NPRG041 – 2024/25 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 introduce this input set as a global constant in the form of a traditional C-style array `constexpr char items[] = { 'A', 'B', 'C', ' D'}` placed in a header file `Input.h`. Moreover, we can also assume that this input array will be valid, i.e., without duplicate elements. Its size is not implicitly limited, though, and so can even be entirely empty.

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 (and consistently at all places). We will therefore use construction `constexpr size_t count = sizeof(items) / sizeof(items[0])` and place the definition of this constant again into the specified header file.

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 differently big input arrays.

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. Value `true` at a given position will mean that the corresponding element will belong to the subset, `false`, on the contrary, not. In other words, this array is therefore nothing else than kind of a mask, mathematically a characteristic (indicator) function of a given subset.

Since we cannot know the size of this array in advance (i.e., during the compilation), we will have to create it 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`.

If we had an input set, e.g., `{ '1', '2', '3' }`, the expected output of the entire program would be as follows:

```
{ 1, 2, 3 }
{ 1, 2 }
{ 1, 3 }
{ 1 }
{ 2, 3 }
{ 2 }
{ 3 }
{ }
```

It is expected that you put your entire solution into just one `*.cpp` file. This file will be the only one you will submit. In other words, you will not submit the `Input.h` file, since it already is a part of the individual prepared tests. The solution must be correct, the 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 the 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 particular subset only once, it is not permissible to generate potential duplicates with their subsequent detection and removal. While printing the elements of a particular already found subset, only a single pass is permitted (it is therefore not possible, for example, to first count the actual number of elements in a given subset, and only then proceed to its actual printing).

Names of all functions, parameters, 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. Although we already know that a traditional C-style array is actually technically understood only as a pointer to its first item, it is still better to use the notation `char items[]` instead of `char* items` in function interfaces, because it is then obvious at first glance that an array really is expected, not just a single isolated item. For various counts, sizes, positions, or indexes, 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.

Finalize and clean up the entire code before submitting the task, i.e., submission of a genuinely final version is expected, not just some working version. Hence, remove any debugging or other code fragments that are not intended to remain in it permanently. Pay attention to the visual style of code as well, i.e., indentation of individual lines, writing of curly braces around loop bodies or condition branches, means of constructing multi-word names, but also free lines or other separators between individual parts of the code. Particular style is not that important, the most important thing in this sense is the absolute consistency.

When it comes to debugging of submitted solutions in ReCodEx, the evaluation of automatic tests works by comparing the actual and expected program outputs, specifically by taking the expected output as a baseline against which the differences are found. All such detected differences can then be found in the log. Lines marked with `+` are extra (they should not be in the output), lines marked as `-` are missing, on the contrary. If the expected and actual lines differ only very little, we may also encounter a compact notation. E.g., `-10/+10: [3]B != [3]C` means that on line 10, in column 3, value B was expected, but C was found. Numbers of the matched lines may, of course, mutually differ, as may the starting positions or lengths of the differing fragments.

Let us also add that it is not necessary to use the classic technique of searching the space of all possible values using a recursive function when looking for our subsets. After all, calling functions is not exactly the fastest thing to do. We can therefore also use the binary counting method. In that case, we can do without recursion and only with local cycles. Even here, however, it makes sense to try to reduce the number of passes through the signature to a minimum, so it is advisable to combine its decrementation and termination condition testing (detection of finding the very last subset, i.e., the empty one) together.