The Journal of Open Source Software

CMakePPLang: An object-oriented extension to CMake

Zachery Crandall[1,2*], **Blake Mulnix**[1,2*], **Branden Butler**[1,2], **Theresa L. Windus**[1,2], **and Ryan M. Richard**[1,2¶]

1 Department of Chemistry, Iowa State University, USA **2** Chemical and Biological Sciences, Ames National Laboratory, USA ¶ Corresponding author * These authors contributed equally.

DOI: 10.21105/joss.05711

Software

- Review
- Repository
- Archive

Editor: Patrick Diehl

Reviewers:

- @bast
- @tianyi93

Submitted: 06 July 2023
Published: 12 September 2023

Summary

CMakePPLang is an object-oriented extension to the CMake language written entirely using the original CMake language with the goal of making projects built on CMake easier to create and maintain. That said, CMakePPLang has different coding practices, paradigms, and standards than the original CMake language, much in the same way that C++ coding differs from C coding despite some level of interoperability. Currently, CMakePPLang is used within the CMakePP organization ("CMakePP Organization," 2023) as the foundation for two in-progress projects: CMakeTest (*CMakeTest*, 2023) and CMaize (*CMaize*, 2022). CMakeTest provides a solution for unit testing CMake and CMakePPLang code. CMaize is a CMake tool to simplify interoperability between projects and writing their build systems.

Statement of Need

CMake (*CMake*, 2023) is an extensible build tool that exceeds at generating build systems for many combinations of platforms, compilers, and build configurations. CMake has become the *de facto* standard tool for building C, C++, and Fortran programs of moderate to large size. However, as the size of a project increases, the complexity of the CMake build code tends to increase as well, and the need arises to make building projects with CMake easier and less error prone. The complexity of builds will also increase as scientific computing moves toward heterogeneous systems, requiring programs to leverage a combination of CPUs, GPUs, and other specialized hardware (Richard et al., 2023). Better utilities and extensions in CMake can help alleviate these issues, but these tools must be able to be designed in a maintainable and testable way ("Software Quality," 2005). Object-oriented programming excels at managing and maintaining large, complex code bases (Ambler, 1998; Wirth, 2006), and there is an increasing need for this in the CMake language.

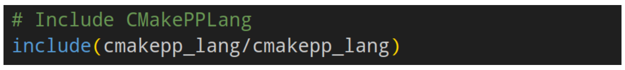
Tobias Becker (Becker et al., 2021) recognized these issues and wrote a purely object-oriented language on top of CMake, called CMake++ (formerly oo-cmake). CMake++ contains an abundance of CMake extensions. Many of those extensions have direct overlap with extensions that are part of CMakePPLang. Features include (among many): maps, objects, tasks/promises. Unfortunately development of CMake++ has largely been done by a single developer and it appears to have been abandoned, as there have only been two commits since July 2017, both in 2021.

One of the primary issues with CMake++ is the lack of documentation. While there is some high-level documentation, there is little to no API or detailed developer documentation. This makes it very challenging for a new developer to figure out what is going on. Initially, forking and expanding on CMake++ was considered, but it was determined that it would take similar time to decipher CMake++ as it would to develop CMakePPLang.

CMakePPLang has been developed to provide extensions to the CMake language which provide objected-oriented functionality and other quality-of-life improvements. The main features of CMakePPLang are the object-oriented functionality, strong data typing, addition of a map structure, and backwards-compatibility with CMake. These features allow for easier general programming in CMake, which is key to writing complex build tools in the language. Although CMakePPLang is built on top of CMake, CMakePPLang mostly relies on fairly fundamental features of the CMake language, so it is versioned independently of CMake using semantic versioning (*Semantic Versioning 2.0.0*, 2023).

