Cppcheck manual

Version 2.6

Cppcheck team

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 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.

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, which specifies the 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 flag for each directory individually. The following command ignores both the src/b and src/c directories:

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

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`.

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

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 that uses a lot of memory. 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 temp
    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
cppcheck --cppcheck-build-dir=b src # <- All files are analyzed
cppcheck --cppcheck-build-dir=b src # <- Faster! Results of unchanged files are reused
```

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
```

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
```

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 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 can generate a compile database, then it is 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
```

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. There will be no Cppcheck analysis without this define. The flag `-U` tells Cppcheck that a name is not defined. There will be no Cppcheck analysis with 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:

```
cppcheck test.c => test all configurations => all bugs are found
```

```
cppcheck -DA test.c => only test configuration "-DA" => No bug is found (#error)
cppcheck -DA -DC test.c => only test configuration "-DA -DC" => The first bug is found
cppcheck -UA test.c => The configuration "-DC" is tested => The last bug is found
cppcheck --force -DA test.c => All configurations with "-DA" are tested => The two first bug
```

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 wor 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 (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 (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, then please report it to the Cppcheck team so that it 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 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 matches any sequence of characters or any single character respectively. It is recommended to 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.

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 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/
```

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]
```

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
}
```

In this example 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

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 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;
  ^
```

User defined output format (single line)

You can write your own pattern. For instance, to get warning messages that are formatted like traditional gcc, then the following format can be used:

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

The output will then 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
```

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="{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.

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

{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

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

cert.py

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

misra.py

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

This standard is proprietary, and open source tools are not allowed to distribute the Misra rule texts. Therefore Cppcheck is not allowed to write the rule texts directly. Cppcheck is allowed to distribute the rules and display the id of each violated rule (for example, [c2012-21.3]). The corresponding rule text can also be written however you need to provide that. To get the rule texts, please buy the PDF from MISRA (https://www.misra.org.uk). If you copy the rule texts from "Appendix A - Summary of guidelines" in the PDF and write those in a text file, then by using that text file Cppcheck can write the proper warning messages. To see how the text file can be formatted, take a look at the files listed here: https://github.com/danmar/cppcheck/blob/main/addons/test/misra/. You can use the option `--rule-texts` to specify your rules text file.

The full list of supported rules is available on Cppcheck home page.

y2038.py

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

threadsafety.py

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

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"
    ]
}
```

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 the standard path which was specified during the installation process. You also can set this path directly, for example:

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

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

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 memory 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.

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` (https://cppcheck.sf.net/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
                   Base directory where source code files can be found.
```

Example usage:

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

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.

Activate this analysis

On the command line you can use `--bug-hunting`. In the GUI go to the project dialog. In the `Analysis` tab there is a check box for `Bug hunting`.

Contracts

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

To provide contracts, you can either annotate your code or configure the contracts in the GUI.

There exists various annotations for C and C++ code. gcc has attributes, there are SAL annotations, and then there are standard C++ annotations. It is our goal to handle various types of annotations, if you can reuse those annotations in Cppcheck analysis that will be an extra benefit.

Function contracts

Here is an example code:

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

The bug hunting analysis will warn about a division by zero. Right now, it can't be proven that x can't be 0 here. A function contract can be used to tell Cppcheck what input "foo(x)" expects.

Annotation

You can use "C++ function contracts" syntax both in C and C++.

For C++ code you can write:

```
int foo(int x)
[[expects: x > 0]]
{
    return 100 / x;  // No division by zero
}

void bar()
{
    foo(-10);  // Warning: Contract is violated!
}
```

For C code you can write (works in C++ too):

```
#ifdef __cppcheck__
#define Expects(EXPR) [[expects: EXPR]]
#else
#define Expects(EXPR)
#endif

int foo(int x)
Expects(x > 0)
{
    return 100 / x;
}

void bar()
{
    foo(-10);  // Warning: Contract is violated!
}
```

Configuration in gui

You can configure contracts in the GUI.

Example code:

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

If you run bug hunting analysis on this code, then because Cppcheck can't prove that x can't be 0, you will get a warning about division by zero.

Either:

- Right click on that warning and select "Edit contract..".
- Open the "Functions" tab at the bottom and lookup the "foo(x)" function. Then double click on that.

A dialog box is shown where you can configure the contract for function "foo(x)". A textbox allows you to edit the "Expects" expression.

Enter the expression "x > 0" in the dialog box and click OK.

Now if you run analysis the division by zero warning will be gone. As for annotations, if the contract is violated somewhere then you will get a warning.

Variable contracts

Here is an example code:

```
int x;

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

The bug hunting analysis will warn about a division by zero. It can't be proven that x can't be 0.

A variable contract specify the allowed values for a variable. Cppcheck use variable contracts both when a variable is read and written: - When a variable is read, Cppcheck will assume that the contract is met. This means you can avoid false positives for impossible variable values. - When a variable is written, Cppcheck will ensure that its contract is not violated. If it can't be determined that the contract is met you will get a warning.

Annotation

You can use Cppcheck attributes `__cppcheck_low__(value)` and `__cppcheck_high__(value)` to configure min and max values for variables and types.

Example code:

```
__cppcheck_low__(1) int x;

int foo()
{
    return 100 / x;  // No division by zero
}
```

Tip: You can create an integer type with a limited value range. For instance here is an unsigned integer type that can only have the values 0-100:

```
typedef __cppcheck_high__(100) unsigned int percent_t;
percent_t x;
x = 110; // <- Cppcheck will warn about this assignment
```

GUI

To configure variable contracts in the GUI, open the "Variables" tab at the bottom.

Lookup the variable you want to configure and double click on that.

A dialog box is shown for the variable, where you can configure the min and max values.

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 continuously 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.