NPRG041 – 2025/26 Winter – Labs MS – **Small Assignment C02**

Options

The goal of this task is to implement a simple application that would allow to process arguments passed to it from the command line at startup. It involves detection of both the expected and unexpected options, correct distinction of their types, detection of the associated and standalone values, and printing all such recognized options and values to the standard output.

We will distinguish two variants of options based on the possible lengths of their names, namely *short* and *long* options. The short ones have exactly one character in their name and are written with one dash at the beginning (e.g., `-x`). Long options are indicated by two dashes and their name contains at least one character (e.g., `--grayscale`). All names are case sensitive.

Regardless of the lengths of option names, we will also distinguish two additional types of options. First, options where we are only interested in their presence, and so we will call them *flag options*. Second, options that are to be followed by an associated value, always exactly one. We will therefore call them *value options*. In addition to these, we can also come across separate values (arguments not starting with `--` or `-`) that are not linked to any value option. We will call them *standalone values*.

Let us now focus on the allowed means of writing individual types of options. The short ones can be passed one after the other separately (e.g., `-x -y`), but it is also possible to have several of them at the same time after just a single opening dash (e.g., `-xy`). In such a case, however, we are sure that there will only be short options in a given group. In other words, long options cannot be grouped.

In the case of long value options, the associated value will always be passed separately, i.e., in the immediately subsequent argument (e.g., `--red 255`). Such a possibility also exists for the short value options (e.g., `-r 255`), but we can still work with the idea of grouping. In this case, however, there can only be at most one value option in the group, and only as the last one. All the following characters are then treated as its value (e.g., `-xyr255`). If there is no such character, then the value will be provided in the next argument (e.g., `-xyr 255`).

In general, the associated value can be anything as for its content. It can even look like an option (start with dashes). In the same way, we do not care if it is a number or something else, everything will be just ordinary strings for us. We also have to deal with the situation when the last argument implies the necessity of finding an associated value, but that value would be missing. The program must not crash in such a situation. As already implied by the previous text, if an argument does not start with dashes and it is not a value associated with the previous value option, it is a standalone value (e.g., `-x 255`). Let us also add that both the associated or standalone values can also be empty strings in edge cases.

The expected options and their types are fixed beforehand. Since we do not have a better reasonable means yet, we are expected to really hardwire them directly into our code. But only in one place (or separately for short and long options, but not repeatedly). Specifically, we expect flag options `-x`, `-y`, `--grayscale`, `-t`, and `--transparent`, and value options `-r`, `--red`, `-g`, `--green`, `-b`, `--blue`, `-a`, and `--alpha`. For the option names themselves, global named constants will be created.

The passed arguments need to be processed in just a single pass, without changing their order. Whenever we detect an option or value, at that moment we write a text message to the standard output that informs about the situation. Specifically, we expect the following types of messages:

- `Flag option <`*name*`> detected`
- `Value option <`*name*`> detected with value <`*value*`>`
- `Value option <`*name*`> detected but its value is missing!`
- `Unknown option <`*name*`> found!`
- `Standalone value detected <`*value*`>`

The format of these messages must be preserved, *name* is replaced by the actual option name without dashes, *value* is an associated or standalone value. The angle brackets `<>` are left unchanged, each line is terminated with `std::endl`.

Your source code needs to be divided into appropriately designed modules, e.g., `Main.cpp` file with solely the `main` function, and a pair of `Options.cpp` and `Options.h` files for all the executive code related to the

actual processing of arguments. The goal of the header file is to allow for the separation of declarations from the implementation. We automatically assume that all of our header files will be protected against undesired repeated inclusion using the mechanism of `#ifndef`, `#define`, and `#endif` directives.

In order we do not have to work with the passed arguments in the form of a traditional array with C-style strings, we will first convert them into a more user-friendly form of an `std::vector` container of `std::string` strings at the very beginning of the `main` function. This is easily achievable using a trick `std::vector<std::string> arguments(argv + 1, argv + argc)`.

We will decompose the intended code into appropriately designed functions. Basic inspiration can then be found in the following overview. Particular functions, their names, and interfaces are not prescribed, though. It is therefore possible to deviate at will.

- `process_arguments`: processes all arguments and so contains a main loop over individual arguments and subsequent branching for long and short options and standalone values
- `process_short_option`: processes a short option, i.e., performs branching to flag, value, and unknown options based on the previously discussed predefined list of the expected options
- `process_long_option`: the same for long options
- `process_flag_option`: processes a short or long flag option
- `process_value_option`: the same for value options
- `process_unknown_option`: processes an unknown option
- `process_standalone_value`: processes a standalone value

As for the main loop over the individual arguments, it is expected that iterators will be used via construction `for (auto it = arguments.begin(); it != arguments.end(); ++it) { ... }`. Perhaps the biggest complication will be the processing of a value option when its value is provided in the next argument. In such a case, we can basically choose from two possible solutions: either we process such an option backwards only at the moment of finding its value, or on the contrary, we process it at the moment of finding it by forward fetching of its value. In the former case, we will have to transfer the necessary information between the individual iterations, while in the latter one, we will have to manually intervene in the course of the loop. This is easy with the iterators, though.