Basic Usage

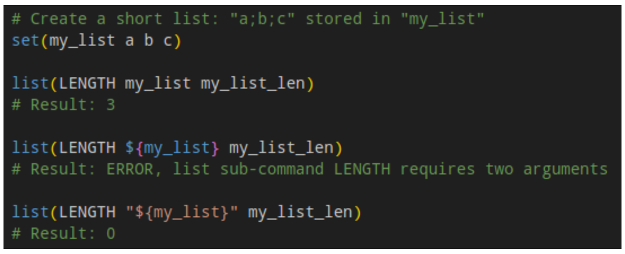
CMakePPLang is developed using CMake, so it is inherently backwards-compatible with CMake code and can be combined with CMake in the same `CMakeLists.txt` or `*.cmake` files. To use CMakePPLang, it is simply included like any other CMake module after it is downloaded (Figure 1).



```
# Include CMakePPLang
include(cmakepp_lang/cmakepp_lang)
```

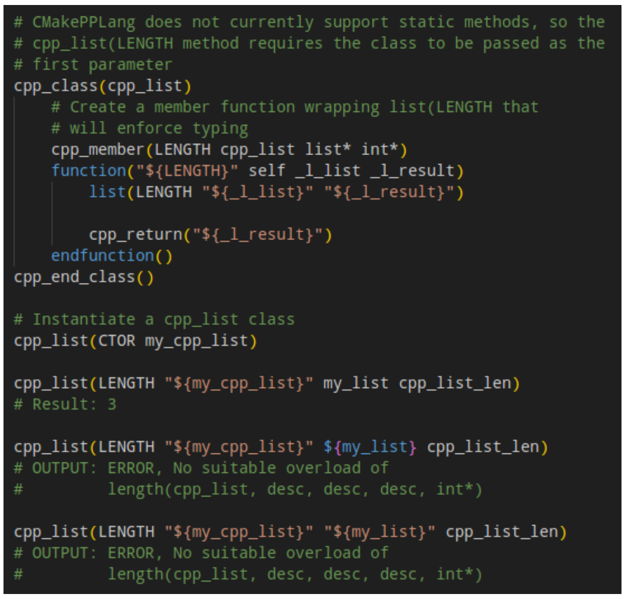
Figure 1: Example of including CMakePPLang in an existing CMake file.

Native CMake is a weakly typed language where all values are strings, and, in certain circumstances, select values are interpreted as being of another type. A common example is when a string is used as an argument to CMake's `if` statement, where the string is implicitly cast to a boolean. In practice, this weak typing can lead to subtle, hard-to-detect errors. CMakePPLang implements strong-typing in order to avoid/catch such errors. An example of weak typing causing issues is the ambiguity when passing a list as an argument to a function, which many CMake users are likely familiar with. In CMake's `list(LENGTH` function, there are three different ways to pass a list to the function, yielding three different results as seen in Figure 2. Looking at the function signature, `list(LENGTH <list> <output variable>)` (*CMake List Length*, 2023), it is unclear which version to use without trial and error. Conversely, using strong typing with CMakePPLang (Figure 3), it is immediately clear that the variable pointing to the list should be used from the types of the signature, `cpp_list(LENGTH cpp_list list* int*)`, where `list*` is a pointer to a list (colloquially it is the variable containing a list) and `int*` is a pointer to an integer where the resulting length will be stored. The other options now throw errors that prompt the user to reconsider the function signature and types being passed in.



```
# Create a short list: "a;b;c" stored in "my_list"
set(my_list a b c)

list(LENGTH my_list my_list_len)
# Result: 3

list(LENGTH ${my_list} my_list_len)
# Result: ERROR, list sub-command LENGTH requires two arguments

list(LENGTH "${my_list}" my_list_len)
# Result: 0
```

Figure 2: Three different methods of passing a list to `list(LENGTH`, showing the results of each call in a comment on the following line. Only the first call returns the correct list length of three.



```
# CMakePPLang does not currently support static methods, so the
# cpp_list(LENGTH method requires the class to be passed as the
# first parameter
cpp_class(cpp_list)
    # Create a member function wrapping list(LENGTH that
    # will enforce typing
    cpp_member(LENGTH cpp_list list* int*)
    function("${LENGTH}" self _l_list _l_result)
        list(LENGTH "${_l_list}" "${_l_result}")

        cpp_return("${_l_result}")
    endfunction()
cpp_end_class()

# Instantiate a cpp_list class
cpp_list(CTOR my_cpp_list)

cpp_list(LENGTH "${my_cpp_list}" my_list cpp_list_len)
# Result: 3

cpp_list(LENGTH "${my_cpp_list}" ${my_list} cpp_list_len)
# OUTPUT: ERROR, No suitable overload of
#         length(cpp_list, desc, desc, desc, int*)

cpp_list(LENGTH "${my_cpp_list}" "${my_list}" cpp_list_len)
# OUTPUT: ERROR, No suitable overload of
#         length(cpp_list, desc, desc, desc, int*)
```

