NPRG041 – 2023/24 Winter – Labs MS – **Small Assignment C03**

Options II

Within this task, we will program an extension to the functionality of the previous task where we enabled basic detection of arguments passed to the application from the command line. This extension will allow us to start distinguishing between several allowed types of values associated with value options, store states and values of all the expected options in an auxiliary data structure, and print this structure to the standard output in the final overview. All the other aspects of the task remain unchanged.

So far, we have only printed the associated values of the value options we encountered, so we did not need to deal with their possibly different data types at all. In other words, everything was just a `std::string` string for us. We will now start distinguishing between string values (thus the current behavior), integer numbers (`int` with a number entered in the decimal system), and numbers with a floating decimal point (`float`).

Each value option will now have a predetermined type of value it assumes. Again, we are expected to hardwire everything in this sense directly into our code. Specifically, we expect the following behavior: options `-r`, `--red`, `-g`, `--green`, `-b`, and `--blue` require an integer, options `-a` and `--alpha` expect a decimal number, and options `-o` and `--output` assume a plain text value.

To parse the corresponding numerical values from an input string, we will use functions `int std::stoi (const std::string& str, std::size_t* pos)` and `std::stof`, respectively. They both have the same interface (at least as for the first two parameters we need), and the same principles of behavior. We need to be careful to correctly detect all the possible error situations should the parsing fail, though. This means that we have to catch both `std::invalid_argument` and `std::out_of_range` exceptions, as well as to correctly handle the situation when an otherwise legitimate numeric value would be followed by anything else, since that is not allowed in our situation, too.

During the process of handling the input arguments, we will preserve the printing of all the information messages we assumed in the previous task entirely unchanged. Beside them, we will also print the following messages:

- `Value <value> is not a valid integer number!`
- `Value <value> is not a valid floating point number!`

It holds that we will always print a standard message about finding a value associated with a given value option, only after it will the abovementioned error information potentially follow as another message. If no error occurs, no additional message will be printed.

We will store the information about the status of finding all the expected options, associated values in the case of the value ones, as well as the values of all the found standalone values into an instance of a structure we will design for this purpose. We are expected to use the `struct` structure, not the `class` structure. We will perceive all its data items as public, and therefore access them directly. In other words, we will not conceal access to them by any member functions (methods). As for the standalone values, we will store them all, preserving their relative order, within a nested vector of strings.

An instance of this structure will be created directly inside the `main` function, our main function `process_arguments` will then take a reference to it and will record all the detected information into the structure gradually during the argument handling process.

Let us add that some of the options are paired. This means that we do not care which particular variant appeared, just one of them is enough. In other words, they are just aliases to each other. As expected, we will specifically work with these linked pairs: `-t` and `--transparent`, `-r` and `--red`, `-g` and `--green`, `-b` and `--blue`, `-a` and `--alpha`, `-o` and `--output`.

At the same time, individual options may appear in the input arguments even repeatedly. Especially for the value ones, this means that the last request wins. That holds even if it would lead to an invalid value. If, for example, we first detect a valid option `-r 255` and then an invalid `-r string`, the resulting state will be such that the option `r` will act as undetected, and therefore without a value.

Let us add that this does not affect the gradually printed information messages, it is just a matter of what information will be stored in our structure. In other words, we remember a newly detected value

option and its value if and only if it was correctly parsed. Otherwise, we will ensure that any previously detected information about this option will be removed from the structure.

As for the decomposition of the problem into appropriately proposed functions, we continue to assume the use of the already existing function `process_value_option`, while we will also add new individual functions `process_string_option`, `process_int_option`, and `process_float_option`. Names of these functions may again be changed arbitrarily.

Before the end of the program execution, we will print the contents of the structure to the standard output. We will use the following message templates for that:

- `Flag option <name|...> is <disabled>`
- `Flag option <name|...> is <enabled>`
- `Value option <name|...> is <disabled>`
- `Value option <name|...> is <enabled> and associated with value <value>`
- `Standalone value <value>`

As usual, each message will be terminated by the end of the line. In angle brackets with a name, we give a single option name (e.g., `x`) or both names for the paired ones (e.g., `r|red`). The order of the options will be as follows: `x`, `y`, `grayscale`, `t|transparent`, `r|red`, `g|green`, `b|blue`, `a|alpha`, and `o|output`. When printing numerical values, we assume the use of the `std::to_string` construct. At the very end, we will also list all the standalone values, if there were any. Detected unexpected options will not appear in this output at all, nor did we store them in our structure in the first place.

The overall program output will contain 1) gradual information messages from the detection phase, 2) one empty line for separation, 3) the aforementioned control dump of our structure, 4) another empty line, and finally 5) information about the intended exit code. The expected output for input arguments `-tr255 --green 10 data.yaml -wa0.5y -oimage.png 20 --red 9988776655 -b` will be as follows:

```
Flag option <t> detected
Value option <r> detected with value <255>
Value option <green> detected with value <10>
Standalone value detected <data.yaml>
Unknown option <w> found!
Value option <a> detected with value <0.5y>
Value <0.5y> is not a valid floating point number!
Value option <o> detected with value <image.png>
Standalone value detected <20>
Value option <red> detected with value <9988776655>
Value <9988776655> is not a valid integer number!
Value option <b> detected but its value is missing!

Flag option <x> is <disabled>
Flag option <y> is <disabled>
Flag option <grayscale> is <disabled>
Flag option <t|transparent> is <enabled>
Value option <r|red> is <disabled>
Value option <g|green> is <enabled> and associated with value <10>
Value option <b|blue> is <disabled>
Value option <a|alpha> is <disabled>
Value option <o|output> is <enabled> and associated with value <image.png>
Standalone value <data.yaml>
Standalone value <20>

Intended exit code <1>
```

Submit all the created source files (`*.cpp` and `*.h`). Accompany more complicated fragments of your code with brief comments (single-line comments `// ...` will suffice). Because of the ReCodEx behavior,

we will again always return exit code `0`. The main goal of the task is to learn how to parse numerical values and work with structures.

Again, be careful not to repeat fragments of similar or even the same code. This also applies to the locations where we invoke our `process_*_option` functions, thus it is also about the smartness of their interfaces. If you choose to fully or partially expand a namespace using the `using` construct, never do so in a header file.

It still applies that we can only hardwire the option names in one place (or separately for short and long options, but not repeatedly). In other words, it is necessary to ensure that the very existence of a given option, distinction of its variant in terms of flag and value options, selection of the associated value type in the case of the value options, and also specification of a place where to store the detected information within our structure must all be provided for each individual option only in one single place. Option names themselves must again be introduced using global named constants.