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. Background

CSpec is a hardware and system specification command-line interface tool, where a user can query the system for different information regarding hardware and software. A user will either 'list' the available queries or namespaces, or 'get' a data member or entire namespace of data members. The output will be delivered in one of several ways: list, compact, value (only), or json, all supporting human-readable numerical values.

With the operating system determined at compile time, the compiler will know which header, source, and template files to include. This is powerful as it reduces the amount of compiled and included code to only what is necessary. This also ensures that the code will not fail to build, since each operating system is required to implement the same methods, where shims can be implemented on a per-system basis.

II. 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

Windows includes a very archaic method of dealing with strings, oftentimes requiring a separate string macro (BSTR) which will use wide characters or whichever legacy character your build of Windows supports. As such, conversion from standard characters to wide characters was included, and static casting was a necessity.

WMI was used for a vast majority of the system information, and [1] was used to wrap the functionality into a class (ensuring safe destruction). WMI requires a significant amount of setup, and at any point in the process could fail, which is why wrapping this in a class provided an intuitive endpoint of failure checking and resource freeing on failure.

The registry was another point of data access, and [2] was used as a guide to set up access and conversion methods.

B. macOS

macOS relied on the use of two system specific libraries; sysctl and system profiler. [6] was used to configure and research the sysctl utilization within C++ and macOS. While fairly intuitive, retrieving strings with sysctl required a temporary buffer and trimming on null characters.

System profiler unfortunately has no public-facing C++ headers, and therefore cannot be invoked directly in C++. As a workaround, running system profiler in bash and capturing the output into a buffer was used. In addition to system profiler's innate slowness (I am ignorant as to why), running a shell command and streaming into a buffer increases the runtime.

C. Linux

Linux was fairly simple to impelement, everything except memory information was able to be retrieved directly from C++. Linux memory information requires a shell command elevated to super-user to retrieve, which is a limitation due to the direct access of the memory on the system bus in order to determine the physical bank location of the memory.

Linux offers a variety of files at the root level which contain a breadth of information regarding os info, kernel info, gpu info, cpu info, and filesystem info. These files can be read by any user, while only offering write permissions to root; this, however, is sufficient to determine the hardware and software in the system.

III. Semantics

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

```
// create object from a raw string literal
auto j = R"({
  "happy": true,
  "pi": 3.141
})"_json;
```

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

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.

A shared namespace is used for general project utilities, such that the amount of header-source files are reduced and incremental patching of source files. Of importance is that this also allow for operating system specific headers and functionality to be included at compile time, with further scoped namespaces.

B. Templates

In standard C++, convention is to include templated methods and classes in either a header file (.h) or header-source file (.hpp). As these two file extensions are based on the notion of being a header, it can obfuscate whether or not the file is compiled, or built into a source file. Header files are neither compiled nor built-in, whereas header-source files are not compiled but built-in to a source file. This removes the ability for make and cmake to determine whether or not the source file has been updated, which is where a template file (.tpp) applies.

Template files will be included at the end of a header (.h) file, and will therefore always mark a source file as updated. Since the source file includes the header file prior to compilation, it will notice a change and rebuild accordingly.

IV. 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 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.

```
// 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.

TABLE I
PROJECT BUILD INFORMATION

OS	Threads	Build Time [s]	Build Size [MB]
Windows 11	24	28.98	1.7351
macOS 12.1 $\beta 1$	10	24.10	1.2134
Ubuntu 21.10	6	76.24	0.8210

B. Argument Parser for Modern C++

For a command-line interface, argument parsing was necessary to differentiate queries, listings, and selection. *Argument Parser for Modern C++* 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.

V. Conclusion

A. Product

Most queries are supported on every operating system, Table II details which queries are supported (✔), not supported (✘), partially supported (+), and requires sudo (−) on each

TABLE II
OS SUPPORT INFORMATION

<table>
<tr><th>Query</th><th>Windows</th><th>macOS</th><th>Linux</th></tr>
<tr><td>cpu</td><td colspan="3">✔</td></tr>
<tr><td>filesystem</td><td colspan="3">✔</td></tr>
<tr><td>gpu</td><td colspan="3">✔</td></tr>
<tr><td>memory</td><td colspan="2">+</td><td>+ −</td></tr>
<tr><td>memory.manufacturer</td><td>✘</td><td>✔</td><td>−</td></tr>
<tr><td>memory.voltage</td><td>✔</td><td colspan="2">✘</td></tr>
<tr><td>system</td><td colspan="3">✔</td></tr>
</table>

TABLE III
OUTPUT STYLE EXAMPLES

Format	List	Compact	Value	JSON
Default	... 2000	...=2000	2000	{"...": 2000}
Human	... 2 KB	...=2 KB	2 KB	{"...": "2 KB"}

operating system. System constraints restriced my ability to fully implement the Windows, macOS, and Linux namespaces.

All different styles of formatting output were implemented successfully, Table III provides views of different types of output, all accessible by the user when making queries or listings.

B. Knowledge

During the course of the project, I learned how to use git submodules, how to use external user-level libraries (see [4] and [3]), and how to interface with lower level system libraries (see [1] and [2]).

Segmentation of large amounts of operating system-dependent source code was an additional topic with which I was unfamiliar; this required different syntax in my makefile which supported recursing through directories of source code.

C. Moving Forward

In the future, I would like to expand on the CLI help and provide examples of queries and listings. Overall, I am satisfied with what I was able to accomplish regarding a cross-platform CLI tool which implements system-level libraries and functionality.

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*, Microsoft, Jan. 7, 2021. Accessed: Nov. 30, 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/wmisdk/example--getting-wmi-data-from-the-local-computer.

[2] G. Dicanio, "Use Modern C++ to Access the Windows Registry" *MSDN*, vol. 32, no. 5, May. 2017. Accessed: Nov. 30, 2021. [Online]. Available: https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/may/c-use-modern-c-to-access-the-windows-registry.

[3] P. Ranav, *Argument Parser for Modern C++ README*. (2021). Accessed: Nov. 30, 2021. [Online]. Available: https://github.com/p-ranav/argparse/blob/master/README.md.

[4] N. Lohmann, *JSON for Modern C++ README*. (2021). Accessed: Nov. 30, 2021. [Online]. Available: https://github.com/nlohmann/json/blob/master/README.md.

[5] *IEEE Reference Guide*, IEEE, Piscataway, New Jersey, November 12, 2018. [Online]. Available: https://ieeeauthorcenter.ieee.org/wp-content/uploads/IEEE-Reference-Guide.pdf.

[6] *Mac OS X SYSCTL Manual*, Apple, Cupertino, California, January 23, 2001. [Online]. Available: https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/sysctl.3.html.