Cppcheck Premium manual

Cppcheck team, Cppcheck Solutions AB

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, and generate as few false positives (wrongly reported warnings) as possible. Cppcheck is designed to analyze your C/C++ code even if it has non-standard syntax, as is common in for example embedded projects.

Supported code and platforms:

- Cppcheck checks non-standard code that contains various compiler extensions, inline assembly code, etc.
- Cppcheck should be compilable by any compiler that supports C++11 or later.
- Cppcheck is cross-platform and is used in various posix/windows/etc environments.

The checks in Cppcheck are not perfect. There are bugs that should be found, that Cppcheck fails to detect.

About static analysis

The kinds of bugs that you can find with static analysis are:

- Undefined behavior
- Using dangerous code patterns
- Coding style

There are many bugs that you can not find with static analysis. Static analysis tools do not have human knowledge about what your program is intended to do. If the output from your program is valid but unexpected, then in most cases this is not detected by static analysis tools. For instance, if your small program writes “Helo” on the screen instead of “Hello”, it is unlikely that any tool will complain about that.

Static analysis should be used as a complement in your quality assurance. It does not replace any of:

- Careful design
- Testing
- Dynamic analysis
- Fuzzing

Getting started

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, which can be generated from CMake/qbs/etc build files
- Borland C++ Builder 6

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

Command line

First test

Here is some 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
```

Checking all files in a folder

Normally a program has many source files. 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
```

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 build environment, such as CMake or Visual Studio.

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 the largest amount of bugs you need to use both approaches. Later chapters will describe this in more detail.

Check files matching a given file filter

With `--file-filter=<str>` you can configure file filter(s) and then only those files matching the filter will be checked.

You can use `**`, `*` and `?` in the file filter pattern.

- `**` matches zero or more characters, including path separators.
- `*` matches zero or more characters, excluding path separators.
- `?` matches any single character except path separators.

For example, this command below means that `src/test1.cpp` could be checked, but `src/file2.cpp` and `src/test/file1.cpp` will not be checked:

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

Cppcheck first collects all files in the specified directory, then applies the filter. Therefore, the filter pattern must include the directory path you specified.

A common use case for `--file-filter` is to check a project, but only check certain files:

```
cppcheck --project=compile_commands.json --file-filter=src/*.c
```

Typically a `compile_commands.json` contains absolute paths. However no matter if `compile_commands.json` contains absolute paths or relative paths, the option `--file-filter=src/*.c` would mean that:

- a file with relative path `test1.c` is not checked.
- a file with relative path `src/test2.c` can be checked.
- a file with relative path `src/test3.cpp` is not checked.

Ignore files matching a given pattern

With `-i <str>` you can configure filename/directory patterns that should be ignored.

A file that is ignored will not be checked directly (the complete translation unit is skipped). Any header `#include`'d from a source file which is not ignored is checked indirectly, regardless if the header is ignored.

> *Note*: If you want to filter out warnings for a header file, then `-i` will not work. Use suppressions instead.

You can use `**`, `*` and `?` in the pattern to specify excluded folders/files.

- `**` matches zero or more characters, including path separators.
- `*` matches zero or more characters, excluding path separators.
- `?` matches any single character except path separators.

A use case for `-i` is to check a project, but exclude certain files/folders:

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

Typically a `compile_commands.json` contains absolute paths. However no matter if `compile_commands.json` contains absolute paths or relative paths, the option `-itest` would mean that:

- a file with relative path `test1.cpp` can be checked.
- a file with relative path `test/somefile.cpp` is not checked

Clang parser (experimental)

By default Cppcheck uses an internal C/C++ parser. However there is an experimental option to use the Clang parser instead.

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

Cppcheck executes clang with the -ast-dump option, imports the output, converts it to Cppcheck's internal format, and then performs standard analysis.

You can also pass a custom Clang executable to the option by using for example `--clang=clang-10`. You can also pass it with a path. On Windows it will append the `.exe` extension unless you use a path.