For the purpose of branching the code while detecting the individual expected and unexpected short options, we will use the `switch` construct (including the possibility of providing multiple `case` labels sharing the same body). Of course, that will not work for the long options, and so we will use ordinary `if`, `else if` and `else` conditions there.

The essential thing is that the `process_flag_option` and `process_value_option` functions really need to be proposed so that we can easily use them for both short and long options at the same time, i.e., with just a single interface. Moreover, we need our function for value options to be capable of finding the associated value in the same or the following argument by itself, without leaving such a task to the caller.

The reason is that we want to achieve as elegant code on the side of the caller as possible, e.g., in the style of a single line `if (name == "red") { process_value_option(name, …); }`. In order to pursue this intention, it suffices to apply a trick where the last few function parameters can have default values specified, thanks to which we do not have to determine values of such parameters at all when calling that function. Let us also note that each function can, of course, only have one expected return value. If that is insufficient to pass all the necessary information, we can use the so-called output parameters, i.e., ordinary parameters passed by modifiable references.

All our functions must be implemented in a way that we can inform about the success of their execution. All that by returning values of the `bool` data type. If everything goes well, `true` is returned. Otherwise, `false` is returned. Such a situation arises when we detect at least one unexpected option or we detect an expected value option, but without its value. Detection of an error situation does not cause our program to be interrupted, the processing of the remaining arguments further continues without any change.

We must be able to forward the error information up to the `main` function, where we use it to infer the exit code of the entire program. In particular, `0` in the case of success, and, `1` on the contrary.

In order to avoid any doubts regarding the correct understanding of our problem and all the partial requirements, let us have a look at the following sample program execution: input arguments `-xwrgb --red --c 255 --grayscale -tb` will return exit code `1` and the following output:

```
Flag option <x> detected
Unknown option <w> found!
Value option <r> detected with value <gb>
Value option <red> detected with value <--c>
Standalone value detected <255>
Flag option <grayscale> detected
Flag option <t> detected
Value option <b> detected but its value is missing!
```

For the purpose of debugging, let us add that we can easily set the arguments to be passed when the application is launched within the MS Visual Studio development environment. You just need to alter the project settings, in particular, via *Project* → *Properties* → *Debugging* → *Command Arguments*.

Submit all source files you created (`*.cpp` and `*.h`). Names of all the aforementioned files or functions are not directly required, you can change them as you like. The main objective is to learn how to work with strings, header files, passing parameters by references, and also to get a basic idea of working with the vector container and its iterators. Do not forget the aspect of quality design, general requirements such as correctness, or avoidance of inadequate constructs or libraries that we have not yet learned to work with.

When it comes to common mistakes and, on the contrary, good practice, do not forget the following aspects. The `main` function must not contain too much code, let alone low-level code. Therefore, just at first glance, it should give you a good high-level idea of what the program is supposed to do.

Always use protection against repeated inclusions in your header files because you never know who and how will use them in the future. However, do not use `#pragma once`, it is not a standard directive. Only include standard libraries and user header files in the scope where they are really necessary. In other words, do not include anything that is not needed. In general, it is not appropriate to automatically put declarations of all our functions into header files. That is only meaningful for those we want to offer to the others for use. This means we will not put there our purely internal functions. Their declarations can (but do not need to) be placed at the beginning of source files.

Choose appropriate names for all files, variables, functions, and their parameters. Do not forget to use the `const` specifier when passing parameters whenever appropriate. If not intended, avoid passing any objects more complex than instances of primitive types (like `int` etc.) by value. This also applies to `std::string` strings. In other words, use references or pointers appropriately. If you do not need the opposite, prefer preincrement over postincrement operations. Avoid repeating similar or even the same code fragments. Avoid excessive nesting of conditional expressions, too.

Technical solution should never conceal the logical nature of the problem. Therefore, e.g., we have to perform the conversion of our `bool` error return values to `int` exit codes explicitly. In other words, we cannot rely on the fact that booleans are actually implemented as numeric values, moreover, exactly those we want. This is just a coincidence, it could be entirely different. Values of the exit codes as such are expected to be defined via named constants. Finally, when gathering partial error values across our functions, do not forget that the mechanism of lazy (short-circuit) evaluation of logical expressions is applied.

Finally, if you decide to fully or just partially expand the `std` namespace using the `using` construct, never do so in a header file. Such a step would be impossible to subsequently undo.