Cppcheck manual

Cppcheck team

Contents

Chapter 1

Introduction

Cppcheck is an analysis tool for C/C++ code. It provides unique code analysis to detect bugs and focuses on detecting undefined behaviour and dangerous coding constructs. The goal is to detect only real errors in the code (i.e. have very few false positives). Cppcheck is designed to be able to analyze your C/C++ code even if it has non-standard syntax (common in embedded projects).

Supported code and platforms:

- You can check non-standard code that contains various compiler extensions, inline assembly code, etc.
- Cppcheck should be compilable by any C++ compiler that handles the latest C++ standard.
- Cppcheck should work on any platform that has sufficient CPU and memory.

Please understand that there are limits of Cppcheck. Cppcheck is rarely wrong about reported errors. But there are many bugs that it doesn't detect.

You will find more bugs in your software by testing your software carefully, than by using Cppcheck. You will find more bugs in your software by instrumenting your software, than by using Cppcheck. But Cppcheck can still detect some of the bugs that you miss when testing and instrumenting your software.

Chapter 2

Getting started

2.1 GUI

It is not required but creating a new project file is a good first step. There are a few options you can tweak to get good results.

In the project settings dialog, the first option you see is “Import project”. It is recommended that you use this feature if you can. Cppcheck can import:

- Visual studio solution / project
- Compile database (can be generated from cmake/qbs/etc build files)
- Borland C++ Builder 6

When you have filled out the project settings and click on OK; the Cppcheck analysis will start.

2.2 Command line

2.2.1 First test

Here is a simple code

```
int main()
{
    char a[10];
    a[10] = 0;
    return 0;
}
```

If you save that into file1.c and execute:

```
cppcheck file1.c
```

The output from cppcheck will then be:

```
Checking file1.c...
[file1.c:4]: (error) Array 'a[10]' index 10 out of bounds
```

2.2.2 Checking all files in a folder

Normally a program has many source files. And you want to check them all. Cppcheck can check all source files in a directory:

```
cppcheck path
```

If “path” is a folder then cppcheck will recursively check all source files in this folder.

```
Checking path/file1.cpp...
1/2 files checked 50% done
Checking path/file2.cpp...
2/2 files checked 100% done
```

2.2.3 Check files manually or use project file

With Cppcheck you can check files manually, by specifying files/paths to check and settings. Or you can use a project file (cmake/visual studio/etc).

We don’t know which approach (project file or manual configuration) will give you the best results. It is recommended that you try both. It is possible that you will get different results so that to find most bugs you need to use both approaches.

Later chapters will describe this in more detail.

2.2.4 Check files matching a given file filter

With `--file-filter=<str>` you can set a file filter and only those files matching the filter will be checked.

For example: if you want to check only those files and folders starting from a subfolder src/ that start with “test” you have to type:

```
cppcheck src/ --file-filter=src/test*
```

Cppcheck first collects all files in src/ and will apply the filter after that. So the filter must start with the given start folder.

2.2.5 Excluding a file or folder from checking

To exclude a file or folder, there are two options. The first option is to only provide the paths and files you want to check.

```
cppcheck src/a src/b
```

All files under src/a and src/b are then checked.

The second option is to use -i, with it you specify files/paths to ignore. With this command no files in src/c are checked:

```
cppcheck -isrc/c src
```

This option is only valid when supplying an input directory. To ignore multiple directories supply the -i multiple times. The following command ignores both the src/b and src/c directories.

```
cppcheck -isrc/b -isrc/c
```

2.2.6 Clang parser

By default Cppcheck uses an internal C/C++ parser. However it is possible to use the Clang parser instead.

Install `clang`. Then use Cppcheck option `--clang`.

Technically, Cppcheck will execute `clang` with its `-ast-dump` option. The Clang output is then imported and converted into our normal Cppcheck format. And then normal Cppcheck analysis is performed on that.

