a fixed number, we would then have to manually change such a number every time the input set would be changed. We will therefore use the following trick: `const size_t count = sizeof(items) / sizeof(items[0])`.

We will decompose the entire program into appropriately designed functions and completely avoid the use of global variables. In other words, we will pass all the necessary data when calling the functions via the parameters expected by their interfaces. The main function responsible for the whole process of finding and printing the subsets must be designed universally. We will thus be able to potentially call it repeatedly, even for any other or otherwise large input arrays. Within the code you will submit, only one input will, in fact, be expected, namely the one already mentioned above.

In order to work with the memory reasonably while ensuring that we are able to find all the possible subsets in the same order, we will generate them using a depth-first search. Specifically, for each element, in the order from left to right, we decide whether we want it in the current subset or not, respectively. We must not change the order of the elements, the presence of a given element takes precedence over its omission.

We will remember the flags for individual elements using an array of logical values. Since we cannot know its size in advance, we will have to create such an array using the dynamic allocation mechanism. Specifically, just use the following construct `bool* signature = new bool[count]`, which gives us a pointer to the first item of such an array. At the very end, we must not forget to deallocate this array, though, using `delete[] signature`. If we forgot to do this, we would irretrievably lose the given memory.

We will print the found subsets to the standard output in the following format: `{ A, C, D }`. The entire set will be wrapped by a pair of curly braces, we will separate the individual elements with a comma. We have to pay attention to spaces, we do not put any comma after the last element, and in the case of an empty set, we only write one space `{ }`. There will be exactly one subset on each line, and each line will be terminated with `std::endl`.

It is expected that you put your entire solution into just one `*.cpp` file. This file will be the only one you will submit. The solution must be correct, the predefined tests must finish successfully without any errors. It is also necessary to ensure that there will be no compilation warnings, let alone errors. The objective of this task is to verify the ability to work with user-defined functions, passing arguments by value or pointer, traditional arrays, and the standard output. It is necessary to focus not only on factual correctness, but also on the code quality, especially the appropriate decomposition into functions and their interfaces.

Finally, let us point out several particular requirements resulting from frequent errors or inappropriate design. In particular, do not use more advanced constructs we have not yet worked with in the class. In other words, we will completely suffice with the `<iostream>` library, the `<<` operator for writing to the standard output stream `std::cout`, and the traditional C-style array. Do not use other libraries, especially no containers, etc. Do not use classes or templates either, the goal is to solve our problem with adequate means only.

Do not use more than one dynamically allocated array for our signature. Generate each specific subset only once, it is not permissible to generate possible duplicates with their subsequent detection and removal. While printing elements of a particular subset, only a single pass is permitted.

Names of all functions and variables should be chosen as sufficiently explanatory. Write everything, including comments, exclusively in English. For each function parameter, consider whether it should be equipped with the `const` specifier. For various counts or sizes, always use the `size_t` type.

Also, make sure that each function handles only one well-bounded and defined task. That is, you have at least separated a function for finding the subsets from a function for their printing. At the same time, you

want to offer the end users a function with a friendly interface, so we need to separate such a public function from the internal recursive one. Among other things, simply because we do not want to force anyone to allocate and deallocate our solely internal signature array, let alone even be aware of its existence.