Parsing command line

Here we discuss how to parse command line to test for the existence of parameters. Read below to find out more!

Command line might contain text arguments that could be interpreted in a few ways:

- as an integer
- as a floating-point
- as a boolean flag
- as a string (not parsed)
- or some other types...

Simple int and string version

We can build a variant that will hold all the possible options.

Here's a simple version with int and string:

```
class CmdLine
{
public:
    using Arg = std::variant<int, std::string>;
private:
    std::map<std::string, Arg> mParsedArgs;
public:
    explicit CmdLine(int argc, char** argv) {
        ParseArgs(argc, argv);
    }
    std::optional<Arg> Find(const std::string& name) const;
```

```
    // ...
};
```

Now let's look at the parsing code!

Parsing Code

```cpp
#include <iostream>
#include <variant>
#include <map>
#include <charconv>
#include <cstring>
#include <optional>
#include <string>
#include <string_view>

class CmdLine
{
public:
    using Arg = std::variant<int, std::string>;

private:
    std::map<std::string, Arg> mParsedArgs;

public:
    explicit CmdLine(int argc, const char** argv) {
        ParseArgs(argc, argv);
    }

    std::optional<Arg> Find(const std::string& name) const;

    void ParseArgs(int argc, const char** argv);
};

std::optional<CmdLine::Arg> CmdLine::Find(const std::string& name) const
{
    if (const auto it = mParsedArgs.find(name); it != mParsedArgs.end())
        return it->second;

    return { };
}

CmdLine::Arg TryParseString(std::string_view sv)
{
    // try with int first
    int iResult = 0;
    const auto last = sv.data() + sv.size();
    const auto res = std::from_chars(sv.data(), last, iResult);
    if (res.ec != std::errc{} || res.ptr != last)
    {
        // if not possible, then just assume it's a string
        return std::string(sv.data(), sv.size());
    }

    return iResult;
}

void CmdLine::ParseArgs(int argc, const char** argv)
{
```

```cpp
        // the form: -argName value -argName value
        // unnamed? later...
        for (int i = 1; i < argc; i += 2)

        {
            if (argv[i][0] != '-')
                throw std::runtime_error("wrong command name");

            mParsedArgs[argv[i] + 1] = TryParseString(argv[i + 1]);
        }
    }

    int main(int argc, const char** argv)
    {
        if (argc == 1)
        {
            std::cout << "run with -paramInt 10 -paramText \"Hello World\"\n";
            return 0;
        }

        try
        {
            CmdLine cmdLine(argc, argv);

            if (auto arg = cmdLine.Find("paramInt"); arg)
                std::cout << "paramInt is " << std::get<int>(*arg) << '\n';

            if (auto arg = cmdLine.Find("paramText"); arg)
                std::cout << "paramText is " << std::get<std::string>(*arg) << '\n';
        }
        catch (std::bad_variant_access& err)
        {
            std::cerr << " ...err: accessing a wrong variant type, " << err.what() << '\n';
        }
        catch (std::runtime_error &err)
        {
            std::cerr << " ...err: " << err.what() << '\n';
        }

        return 0;
    }
```

Explanation

The idea of `TryParseString` is to try parsing the input string into the best matching type. So if it looks like an integer, then we try to fetch an integer. Otherwise, we'll return an unparsed string. Of course, we can extend this approach.

After the parsing is complete the client can use `Find()` method to test for the existence of a parameter:

```cpp
std::optional<CmdLine::Arg> CmdLine::Find(const std::string& name) const {
    if (const auto it = mParsedArgs.find(name); it != mParsedArgs.end())

        return it->second;

    return { };
}
```

`Find()` uses `std::optional` to return the value. If the argument cannot be found in the map, then the client will get empty optional.

The above example (*line 72-89*) uses `cmdLine.Find()` to check if there's a given parameter. It returns `optional` so we have to check if it's not empty. When we're sure the parameter is available, we can check its type (pass it to `holds_alternative` to be sure it contains the proper type).

Variant with `int`, `float` and `string`

At the moment of writing, `std::from_chars` in GCC/Clang only supports integers.

MSVC starting from the version 2017 15.8 has full support also for floating-point numbers. You can read more about `from_chars` in the separate String Conversions Chapter.

If you want to have a look at what the code with floating-point numbers will look like, you can download the following file:

⊘ variant_parsing_int_float.cpp ⤓ ↗

The next section deals with the use of visitors in state machines. To learn more details keep on reading.