Severities

The possible severities for messages are:

error

when code is executed there is either undefined behavior or other error, such as a memory leak or resource leak

warning

when code is executed there might be undefined behavior

style

stylistic issues, such as unused functions, redundant code, constness, operator precedence, possible mistakes.

performance

run time performance suggestions based on common knowledge, though it is not certain any measurable speed difference will be achieved by fixing these messages.

portability

portability warnings. Implementation defined behavior. 64-bit portability. Some undefined behavior that probably works "as you want", etc.

information

configuration problems, which does not relate to the syntactical correctness, but the used Cppcheck configuration could be improved.

Possible speedup analysis of template code

Cppcheck instantiates the templates in your code.

If your templates are recursive, this can lead to slow analysis and high memory usage. Cppcheck will write information messages when there are potential problems.

Example code:

```
template <int i>
void a()
{
    a<i+1>();
}

void foo()
{
    a<0>();
```

```
}
```

Cppcheck output:

```
test.cpp:4:5: information: TemplateSimplifier: max template
recursion (100) reached for template 'a<101>'. You might
want to limit Cppcheck recursion. [templateRecursion]
    a<i+1>();
    ^
```

As you can see Cppcheck has instantiated `a<i+1>` until `a<101>` was reached and then it bails out.

To limit template recursion, you can:

- add template specialisation
- configure Cppcheck, which can be done in the GUI project file dialog

Example code with template specialisation:

```
template <int i>
void a()
{
    a<i+1>();
}

void foo()
{
    a<0>();
}

#ifdef __cppcheck__
template<> void a<3>() {}
#endif
```

You can pass `-D__cppcheck__` when checking this code.

Cppcheck build folder

Using a Cppcheck build folder is not mandatory but it is recommended.

Cppcheck save analyzer information in that folder.

The advantages are:

- It speeds up the analysis as it makes incremental analysis possible. Only changed files are analyzed when you recheck.
- Whole program analysis also when multiple threads are used.

On the command line you configure that through `--cppcheck-build-dir=path`. Example:

```
mkdir b

# All files are analyzed
cppcheck --cppcheck-build-dir=b src

# Faster! Results of unchanged files are reused
cppcheck --cppcheck-build-dir=b src
```

In the GUI it is configured in the project settings.

Importing a project

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

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. The command line tool usage is kept intentionally simple and the options are therefore limited.

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

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

Compilation database (CMake etc.)

Many build systems can generate a compilation database (a JSON file containing compilation commands for each source file). Example `cmake` command to generate the file:

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

When you have a `compile_commands.json` file, you can run Cppcheck like this:

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

By default only 1 configuration is checked because that is consistent with the compilation. If you want to check more configurations, you can use `--max-configs` or `--force`. For example:

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

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
```

Visual Studio

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

Running Cppcheck on an entire Visual Studio solution:

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

Running Cppcheck on an entire Visual Studio 2026 solution:

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

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 option to only analyze a single debug configuration. If you want to use this option 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
```

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
```

Other

If you generate a compilation database, then it is possible to import that in Cppcheck.

Makefile

In Linux you can convert a Makefile to a `compile_commands.json` using for instance `bear` (build ear) utility:

```
bear -- make
```

If you don't use Linux, there are Python scripts that converts a Makefile into a compilation database.

Preprocessor settings

If you use `--project`, then Cppcheck will automatically use the preprocessor settings in the imported project file and likely you don't have to configure anything extra.

If you don't use `--project`, then a bit of manual preprocessor configuration might be required. However Cppcheck has automatic configuration of defines.

Automatic configuration of preprocessor defines

Cppcheck automatically test different combinations of preprocessor defines to achieve as high coverage in the analysis as possible.

Here is a file that has 3 bugs (when `x`, `y`, `z` are assigned).