2.3 Severities

The possible severities for messages are:

error

used when bugs are found

warning

suggestions about defensive programming to prevent bugs

style

stylistic issues related to code cleanup (unused functions, redundant code, constness, and such)

performance

Suggestions for making the code faster. These suggestions are only based on common knowledge. It is not certain you'll get any measurable difference in speed by fixing these messages.

portability

portability warnings. 64-bit portability. code might work different on different compilers. etc.

information

Configuration problems. The recommendation is to only enable these during configuration.

Chapter 3

Importing project

You can import some project files and build configurations into Cppcheck.

3.1 Cppcheck GUI project

You can import and use Cppcheck GUI project files in the command line tool:

```
cppcheck --project=foobar.cppcheck
```

The Cppcheck GUI has a few options that are not available in the command line directly. To use these options you can import a GUI project file. We want to keep the command line tool usage simple and limit the options by intention.

To ignore certain folders in the project you can use `-i`. This will skip analysis of source files in the `foo` folder.

```
cppcheck --project=foobar.cppcheck -ifoo
```

3.2 CMake

Generate a compile database:

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON .
```

The file `compile_commands.json` is created in the current folder. Now run Cppcheck like this:

```
cppcheck --project=compile_commands.json
```

To ignore certain folders you can use `-i`. This will skip analysis of source files in the `foo` folder.

```
cppcheck --project=compile_commands.json -ifoo
```

3.3 Visual Studio

You can run Cppcheck on individual project files (*.vcxproj) or on a whole solution (*.sln)

Running Cppcheck on an entire Visual Studio solution:

```
cppcheck --project=foobar.sln
```

Running Cppcheck on a Visual Studio project:

```
cppcheck --project=foobar.vcxproj
```

Both options will analyze all available configurations in the project(s). Limiting on a single configuration:

```
cppcheck --project=foobar.sln "--project-configuration=Release|Win32"
```

In the `Cppcheck GUI` you have the choice to only analyze a single debug configuration. If you want to use this choice on the command line then create a `Cppcheck GUI` project with this activated and then import the GUI project file on the command line.

To ignore certain folders in the project you can use `-i`. This will skip analysis of source files in the `foo` folder.

```
cppcheck --project=foobar.vcxproj -ifoo
```

3.4 C++ Builder 6

Running Cppcheck on a C++ Builder 6 project:

```
cppcheck --project=foobar.bpr
```

To ignore certain folders in the project you can use `-i`. This will skip analysis of source files in the `foo` folder.

```
cppcheck --project=foobar.bpr -ifoo
```

3.5 Other

If you can generate a compile database then it's possible to import that in Cppcheck.

In Linux you can use for instance the `bear` (build ear) utility to generate a compile database from arbitrary build tools:

```
bear make
```

Chapter 4

Platform

You should use a platform configuration that match your target.

By default Cppcheck uses native platform configuration that works well if your code is compiled and executed locally.

Cppcheck has builtin configurations for Unix and Windows targets. You can easily use these with the `--platform` command line flag.

You can also create your own custom platform configuration in a XML file. Here is an example:

```
<?xml version="1"?>
<platform>
  <char_bit>8</char_bit>
  <default-sign>signed</default-sign>
  <sizeof>
    <short>2</short>
    <int>4</int>
    <long>4</long>
    <long-long>8</long-long>
    <float>4</float>
    <double>8</double>
    <long-double>12</long-double>
    <pointer>4</pointer>
    <size_t>4</size_t>
    <wchar_t>2</wchar_t>
  </sizeof>
</platform>
```

Chapter 5

Preprocessor Settings

If you use `--project` then Cppcheck will use the preprocessor settings from the imported project. Otherwise you'll probably want to configure the include paths, defines, etc.

5.1 Defines

Here is a file that has 2 preprocessor configurations (with A defined and without A defined):

```
#ifdef A
    x = y;
#else
    x = z;
#endif
```

By default Cppcheck will check all preprocessor configurations (except those that have #error in them). So the above code will by default be analyzed both with `A` defined and without `A` defined.

You can use `-D` to change this. When you use `-D`, cppcheck will by default only check the given configuration and nothing else. This is how compilers work. But you can use `--force` or `--max-configs` to override the number of configurations.

Check all configurations:

```
cppcheck file.c
```

Only check the configuration A:

```
cppcheck -DA file.c
```

Check all configurations when macro A is defined

```
cppcheck -DA --force file.c
```

Another useful flag might be `-U`. It tells Cppcheck that a macro is not defined. Example usage:

```
cppcheck -UX file.c
```

That will mean that X is not defined. Cppcheck will not check what happens when X is defined.

5.2 Include paths

To add an include path, use `-I`, followed by the path.

Cppcheck's preprocessor basically handles includes like any other preprocessor. However, while other preprocessors stop working when they encounter a missing header, cppcheck will just print an information message and continues parsing the code.

The purpose of this behaviour is that cppcheck is meant to work without necessarily seeing the entire code. Actually, it is recommended to not give all include paths. While it is useful for cppcheck to see the declaration of a class when checking the implementation of its members, passing standard library headers is highly discouraged because it will result in worse results and longer checking time. For such cases, .cfg files (see below) are the better way to provide information about the implementation of functions and types to cppcheck.

Chapter 6

Suppressions

If you want to filter out certain errors you can suppress these.

Please note that if you see a false positive then we (the Cppcheck team) want that you report it so we can fix it.

6.1 Plain text suppressions

You can suppress certain types of errors. The format for such a suppression is one of:

```
[error id]:[filename]:[line]
[error id]:[filename2]
[error id]
```

The `error id` is the id that you want to suppress. The easiest way to get it is to use the –template=gcc command line flag. The id is shown in brackets.

The filename may include the wildcard characters * or ?, which match any sequence of characters or any single character respectively. It is recommended that you use "/" as path separator on all operating systems. The filename must match the filename in the reported warning exactly. For instance, if the warning contains a relative path then the suppression must match that relative path.

6.2 Command line suppression

The `--suppress=` command line option is used to specify suppressions on the command line. Example:

```
cppcheck --suppress=memleak:src/file1.cpp src/
```

6.3 Suppressions in a file

You can create a suppressions file. Example:

```
// suppress memleak and exceptNew errors in the file src/file1.cpp
memleak:src/file1.cpp
exceptNew:src/file1.cpp

// suppress all uninitvar errors in all files
uninitvar
```

Note that you may add empty lines and comments in the suppressions file.

You can use the suppressions file like this:

```
cppcheck --suppressions-list=suppressions.txt src/
```

6.4 XML suppressions

You can specify suppressions in a XML file. Example file:

```
<?xml version="1.0"?>
<suppressions>
  <suppress>
    <id>uninitvar</id>
    <fileName>src/file1.c</fileName>
    <lineNumber>10</lineNumber>
    <symbolName>var</symbolName>
  </suppress>
</suppressions>
```

The XML format is extensible and may be extended with further attributes in the future.

You can use the suppressions file like this:

```
cppcheck --suppress-xml=suppressions.xml src/
```

6.5 Inline suppressions

Suppressions can also be added directly in the code by adding comments that contain special keywords. Before adding such comments, consider that the code readability is sacrificed a little.

This code will normally generate an error message:

```
void f() {
    char arr[5];
    arr[10] = 0;
}
```

The output is:

```
cppcheck test.c
[test.c:3]: (error) Array 'arr[5]' index 10 out of bounds
```

To activate inline suppressions:

```
cppcheck --inline-suppr test.c
```

6.5.1 Format

You can suppress a warning `aaaa` with:

```
// cppcheck-suppress aaaa
```

Suppressing multiple ids in one comment by using []:

```
// cppcheck-suppress [aaaa, bbbb]
```

6.5.2 Comment before code or on same line

The comment can be put before the code or at the same line as the code;

Before the code:

```
void f() {
    char arr[5];

    // cppcheck-suppress arrayIndexOutOfBounds
    arr[10] = 0;
}
```

Or at the same line as the code:

```
void f() {
    char arr[5];

    arr[10] = 0;  // cppcheck-suppress arrayIndexOutOfBounds
}
```

6.5.3 Multiple suppressions

For a line of code there might be several warnings you want to suppress.

There are several options;

Using 2 suppression comments before code:

