**Static program analysis using Cppcheck**

Amariei G. Daniel

daniel.g.amariei@gmail.com

**1. Introduction**

The purpose of this document is to offer a concise view of program analysis, especially static program analysis, using *Cppcheck* as a tool. In order to accomplish this goal, we have structured the document as follows: *section 2* describes the levels at which a program analysis can be performed; *section 3* further details program analysis and possible approaches in performing it; *section 4* presents *Cppcheck*, both from an architectural and practical standpoints, while in *section 5* we conclude.

**2. Analysis levels**

In order to ensure the safety, robustness, efficiency, ease of maintenance, and the like, for a system, an analysis is required to be performed on the system. This analysis can be carried at different *architectural* levels. The complexity increases while higher in the hierarchy.

| Mission- / Business-level analysis |
| --- |
| System-level analysis |
| Technology-level analysis |
| Unit-level analysis |

*Unit-level analysis* is performed on *a single unit of code* (*e.g.*,a method, a function, etc.). The *technology-level analysis* takes into account the interaction between code units, thus it happens one *an integrated collection of code units written in the same language.* Moving up the hierarchy, the *system-level analysis* takes place *on all the different code units and different layers of technology*. While at the top of the hierarchy, the *business-level analysis* takes into account the *business and processes that take place in the system from an operational point of view* (*i.e.*, business logic).

For more details please consult [2, 3].

**3. Program analysis**

*Program analysis is the process of automatically analyzing the behavior of computer programs* [1]. The main benefits of performing program analysis are program correctness and program optimization. There are two main approaches that are employed in order to analyse a program:

1. static program analysis;
2. dynamic program analysis.

Since *Cppcheck*is a tool that performs static program analysis, we will consider only this topic, although at a very basic level.

*Static program analysis* is the analysis of computer software without actually executing programs. This analysis can be performed on the actual source code, object code, or intermediate code.

**4. Cppcheck**

The purpose of this section is to present *Cppcheck*. Its presentation will be pursued from two perspectives: first the architectural one and then the practical one.

**Design (Architecture)**

The internal flow of data in *Cppcheck* used by checks happens as follows:



The first step of the process involves *preprocessing* the source files. Next, the preprocessed source code is *tokenized*, producing a list of tokens. Further, the tokens in the token list are used as nodes in a *syntax tree*. The next step in the process involves creating a *symbol database* which stores information about types, function, variables and the like, found in the source code. After the *symbol database* is constructed, a *value flow* analysis is made; this analysis involves tracking – forwards and backwards – the possible values for all the variables. After this stage completes, each token will have a list of possible value attached to it. The next step involves extracting the information from the *library*; the *library* contains information in the form of configuration files provided by the user. After all the previous steps are completed, the *checkers* can be executed, producing the result of the analysis.

For more details we recommend [4].

**Code analysis**

*Cppcheck* is a tool for static C/C++ code analysis. The philosophy of the tool is to provide zero false positives (*i.e.*, a report of an error, even though there is none). Unlike other types of tools and C/C++ compilers, *Cppcheck* does not detect syntax errors, focusing on the following:

* Out of bounds;
* Exception safety;
* Memory leaks;
* Obsolete functions usage;
* Invalid usage of STL;
* Uninitialized variables;
* Unused function;
* And others.

As static program analysis has it’s limitations in conjunction with the fact that *Cppcheck* covers a limited subset from all the possible bugs, the usage of similar tools is recommended. Moreover, since *Cppcheck* is integrated with various development tools, (*e.g.*, Visual Studio, Eclipse, gedit, etc.) one can easily integrate it in the *build automation* process.

*Cppcheck* can be used either though the *Graphical User Interface* (*GUI*), or the *command line*. In this document we will detail the second approach.

*Command-line syntax*

cppcheck <options> <path>

The <path> parameter may specify either a file to check, or a directory, in which case it will check all the source file present in the directory.

Assuming we have the folder tests which contains the file test1.cpp, running the specified commands will produce the following results – the command should be executed while in the test folder:



> cppcheck test1.cpp

The output is structured as follows:



The severities include:

* *error* – used for specifying bugs;
* *warning* – used for defensive programming suggestions;
* *style* – used for specifying stylisting issues (*e.g.*, unused functions, redundant code, etc.);
* *performance* – used for specifying suggestions to improve code performance;
* *portability*– used for specifying portability warnings;
* *information*– used for specifying informational messages.