```
#ifdef A
    x=100/0;
    #ifdef B
        y=100/0;
    #endif
#else
    z=100/0;
#endif

#ifndef C
#error C must be defined
#endif
```

The flag `-D` tells Cppcheck that a name is defined. Cppcheck will only analyze configurations that contain this define.

The flag `-U` tells Cppcheck that a name is not defined. Cppcheck will only analyze configurations that does not contain this define.

The flag `--force` and `--max-configs` is used to control how many combinations are checked. When `-D` is used, Cppcheck will only check 1 configuration unless these are used.

Example:

```
# test all configurations
# all bugs are found
cppcheck test.c

# only test configuration "-DA"
# No bug is found; because C is not defined the #error will cause a preprocessor error
cppcheck -DA test.c

# only test configuration "-DA -DC"
# The first bug is found
cppcheck -DA -DC test.c

# Test all configurations that does not define "A"
# The last bug is found
cppcheck -UA test.c

# All configurations with "-DA" are tested
# The two first bugs are found
cppcheck --force -DA test.c

# only test 1 valid configuration
# Bug(s) will be found
cppcheck --max-configs=1 test.c

# test 2 valid configurations with "X" defined.
# Bug(s) will be found
cppcheck --max-configs=2 -DX test.c
```

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 discouraged, because the analysis will not work fully and lead to a longer checking time. For such cases, `.cfg` files are the preferred way to provide information about the implementation of functions and types to Cppcheck, see

below for more information.

Platform

You should use a platform configuration that matches your target environment.

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>
```

C/C++ standard

Use `--std` on the command line to specify a C/C++ standard.

Cppcheck assumes that the code is compatible with the latest C/C++ standard, but it is possible to override this.

The available options are:

- c89: C code is C89 compatible
- c99: C code is C99 compatible
- c11: C code is C11 compatible
- c17: C code is C17 compatible
- c23: C code is C23 compatible (default)
- c++03: C++ code is C++03 compatible
- c++11: C++ code is C++11 compatible
- c++14: C++ code is C++14 compatible
- c++17: C++ code is C++17 compatible
- c++20: C++ code is C++20 compatible
- c++23: C++ code is C++23 compatible
- c++26: C++ code is C++26 compatible (default)

Cppcheck build dir

It's a good idea to use a Cppcheck build dir. On the command line use `--cppcheck-build-dir`. In the GUI, the build dir is configured in the project options.

Rechecking code will be much faster. Cppcheck does not analyse unchanged code. The old warnings are loaded from the build dir and reported again.

Whole program analysis does not work when multiple threads are used unless you use a Cppcheck build dir. For instance, the unusedFunction warnings require whole program analysis.

Suppressions

If you want to filter out certain errors from being generated, then it is possible to suppress these.

If you encounter a false positive, please report it to the Cppcheck team so that the issue can be fixed.

Plain text suppressions

The format for an error 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 id of a warning is shown in brackets in the normal Cppcheck text output.

The `error id` and `filename` patterns may contain `**`, `*` or `?`.

- `**` matches zero or more characters, including path separators.
- `*` matches zero or more characters, excluding path separators.
- `?` matches any single character except path separators.

It is recommended to use forward-slash `/` in the filename pattern as path separator on all operating systems.

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/
```

Suppressions in a file

You can create a suppressions file for example as follows:

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

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

Note that you may add empty lines and comments in the suppressions file. Comments must start with `#` or `//` and be at the start of the line, or after the suppression line.

The usage of the suppressions file is as follows:

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

XML suppressions

You can specify suppressions in a XML file, for example as follows:

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

The `id` and `fileName` patterns may contain `**`, `*` or `?`.

- `**` matches zero or more characters, including path separators.
- `*` matches zero or more characters, excluding path separators.
- `?` matches any single character except path separators.

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

The usage of the suppressions file is as follows:

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

The `<hash>` element

Cppcheck calculates a unique ID for each error, called the hash. The hash depends on the code that is related to the error, not on where that code is located. This means that the hash for an error stays the same even when unrelated code elsewhere in the file is added, removed or moved, shifting the line numbers around. The hash only changes if the related code itself is modified.

This makes hash-based suppressions more robust than line-based suppressions: once you have reviewed and suppressed a specific warning, the suppression keeps working even after the file is edited, as long as the offending code is not changed.

The hash for an error is included as the `hash` attribute in the XML output. You can copy that value into a `<hash>` element in a suppressions XML file:

```
<?xml version="1.0"?>
<suppressions>
  <suppress>
    <id>uninitvar</id>
    <fileName>src/file1.c</fileName>
    <hash>12345678</hash>
  </suppress>
</suppressions>
```

A suppression can use `<hash>` on its own, without `<id>`, to suppress a specific error regardless of its id. It is also possible to combine `<hash>` with `<id>`, `<fileName>`, `<lineNumber>` and `<symbolName>`; when several of these are specified, all of them must match for the suppression to apply.

Inline suppressions

Suppressions can also be added directly in the code by adding comments that contain special keywords. Note that adding comments sacrifices the readability of the code somewhat.

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
```

Format

You can suppress a warning `aaaa` with:

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

Suppressing multiple ids in one comment by using `[]`:

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

Suppressing warnings `aaaa` on a block of code:

```
// cppcheck-suppress-begin aaaa
...
// cppcheck-suppress-end aaaa
```

Suppressing multiple ids on a block of code:

```
// cppcheck-suppress-begin [aaaa, bbbb]
...
// cppcheck-suppress-end [aaaa, bbbb]
```

Suppressing warnings `aaaa` for a whole file:

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

Suppressing multiple ids for a whole file:

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

Suppressing warnings `aaaa` where macro is used:

```
// cppcheck-suppress-macro aaaa
#define MACRO ...
...
x = MACRO; // <- aaaa warnings are suppressed here
```

Suppressing multiple ids where macro is used:

```
// cppcheck-suppress-macro [aaaa, bbbb]
#define MACRO ...
...
x = MACRO; // <- aaaa and bbbb warnings are suppressed here
```

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
}
```

The suppression comment and the line of code may be separated by additional comments or empty lines:

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

    // cppcheck-suppress arrayIndexOutOfBounds

    arr[10] = 0;

    // cppcheck-suppress arrayIndexOutOfBounds
    // Set the tenth element of arr to zero
    arr[10] = 0;
}
```

In the example below there are 2 lines with code and 1 suppression comment. The suppression comment only applies to 1 line: `a = b + c;`.

```
void f() {
    a = b + c; // cppcheck-suppress abc
    d = e + f;
}
```

As a special case for backwards compatibility, if you have a `{` on its own line and a suppression comment after that, then that will suppress warnings for both the current and next line. This example will suppress `abc` warnings both for `{` and for `a = b + c;`:

```
void f()
{ // cppcheck-suppress abc
    a = b + c;
}
```

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]
}
```

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]
```

Comment about suppression

You can write comments about a suppression as follows:

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

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>
```

The `<error>` element

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

id

id of error, and which are 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; note that this attribute is only used when the CWE ID for the message is known

remark

Optional attribute. The related remark/justification from a remark comment.

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)

Reformatting the text output

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

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 out of bounds.
```

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 out of bounds. [arrayIndexOutOfBounds]
a[2] = 0;
  ^
```

User defined output format (single line)

You can write your own pattern. For instance:

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

The output will then look like this:

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

A comma separated format:

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

The output will look like this:

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

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, for example:

```
cppcheck \
--template="{file}:{line}: {severity}: {message}\n{code}" \
--template-location="{file}:{line}: note: {info}\n{code}" multiline.c
```

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.

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 text that does not contain `}`. Example: `{inconclusive:inconclusive,}`

{severity}

error/warning/style/performance/portability/information

{message}

The warning message

{id}

Warning id

{remark}

The remark text if a remark comment has been provided

{code}

The real code

\t

Tab

\n

Newline

\r

Carriage return

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 the current location

{code}

The real code

\t

Tab

\n

Newline

\r

Carriage return

Justifications for warnings in the report

You can add remark comments in the source code that justify why there is a warning/violation.

Such a remark comment shall:

- start with REMARK.
- can either be added above the source code that generates the warning, or after the code on the same line.

Example code:

```
void foo(void) {
    // REMARK Initialize x with 0
    int x = 0;
}
```

In Cppcheck text output, the remarks are not shown by default, you can use `--template` option `{remark}` to show remarks:

```
$ ./cppcheck --enable=style \
--template="{file}:{line}: {message} [{id}]\\n{remark}" test1.c

