

Hardware and System Specification Command-Line-Interface Tool

Evan Elias Young
College of Engineering and Computing
Missouri University of Science and Technology
Rolla, Missouri 65401–3066
Email: eeymrr@umsystem.edu

Abstract—There are many tools available to retrieve system specifications, however many of these lack cross-platform support and machine-readable output. Many different operating systems include a command-line-interface to retrieve some part of the information, however this is often not human-readable. A single tool which works on every operating system, includes human-readable output, and implements native libraries to quickly retrieve accurate information is necessary. Working together many different system libraries, formats, data types, and parsing revealed which operating systems include the most legacy code, which operating systems are the most streamlined, and the way in which libraries come together to form a cohesive application.

Index Terms—Namespace, WMI (Windows Management Instrumentation), JSON (JavaScript Object Notation)

I. IMPLEMENTATION

Both the makefile and the primary header determine which operating system the host is running, either to link different libraries or to define override functions. Windows requires several libraries to be built into the distributable, and several libraries to be linked to provide access to WMI and the registry.

- A. Windows
- B. macOS
- C. Linux

II. SEMANTICS

Semantics describe the methodology behind design choices which do not effect the functionality of the project.

A. Organization

Within each namespace, there are different queries a user can make; Separating these queries into folders identified by the namespace was a logical conclusion. Each namespace folder contains a single header for function headers, namespace types, header inclusions, etc., a source file for collecting queries, and a source file for JSON conversion.

III. LIBRARIES

A. JSON for Modern C++

JSON parsing was desired as it is easily machine readable and very portable, therefore some implementation of JSON was required. *JSON for Modern C++* was chosen as a single-header option, written for C++17 it includes very modern

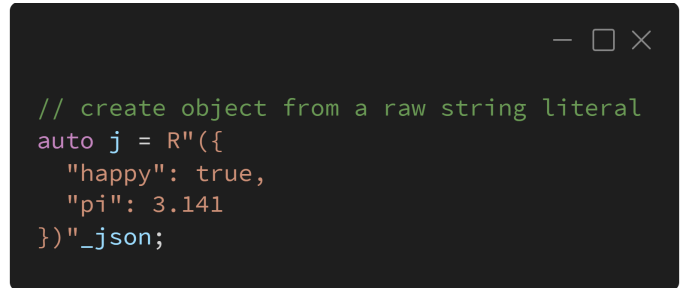


Fig. 1. Declaration of simple JSON object through the use of raw string literal suffixes.

features and encourages the use of these quite well. In addition to modern features, the structuring and destructing is simple to implement and very efficient.

B. *argparse*

For a command-line interface, argument parsing was necessary to differentiate queries, listings, and selection. *argparse* was chosen as a single-header option, written for C++17 it includes very modern features and encourages the use of these quite well. In addition to modern features, the implementation and usage is very similar to python and is quite efficient too.

IV. RESULTS

V. CONCLUSION

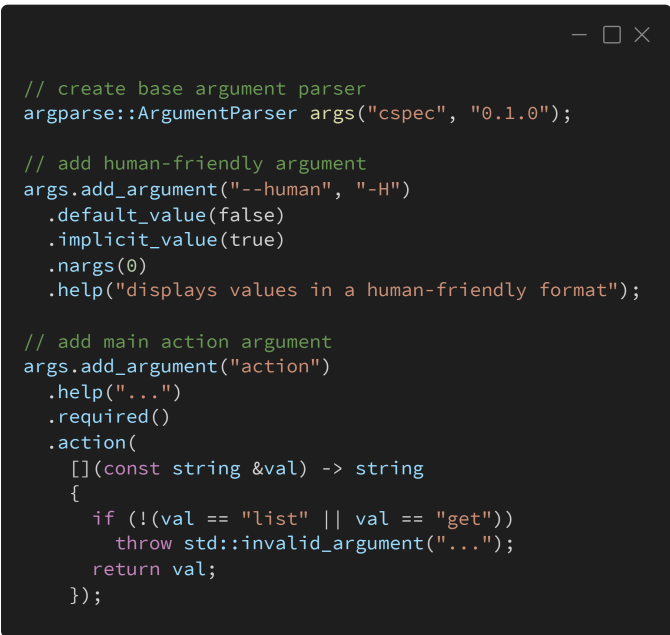
The conclusion goes here.

ACKNOWLEDGMENT

I would like to thank Niels Lohmann et. al. for their contributions to *JSON for Modern C++*, and Pranav et. al. for their contributions to *argparse*.

REFERENCES

- [1] S. Whims, V. Kents, et. al., *Example: Getting WMI Data from the Local Computer*. Seattle, Washington: Microsoft, 2021.
- [2] G. Dicanio, *Use Modern C++ to Access the Windows Registry*, vol. 32 no. 5. Seattle, Washington: Microsoft, 2017.



```
// create base argument parser
argparse::ArgumentParser args("cspec", "0.1.0");

// add human-friendly argument
args.add_argument("--human", "-H")
    .default_value(false)
    .implicit_value(true)
    .nargs(0)
    .help("displays values in a human-friendly format");

// add main action argument
args.add_argument("action")
    .help("...")
    .required()
    .action(
        [](const string &val) -> string
        {
            if (!(val == "list" || val == "get"))
                throw std::invalid_argument("...");
            return val;
        });
```

Fig. 2. Declaration of argument parser object with one optional flag and one positional action.