```
void f() {
    char arr[5];

    // cppcheck-suppress arrayIndexOutOfBounds
    // cppcheck-suppress zerodiv
    arr[10] = arr[10] / 0;
}
```

Using 1 suppression comment before the code:

```
void f() {
    char arr[5];

    // cppcheck-suppress[arrayIndexOutOfBounds,zerodiv]
    arr[10] = arr[10] / 0;
}
```

Suppression comment on the same line as the code:

```
void f() {
    char arr[5];

    arr[10] = arr[10] / 0;  // cppcheck-suppress[arrayIndexOutOfBounds,zerodiv]
}
```

6.5.4 Symbol name

You can specify that the inline suppression only applies to a specific symbol:

```
// cppcheck-suppress aaaa symbolName=arr
```

Or

```
// cppcheck-suppress[aaaa symbolName=arr, bbbb]
```

6.5.5 Comment about suppression

You can write comments about a suppression like so:

```
// cppcheck-suppress[warningid] some comment
// cppcheck-suppress warningid ; some comment
// cppcheck-suppress warningid // some comment
```

Chapter 7

XML output

Cppcheck can generate output in XML format. Use `--xml` to enable this format.

A sample command to check a file and output errors in the XML format:

```
cppcheck --xml file1.cpp
```

Here is a sample report:

```
<?xml version="1.0" encoding="UTF-8"?>
<results version="2">
  <cppcheck version="1.66"/>
  <errors>
    <error id="someError" severity="error" msg="short error text"
       verbose="long error text" inconclusive="true" cwe="312">
      <location file0="file.c" file="file.h" line="1"/>
   </error>
  </errors>
</results>
```

7.1 The `<error>` element

Each error is reported in a `<error>` element. Attributes:

id

id of error. These are always valid symbolnames.

severity

error/warning/style/performance/portability/information

msg

the error message in short format

verbose

the error message in long format

inconclusive

this attribute is only used when the error message is inconclusive

cwe

CWE ID for the problem. This attribute is only used when the CWE ID for the message is known.

7.2 The `<location>` element

All locations related to an error are listed with `<location>` elements. The primary location is listed first.

Attributes:

file

filename. both relative and absolute paths are possible.

file0

name of the source file (optional)

line

line number

info

short information for each location (optional)

Chapter 8

Reformatting the text output

If you want to reformat the output so it looks different you can use templates.

8.1 Predefined output formats

To get Visual Studio compatible output you can use –template=vs:

```
cppcheck --template=vs samples/arrayIndexOutOfBounds/bad.c
```

This output will look like this:

```
Checking samples/arrayIndexOutOfBounds/bad.c ...
samples/arrayIndexOutOfBounds/bad.c(6): error: Array 'a[2]' accessed at index 2, which is ou
```

To get gcc compatible output you can use –template=gcc:

```
cppcheck --template=gcc samples/arrayIndexOutOfBounds/bad.c
```

The output will look like this:

```
Checking samples/arrayIndexOutOfBounds/bad.c ...
samples/arrayIndexOutOfBounds/bad.c:6:6: warning: Array 'a[2]' accessed at index 2, which is
a[2] = 0;
  ^
```

8.2 User defined output format (single line)

You can write your own pattern. For instance, to get warning messages that are formatted like old gcc such format can be used:

```
cppcheck --template="{file}:{line}: {severity}: {message}" samples/arrayIndexOutOfBounds/bad
```

The output will look like this:

```
Checking samples/arrayIndexOutOfBounds/bad.c ...
samples/arrayIndexOutOfBounds/bad.c:6: error: Array 'a[2]' accessed at index 2, which is out
```

A comma separated format:

```
cppcheck --template="{file},{line},{severity},{id},{message}" samples/arrayIndexOutOfBounds/
```

The output will look like this:

```
Checking samples/arrayIndexOutOfBounds/bad.c ...
samples/arrayIndexOutOfBounds/bad.c,6,error,arrayIndexOutOfBounds,Array 'a[2]' accessed at i
```

8.3 User defined output format (multi line)

Many warnings have multiple locations. Example code:

```
void f(int *p)
{
    *p = 3;       // line 3
}

int main()
{
    int *p = 0;   // line 8
    f(p);         // line 9
    return 0;
}
```

There is a possible null pointer dereference at line 3. Cppcheck can show how it came to that conclusion by showing extra location information. You need to use both –template and –template-location at the command line.

Example command:

```
cppcheck --template="{file}:{line}: {severity}: {message}\n{code}" --template-location="{fil
```

The output from Cppcheck is:

```
Checking multiline.c ...
multiline.c:3: warning: Possible null pointer dereference: p
    *p = 3;
     ^
multiline.c:8: note: Assignment 'p=0', assigned value is 0
    int *p = 0;
             ^
multiline.c:9: note: Calling function 'f', 1st argument 'p' value is 0
    f(p);
      ^
```

```
multiline.c:3: note: Null pointer dereference
    *p = 3;
     ^
```

The first line in the warning is formatted by the –template format.

The other lines in the warning are formatted by the –template-location format.

8.3.1 Format specifiers for –template

The available specifiers for –template are:

{file}

File name

{line}

Line number

{column}

Column number

{callstack}

Write all locations. Each location is written in [{file}:{line}] format and the locations are separated by ->. For instance it might look like: [multiline.c:8] -> [multiline.c:9] -> [multiline.c:3]

{inconclusive:text}

If warning is inconclusive then the given text is written. The given text can be any arbitrary text that does not contain }. Example: {inconclusive:inconclusive,}

{severity}

error/warning/style/performance/portability/information

{message}

The warning message

{id}

Warning id

{code}

The real code.

\t

Tab

\n

Newline

\r

Carriage return

8.3.2 Format specifiers for –template-location

The available specifiers for `--template-location` are:

{file}

File name

{line}

Line number

{column}

Column number

{info}

Information message about current location

{code}

The real code.

\t

Tab

\n

Newline

\r

Carriage return

Chapter 9

Addons

Addons are scripts that analyses Cppcheck dump files to check compatibility with secure coding standards and to locate various issues.

Cppcheck is distributed with a few addons which are listed below.

9.1 Supported addons

9.1.1 cert.py

cert.py checks for compliance with the safe programming standard SEI CERT.

9.1.2 misra.py

misra.py is used to verify compliance with MISRA C 2012 - a proprietary set of guidelines to avoid such questionable code, developed for embedded systems.

Since this standard is proprietary, cppcheck does not display error text by specifying only the number of violated rules (for example, [c2012-21.3]). If you want to display full texts for violated rules, you will need to create a text file containing MISRA rules, which you will have to pass when calling the script with `--rule-texts` key. Some examples of rule texts files available in tests directory.

You can also suppress some unwanted rules using `--suppress-rules` option. Suppressed rules should be set as comma-separated listed, for example: `--suppress-rules 21.1,18.7`. The full list of supported rules is available on Cppcheck home page.

9.1.3 y2038.py

y2038.py checks Linux system for year 2038 problem safety. This required modified environment. See complete description here.

9.1.4 threadsafety.py

threadsafety.py analyse Cppcheck dump files to locate thread safety issues like static local objects used by multiple threads.

9.2 Running Addons

Addons could be run through Cppcheck command line utility as follows:

```
cppcheck --addon=misra.py somefile.c
```

This will launch all Cppcheck checks and additionally calls specific checks provided by selected addon.

Some addons need extra arguments. You can configure how you want to execute an addon in a json file. For example put this in misra.json:

```
{
    "script": "misra.py",
    "args": [
        "--rule-texts=misra.txt",
        "--suppress-rules 17.3,21.12"
    ]
}
```

And then the configuration can be executed on the cppcheck command line:

```
cppcheck --addon=misra.json somefile.c
```

By default Cppcheck would search addon at standard path which was specified in installation process. You also can set this path directly, for example:

```
cppcheck --addon=/opt/cppcheck/configurations/my_misra.json somefile.c
```

This allows you create and manage multiple configuration files for different projects.

Chapter 10

Library configuration

When external libraries are used, such as WinAPI, POSIX, gtk, Qt, etc, Cppcheck doesn't know how the external functions behave. Cppcheck then fails to detect various problems such as leaks, buffer overflows, possible null pointer dereferences, etc. But this can be fixed with configuration files.

Cppcheck already contains configurations for several libraries. They can be loaded as described below. Note that the configuration for the standard libraries of C and C++, std.cfg, is always loaded by cppcheck. If you create or update a configuration file for a popular library, we would appreciate if you upload it to us.

10.1 Using your own custom .cfg file

You can create and use your own .cfg files for your projects. Use `--check-library` and `--enable=information` to get hints about what you should configure.

You can use the `Library Editor` in the `Cppcheck GUI` to edit configuration files. It is available in the `View` menu.

The .cfg file format is documented in the `Reference: Cppcheck .cfg format` (http://cppcheck.sf.net/reference-cfg-format.pdf) document.

Chapter 11

HTML Report

You can convert the XML output from cppcheck into a HTML report. You'll need Python and the pygments module (http://pygments.org/) for this to work. In the Cppcheck source tree there is a folder htmlreport that contains a script that transforms a Cppcheck XML file into HTML output.

This command generates the help screen:

```
htmlreport/cppcheck-htmlreport -h
```

The output screen says:

```
Usage: cppcheck-htmlreport [options]

Options:
   -h, --help      show this help message and exit
   --file=FILE     The cppcheck xml output file to read defects from.
                   Default is reading from stdin.
   --report-dir=REPORT_DIR
                   The directory where the html report content is written.
   --source-dir=SOURCE_DIR
                   Base directory where source code files can be found.
```

An example usage:

```
./cppcheck gui/test.cpp --xml 2> err.xml
htmlreport/cppcheck-htmlreport --file=err.xml --report-dir=test1 --source-dir=.
```

Chapter 12

Bug hunting

If you want to detect most bugs and can accept false alarms then Cppcheck has analysis for that.

This analysis is “soundy”; it should diagnose most bugs reported in CVEs and from dynamic analysis.

You have to expect false alarms. However Cppcheck tries to limit false alarms. The purpose of the data flow analysis is to limit false alarms.

Some possible use cases; * you are writing new code and want to ensure it is safe. * you are reviewing code and want to get hints about possible UB. * you need extra help troubleshooting a weird bug. * you want to check if a release candidate is safe.

The intention is that this will be used primarily in the GUI.

12.1 Activate this analysis

On the command line you can use `--bug-hunting` however then you can’t configure contracts (see below).

In the GUI goto the project dialog. In the `Analysis` tab there is a check box for `Bug hunting`.

12.2 Cppcheck contracts

To handle false alarms and improve the analysis you are encouraged to use contracts.

You can use Cppcheck contracts both for C and C++ code.

Example code:

```
int foo(int x)
{
    return 100 / x;
}
```

A division by zero would not be impossible so Cppcheck will diagnose it:

```
[test1.cpp:3] (error) There is division, cannot determine that there can't be a division by
```

This Cppcheck contract will silence that warning:

```
function: foo(x)
expects: x > 0
```

That contract will improve the intra procedural analysis. Every time `foo` is called it will be checked that the contract is satisfied:

```
void bar(int x)
{
    foo(x);
}
```

Cppcheck will warn:

```
[test1.cpp:10] (error) Function 'foo' is called, can not determine that its contract 'x>0'
```

12.3 Adding a contract in the GUI

There are two ways: * Open the “Contracts” tab at the bottom of the screen. Find the function in the listbox and double click on it. * Right click on a warning and click on “Edit contract..” in the popup menu. This popup menu item is only available if the warning is not inconclusive.

12.4 Incomplete analysis

The data flow analysis can analyze simple functions completely but complex functions are not analyzed completely (yet). The data flow analysis will be continously improved in the future but it will never be perfect.

It is likely that you will get false alarms caused by incomplete data flow analysis. Unfortunately it is unlikely that such false alarms can be fixed by contracts.