Figure 3: Three different methods of passing a list to a function, `cpp_list(LENGTH`, which wraps `list(LENGTH` and provides strong typing. Results of each call are shown in a comment on the proceeding line. Only the first call is correct and the other two result in errors.

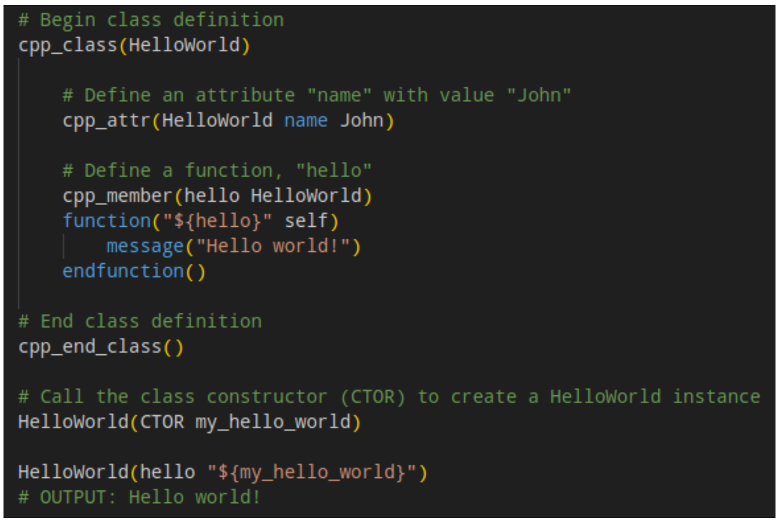
CMakePPLang conceptually has three classifications of types: CMake types, Quasi-CMake types, and pure CMakePPLang types.

First, CMakePPLang recognizes the types that CMake may interpret a string to be in certain contexts. These types include: Boolean, Command, File path, Floating-point numbers, Generator expressions, Integers, and Targets.

Quasi-CMake types are types which conceptually exist in traditional CMake, but are not explicitly defined. These types are: Description and Type. Descriptions are the subset of strings that have no other intrinsic type aside from being a string, *i.e.*, they can only be interpreted as text. Types are string values used to represent the type of an object, in CMakePPLang they amount to strings reserved for the in-code keywords representing a type, like `str` for a string, `int` for an integer, and `desc` for a description.

CMakePPLang also defines types that are outside of what can easily be represented in CMake: Class, Map, and Object. The Class type is used for objects which hold the specification of a user-defined type. Classes in CMakePPLang can contain attributes and functions and support inheritance. Instances of these user-defined classes can be created to be used in CMake modules. Currently, Classes are represented using Maps. An object of the Map type is an associative array for storing key-value pairs. The CMakePPLang Map provides the same basic functionality as a C++ `std::map` ("std::map," 2023), Python `dictionary`("Built-in Types," 2023), or JavaScript `Map` ("Map - JavaScript," 2023). Users can use maps in their code wherever they see fit, and maps are used in CMakePPLang to hold the state of object instances. Finally, the Object type is the base class for all user-defined classes. The CMakePPLang Object defines the default implementations for the equality, copy, and serialization functionalities.