**Options**

The tool has various options including:

*Severities*

By default, only the messages corresponding to error severity are shown. We can use the --enable command to specify the severities we are interested in; using the argument all will display messages for all the severities available, if found.

*Re-formatting the output*

We have the ability to reformat the output using the --template=”format”option. The format argument is a string that can make use of the following format specifiers: callstack, file, id, line, message, severity. Following, we give an example of usage: [{severity}]:[{id}]:[{file}]:[{line}]: {message}. When passed as argument to the template option, it will act as a pattern, and all output will be formatted according to it.

*Multithreaded checking*

We can specify the number of threads to use when checking a file or a folder using the -j n option, where n specifies the number of threads to use.

*Generating XML output*

If the need arises, we have the possibility to generate the output in *XML* format using the --xml-version=2 option. An example of the previous output in *XML* format is:



*Using configuration files*

When external files are used, since *Cppcheck* does not know how the functions included in those files behave, it fails to detect different problems (*e.g.*, memory leaks, buffer overflows, null pointer dereferences, etc.). In order to tackle this inconvenient, one can create a configuration file and pass it as argument to the --library option. This files contain rules about how the functions contained in the files should behave (*e.g.*, return value, value range, function arguments restrictions, etc.).

*Suppressing listings*

*Cppcheck* offers the possibility to suppress certain types of errors. We have three possibilities to inform the checker that we want to suppress a certain type of error:

1. Command-line – using the --supress=id;
2. Listing suppression ids in a file and passing it as argument to the --suppressions command;
3. Inline suppressions, by adding comments that have the following syntax: cppcheck-suppress id.

*Rule files*

*Cppcheck* offers the option to the users to define their own rules. This is done using an *XML* file. In the file we will define the rules to watch out for using *regular expressions*. As such, whenever the pattern is matched, a message will be prompted containing: the *id*, *severity*, and the *summary* of the message. In order to use *rule files* the tool must be used with the --rule-file=path/to/rule/file option.



**5. Conclusions**

As software systems become more and more complex, ensuring the safety, robustness, efficiency, ease of maintenance, and the like for them becomes a daunting task. While no software system can achieve perfection we can, at least, come close to it using various helping tools. One such tool is *Cppcheck*, which is a static program analysis tool for C/C++.

A good addition to the *build automation* process is a tool like this which can help the programmer discover possible errors. As such, a good advice for C and C++ programmer would be: *Cppcheck you code!*

This document started with a presentation of the possible levels at which a program analysis can be made. After that we described at a basic level what *program analysis* is, following with a description of the architecture of *Cppcheck* and some details on how to effectively use the tool at the command line.

We must mention that this document is complementary to the presentation and the code files. All of them should be followed in order to get a rather complete view.

**A. References and bibliography**

1. \*\*\*, Wikipedia. *Program analysis*. https://en.wikipedia.org/wiki/Program\_analysis.
2. \*\*\*, Object Management Group. *How to Deliver Resilient, Secure, Efficient, and Easily Changed IT Systems in Line with CISQ Recommendations*. www.omg.org, http://www.omg.org/CISQ\_compliant\_IT\_Systemsv.4-3.pdf.
3. \*\*\*, Wikipedia. *Static program analysis*. https://en.wikipedia.org/wiki/Static\_program\_analysis
4. \*\*\*, Cppcheck, Static source code analysis tool for C and C++ code. *Cppcheck Design*. http://sourceforge.net/projects/cppcheck/files/Articles/
5. \*\*\*, Cppcheck, Static source code analysis tool for C and C++ code. *Cppcheck Design 2010*. http://sourceforge.net/projects/cppcheck/files/Articles/
6. \*\*\*, Cppcheck, Static source code analysis tool for C and C++ code. *Writing Cppcheck rules (Part 1)*. http://sourceforge.net/projects/cppcheck/files/Articles/
7. \*\*\*, Cppcheck, Static source code analysis tool for C and C++ code. *Writing Cppcheck rules (Part 2)*. http://sourceforge.net/projects/cppcheck/files/Articles/
8. \*\*\*, Cppcheck, Static source code analysis tool for C and C++ code. *Cppcheck Manual*. http://sourceforge.net/projects/cppcheck/files/Articles/
9. \*\*\*, Cppcheck. *A tool for static C/C++ code analysis*. http://cppcheck.sourceforge.net/
10. \*\*\*, sgi. *Download STL source code*. https://www.sgi.com/tech/stl/download.html