Checking test1.c ...
test1.c:4: Variable 'x' is assigned a value that is never used. [unreadVariable]
Initialize x with 0
```

In XML output, the comment text is provided in a `remark` attribute:

```
$ ./cppcheck --enable=style --xml test1.c
...
remark="Initialize x with 0"
...
```

Addons

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

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

Supported addons

namingng.py

`namingng.py` allows you to configure and check naming conventions.

You need to have a configuration file that defines your naming conventions. By default the filename `namingng.config.json` is used but there is an option so you can use any filename you want.

Example configuration of naming conventions:

```
{
    "RE_VARNAME": ["[a-z]*[a-zA-Z0-9_]*\\Z"],
    "RE_PRIVATE_MEMBER_VARIABLE": null,
    "RE_FUNCTIONNAME": ["[a-z0-9A-Z]*\\Z"],
    "_comment": "comments can be added to the config with underscore-prefixed keys",
    "include_guard": {
        "input": "path",
        "prefix": "GUARD_",
        "case": "upper",
        "max_linenr": 5,
        "RE_HEADERFILE": "[^/].*\\.h\\Z",
        "required": true
    },
    "var_prefixes": {"uint32_t": "ui32"},
    "function_prefixes": {"uint16_t": "ui16",
                          "uint32_t": "ui32"}
}
```

threadsafety.py

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

y2038.py

y2038.py checks source code for year 2038 problem safety.

Running addons

Addons can be executed with the --addon option:

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

Likewise, if you have created your own script, you can execute that:

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

You can configure how you want to execute an addon in a JSON file. For example:

```
{
    "script": "mychecks.py",
    "args": [
        "--some-option"
    ],
    "ctu": false
}
```

To use that JSON file to execute your addon, use the --addon option:

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

Cppcheck search for addons in the local folder first and then in the installation folder. A different path can be specified explicitly, for instance:

```
cppcheck --addon=path/to/my-addon.py somefile.c
```

Library configuration

When external libraries are used, such as WinAPI, POSIX, gtk, Qt, etc., Cppcheck has no information about functions, types, or macros contained in those libraries. Cppcheck then fails to detect various problems in the code, or might even abort the analysis. But this can be fixed by using the appropriate 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 supplied it to the Cppcheck project.

Using a `.cfg` file

To use a `.cfg` file shipped with Cppcheck, pass the `--library=<lib>` option. The table below shows the currently existing libraries:

`.cfg` file	Library	Comment
`avr.cfg`		
`bento4.cfg`	Bento4	
`boost.cfg`	Boost	
`bsd.cfg`	BSD	
`cairo.cfg`	cairo	
`cppcheck-lib.cfg`	Cppcheck	Used in selfcheck of the Cppcheck code base
`cppunit.cfg`	CppUnit	
`dpdk.cfg`		
`embedded_sql.cfg`		
`emscripten.cfg`		
`ginac.cfg`		
`gnu.cfg`	GNU	
`googletest.cfg`	GoogleTest	
`gtk.cfg`	GTK	

.cfg file	Library	Comment
icu.cfg		
kde.cfg	KDE	
libcerror.cfg	libcerror	
libcurl.cfg	libcurl	
libsigc++.cfg	libsigc++	
lua.cfg		
mfc.cfg	MFC	
microsoft_atl.cfg	ATL	
microsoft_sal.cfg	SAL annotations	
microsoft_unittest.cfg	CppUnitTest	
motif.cfg		
nspr.cfg		
ntl.cfg		
opencv2.cfg	OpenCV	
opengl.cfg	OpenGL	
openmp.cfg	OpenMP	
openssl.cfg	OpenSSL	
pcre.cfg	PCRE	
posix.cfg	POSIX	
python.cfg		
qt.cfg	Qt	
ruby.cfg		
sdl.cfg		
sfml.cfg		
sqlite3.cfg	SQLite	
std.cfg	C/C++ standard library	Loaded by default
tinyxml2.cfg	TinyXML-2	
vcl.cfg		
windows.cfg	Win32 API	
wxsqlite3.cfg		
wxsvg.cfg		
wxwidgets.cfg	wxWidgets	
zephyr.cfg		
zlib.cfg	zlib	

Creating a custom `.cfg` file

You can create and use your own `.cfg` files for your projects. Use `--check-library` 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`

(https://cppcheck.sourceforge.io/reference-cfg-format.pdf) document.

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|URL
                   Base directory where source code files can be found, or
                   a URL to a remote GitHub/GitLab repository including a
                   branch, e.g.:
                     --source-dir=https://github.com/<username>/<repo>/blob/<branch>/
```