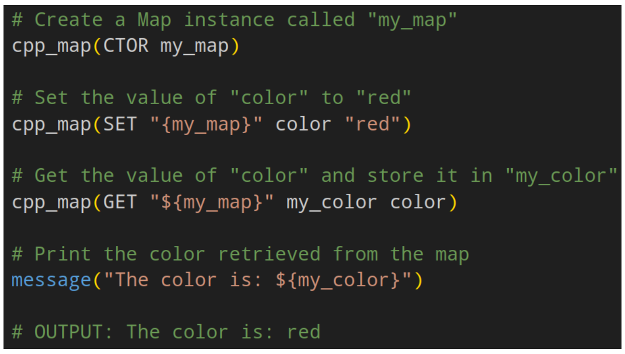
CMakePPLang is designed primarily to provide object-oriented funcionality for tools designed in CMake. The first step in this process is defining a class (Figure 4). Strong typing of the member function parameters can be seen in the example as well.



```
# Begin class definition
cpp_class(HelloWorld)

    # Define an attribute "name" with value "John"
    cpp_attr(HelloWorld name John)

    # Define a function, "hello"
    cpp_member(hello HelloWorld)
    function("${hello}" self)
        message("Hello world!")
    endfunction()

# End class definition
cpp_end_class()

# Call the class constructor (CTOR) to create a HelloWorld instance
HelloWorld(CTOR my_hello_world)

HelloWorld(hello "${my_hello_world}")
# OUTPUT: Hello world!
```

Figure 4: Example of defining a CMakePPLang class, creating an instance, and calling a member function to print "Hello world!".

Users can also define a map to hold information, like a map that stores a color value under the "color" key (Figure 5), along with other relevant values.



```
# Create a Map instance called "my_map"
cpp_map(CTOR my_map)

# Set the value of "color" to "red"
cpp_map(SET "{my_map}" color "red")

# Get the value of "color" and store it in "my_color"
cpp_map(GET "${my_map}" my_color color)

# Print the color retrieved from the map
message("The color is: ${my_color}")

# OUTPUT: The color is: red
```

Figure 5: Example of creating a CMakePPLang map, adding a key-value pair, and retrieving the value using the key.

Using these tools, CMakePPLang users can leverage the benefits of object- oriented programming to create more easily maintainable and testable utilities for CMake development.

Acknowledgement

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The software was developed in the NWChemEx subproject and the research was performed at the Ames National Laboratory, which is operated for the US DOE by Iowa State University under [Contract No. DE-AC02-07CH11358].

References

Ambler, S. (1998). A Realistic Look at Object-Oriented Reuse. In *Dr. Dobb's*. http://www.drdobbs.com/a-realistic-look-at-object-oriented-reus/184415594

Becker, T., Hück, A., Sánchez, M., Baratov, R., Loitsch, F., & Remes, J. (2021). *CMake++*. https://github.com/toeb/cmakepp

Built-in Types: Mapping Types - dict. (2023). In *Python documentation*. https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

CMaize. (2022). CMakePP. https://github.com/CMakePP/CMaize

CMake. (2023). https://cmake.org/

CMake List Length. (2023). https://cmake.org/cmake/help/latest/command/list.html#length

CMakePP Organization. (2023). In *cmakepp.github.io*. https://cmakepp.github.io/

CMakeTest. (2023). CMakePP. https://github.com/CMakePP/CMakeTest

Map - JavaScript. (2023). In *JavaScript Reference*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Richard, R. M., Keipert, K., Waldrop, J., Keçeli, M., Williams-Young, D., Bair, R., Boschen, J., Crandall, Z., Gasperich, K., Mahmud, Q. I., Panyala, A., Valeev, E., Dam, H. van, Jong, W. A. de, & Windus, T. L. (2023). PluginPlay: Enabling exascale scientific software one module at a time. *The Journal of Chemical Physics*, *158*(18), 184801. https://doi.org/10.1063/5.0147903

Semantic Versioning 2.0.0. (2023). https://semver.org/

Software Quality. (2005). In *Software Engineering: A Practitioner's Approach* (7th ed., pp. 400–406). Palgrave Macmillan. ISBN: 978-0-07-301933-8

std::map. (2023). In *cplusplus.com*. https://cplusplus.com/reference/map/map/

Wirth, N. (2006). Good ideas, through the looking glass [computing history]. *Computer*, *39*(1), 28–39. https://doi.org/10.1109/MC.2006.20