NPRG041 – 2023/24 Winter – Labs MS – **Small Assignment C02**

Options I

The goal of this task is to implement a simple application that would allow to process all arguments passed to it from the command line at startup. It involves the 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`).

Regardless of the lengths of option names, we will also distinguish another two types of options. First, options where we are only interested in their presence, 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*.

Attention also needs to be paid to the allowed means of specifying individual option types. 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). 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`).

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.

Your code also needs to be decomposed into appropriately designed functions. This at least means to separate a function for the processing of the arguments. We will design it so that it expects the arguments in the form of a `std::vector` container of `std::string` strings, not pointers to the traditional C-style strings, as we obtained them within the `main` function. We thus need to convert them first, simply using the `std::vector<std::string> arguments(argv + 1, argv + argc)` trick.

In order to learn to work with other intended constructs, too, it is necessary to use the mechanism of iterators offered by the vector container to iterate through the individual arguments. Just use `for (auto it = arguments.begin(); it != arguments.end(); ++it) { ... }`. For the purpose of branching the code while detecting the individual expected and unexpected short options, it is necessary to use the `switch` construct. Of course, that will not work for the long ones, and so we will use ordinary conditions there.

With respect to the planned extension of our application within the following assignment Options II, it is expected to introduce separate functions for processing individual variants of detected options and values, irrespective of the fact that we will currently only use them to print the already discussed information messages. In particular, let us assume functions `process_flag_option`, `process_value_option`, `process_unknown_option`, and `process_standalone_value`.

The first two mentioned functions need to be designed in a way that we can easily use them for both short and long option names with just a single interface. Especially as for the value options, this means that our function itself must be able to find the associated value in the same or the following argument, depending on the situation, without leaving such a task to the caller. Therefore, the goal is to achieve as elegant code as it is possible, e.g., something like `… if (name == "red") { process_value_option(name, …); } …`.

In order to achieve 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 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 so that we will be able to inform about their successful execution or, on the contrary, problems encountered. All that by returning values of `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.

We must be able to forward this error information up to the `main` function. There we would normally use it to infer the exit code of the entire program, i.e., `0` in the case of success, and, for example, `1` vice versa. Unfortunately, ReCodEx cannot work with such codes and considers anything other than `0` as a test failure. For that reason, we will always return exit code `0` regardless of the situation.

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 the 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 necessary to 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.

Choose appropriate names for all files, variables, functions, and their parameters. Do not forget to use the `const` flag 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 preincrementation over postincrementation. Avoid repeating similar or even the same code fragments. Avoid excessive nesting of conditional expressions, too.