Example usage:

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

or

```
cppcheck gui/test.cpp --xml 2> err.xml
cppcheck-htmlreport --file=err.xml --report-dir=test1 \
     --source-dir=https://github.com/<username>/<repo>/blob/<branch>/
```

Choosing between local annotated HTML and remote repository links

`cppcheck-htmlreport` supports two modes for linking to source files:

- Local annotated HTML files (default when `--source-dir` is a filesystem path)
- Remote GitHub/GitLab links (when `--source-dir` is a URL)

Pointing `--source-dir` to a filesystem path generates local annotated HTML files. This is useful when you need a fully self-contained report that works offline, includes inline annotations, and is ideal for small or medium projects where generation is fast. Using a remote GitHub/GitLab URL avoids generating per-file HTML and keeps the summary report lightweight and fast to produce. This mode is ideal when the source is already hosted online and local duplication is unnecessary. Remote mode is especially helpful when the HTML report may be public or widely distributed but the source code should remain private, since access control is handled by the hosting service. In general, local mode fits air-gapped environments, while remote mode works best for CI workflows and large or private repositories.

Check level

Reduced

The “reduced” check level performs a limited data flow analysis. If developers want to run Cppcheck directly during development and require faster results than “normal” provides, then this reduced checking can be an option.

Normal

The “normal” check level is chosen by default. Our aim is that this checking level will provide an effective checking in “reasonable” time.

The “normal” check level should be useful during active development:

- checking files while you edit them.
- block changes to the repo
- etc.

Exhaustive

When you can wait longer for the results you can enable the “exhaustive” checking, by using the option `--check-level=exhaustive`.

Exhaustive checking level should be useful for scenarios where you can wait for results. For instance:

- nightly builds
- etc.

Speeding up analysis

Limit preprocessor configurations

For performance reasons it might be a good idea to limit preprocessor configurations to check.

Limit ValueFlow: max `if` count

The command line option `--performance-valueflow-max-if-count` adjusts the max count for number of `if` in a function.

When that limit is exceeded, there is a limitation of data flow in that function. It is not drastic:

- Analysis of other functions are not affected.
- It's only for some specific data flow analysis, we have data flow analysis that is always executed.
- All checks are always executed. There can still be plenty of warnings in the limited function.

There is data flow analysis that slows down exponentially when number of `if` increase. And the limit is intended to avoid that analysis time explodes.

GUI options

In the GUI there are various options to limit analysis.

In the GUI:

- Open the project dialog.
- In the "Analysis" tab there are several options.

If you want to use these limitations on the command line too, you can import the GUI project file with `--project`.

Cppcheck Premium

Bug hunting

This is analysis that is more noisy than normal analysis. Most warnings will be false positives (Cppcheck will wrongly claim that there are bugs). The design goal is to not have more than roughly 5 to 10 false positives in each file.

It is not intended to be used in normal CI or regular static analysis by developers. The noise makes it useless for that.

It is intended to be used when you are looking for bugs and you really can accept noise. For example:

- You have developed a brand new feature and want to ensure that there are no bugs.
- Maybe as part of release testing your product you can run bug hunting on modified files.
- Etc.

Technically, analysis that is "sound" will detect all bugs. Analysis that is "soundy" has the goal to detect most bugs and it tries to keep the noise at an reasonable level.

The Cppcheck bug hunting analysis is "soundy".

Command:

```
cppcheck --premium=bughunting ...
```

Coding standards

Command to activate Autosar checkers:

```
cppcheck --premium=autosar ...
```

Command to activate Cert C checkers:

```
cppcheck --premium=cert-c ...
```

Command to activate Cert C++ checkers:

```
cppcheck --premium=cert-c++ ...
```

Command to activate Misra C 2012 checkers:

```
cppcheck --premium=misra-c-2012 ...
```

Command to activate Misra C 2023 checkers:

```
cppcheck --premium=misra-c-2023 ...
```

Command to activate Misra C 2025 checkers:

```
cppcheck --premium=misra-c-2025 ...
```

Command to activate Misra C++ 2008 checkers:

```
cppcheck --premium=misra-c++-2008 ...
```

Command to activate Misra C++ 2023 checkers:

```
cppcheck --premium=misra-c++-2023 ...
```

Checking all C and C++ files

The `cert-c` and `misra-c-*` coding standards target C and therefore the checkers only check C files by default.

The `autosar`, `cert-c++` and `misra-c++-*` coding standards target C++ and therefore the checkers only check C++ files by default.

If you want to check all files, you can append `:all` to the coding standard. Example:

```
# Misra C checkers are executed on C files, not on C++ files
cppcheck --premium=misra-c-2025 path

# Misra C checkers are executed on C and C++ files
cppcheck --premium=misra-c-2025:all path
```

Compliance report

Command line

There is a tool `compliance-report` that is distributed with Cppcheck Premium. To see all available options, run `compliance-report` in a terminal with no options.

Below is example usage to generate a Misra C 2012 compliance report.

```
cppcheck --premium=misra-c-2012 --xml-version=3 src 2> results.xml
```

```
compliance-report --misra-c-2012 --project-name=Device --project-version=2.3
--output-file=report.html results.xml
```

Description of the options:

- `--misra-c-2012`: Generate a compliance report for misra-c-2012
- `--project-name`: The name of the project
- `--project-version`: The version of the project
- `--output-file`: HTML filename that the report should be written to
- `results.xml`: The XML output from Cppcheck

Metrics

To generate metrics, add option `--premium=metrics`. The metrics are saved in the XML v3 report. Example:

```
cppcheck --premium=metrics test.c --xml-version=3 2> res.xml
```

We provide a small simple python script that creates a metrics report in CSV format:

```
python3 HISReport.py -f res.xml -j path/to/cppcheck-id-mapping.json -o test.csv
```

The `cppcheck-id-mapping.json` is provided in the Cppcheck premium installation folder, i.e. `/opt/cppcheckpremium` or `C:\Program Files\Cppcheck Premium`.

We do not have a ready-made solution to generate a HTML/pdf report. You can easily tweak our `HISReport.py` script so that it generates HTML and get the report exactly as you want.

Licenses

Commercial terms

Information about Cppcheck Premium licenses: https://www.cppcheck.com/plans-pricing

Installation / registration

This is described on the Cppcheck Premium website: https://www.cppcheck.com

License file path

There are predefined paths where the premium addon search for license files. If you want to provide an arbitrary license file path on the command line, you can use the option `--premium-license-file`. Example:

```
cppcheck --premium-license-file=path/to/file.lic test.cpp
```

If an explicit path is provided like this, then premium addon does not search for license files in the predefined paths.

Troubleshooting

Step 1: check `premiumaddon` debug output

If your license does not work, you can get some details about the license validation by executing `premiumaddon` binary with the `--debug` option.

Windows:

```
premiumaddon.exe --debug
```

Linux/Mac:

```
premiumaddon --debug
```

These commands can be executed from an arbitrary folder.

Step 2: clean up Cppcheck build dir

Command line: If you use `--cppcheck-build-dir`, then remove all files in the specified folder and recheck.

Cppcheck GUI: The GUI normally by default creates a Cppcheck build dir. Clear all results and recheck. You can clear all results by clicking on the brush icon in the toolbar. Or by open `Edit` menu and selecting menu item `Clear results`.

Step 3: remove `cppcheck-premium-loc` files

If you have `cppcheck-premium-loc` files in your project folders, those should be removed.

If such files are generated during analysis, then review your scripts to check why those files are generated.