Cppcheck 1.61

Cppcheck 1.61

Table of Contents

Chapter 1. Introduction

Cppcheck is an analysis tool for C/C++ code. Unlike C/C++ compilers and many other analysis tools, it doesn't detect syntax errors. Cppcheck only detects the types of bugs that the compilers normally fail to detect. The goal is no false positives.

Supported code and platforms:

- You can check non-standard code that includes 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.

Accuracy

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. 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. 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 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
```

2.3. 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
```

2.4. 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

Informational messages about checking problems.

2.5. Enable messages

By default only *`error`* messages are shown. Through the *`--enable`* command more checks can be enabled.

```
# enable warning messages
cppcheck --enable=warning file.c
```

```
# enable performance messages
cppcheck --enable=performance file.c

# enable information messages
cppcheck --enable=information file.c

# For historical reasons, --enable=style enables warning, performance,
# portability and style messages. These are all reported as "style" when
# using the old xml format.
cppcheck --enable=style file.c

# enable warning and information messages
cppcheck --enable=warning,information file.c

# enable unusedFunction checking. This is not enabled by --enable=style
# because it doesn't work well on libraries.
cppcheck --enable=unusedFunction file.c

# enable all messages
cppcheck --enable=all
```

Please note that `--enable=unusedFunction` should only be used when the whole program is scanned. And therefore `--enable=all` should also only be used when the whole program is scanned. The reason is that the unusedFunction checking will warn if a function is not called. There will be noise if function calls are not seen.

2.5.1. Inconclusive checks

By default Cppcheck only writes error messages if it is certain. With *`--inconclusive`* error messages will also be written when the analysis is inconclusive.

```
cppcheck --inconclusive path
```

This can of course cause false warnings, it might be reported that there are bugs even though there are not. Only use this command if false warnings are acceptable.

2.6. Saving results in file

Many times you will want to save the results in a file. You can use the normal shell redirection for piping error output to a file.

```
cppcheck file1.c 2> err.txt
```

2.7. Multithreaded checking

The option -j is used to specify the number of threads you want to use. For example, to use 4 threads to check the files in a folder:

```
cppcheck -j 4 path
```

Chapter 3. Preprocessor configurations

By default Cppcheck will check all preprocessor configurations (except those that have #error in them).

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 undefines a symbol. Example usage:

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

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

Chapter 4. XML output

Cppcheck can generate the output in XML format. There is an old XML format (version 1) and a new XML format (version 2). Please use the new version if you can.

The old version is kept for backwards compatibility only. It will not be changed. But it will likely be removed someday. Use *--xml* to enable this format.

The new version fixes a few problems with the old format. The new format will probably be updated in future versions of cppcheck with new attributes and elements. A sample command to check a file and output errors in the new XML format:

```
cppcheck --xml-version=2 file1.cpp
```

Here is a sample version 2 report:

```
<?xml version="1.0" encoding="UTF-8"?>
<results version="2">
  <cppcheck version="1.53">
  <errors>
    <error id="someError" severity="error" msg="short error text"
           verbose="long error text" inconclusive="true">
      <location file="file.c" line="1"/>
    </error>
  </errors>
</results>
```

4.1. The <error> element

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

`id`

id of error. These are always valid symbolnames.

`severity`

either: `error`, `warning`, `style`, `performance`, `portability` or `information`

`msg`

the error message in short format

`verbose`

the error message in long format.

`inconclusive`

This attribute is only used when the message is inconclusive.

4.2. The <location> element

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

Attributes:

`file`

filename. Both relative and absolute paths are possible

`line`

a number

`msg`

this attribute doesn't exist yet. But in the future we may add a short message for each location.

Chapter 5. Reformatting the output

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

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

```
cppcheck --template=vs gui/test.cpp
```

This output will look like this:

```
Checking gui/test.cpp...
gui/test.cpp(31): error: Memory leak: b
gui/test.cpp(16): error: Mismatching allocation and deallocation: k
```

To get gcc compatible output you can use *--template=gcc*:

```
cppcheck --template=gcc gui/test.cpp
```

The output will look like this:

```
Checking gui/test.cpp...
gui/test.cpp:31: error: Memory leak: b
gui/test.cpp:16: error: Mismatching allocation and deallocation: k
```

You can write your own pattern (for example a comma-separated format):

```
cppcheck --template="{file},{line},{severity},{id},{message}" gui/test.cpp
```

The output will look like this:

```
Checking gui/test.cpp...
gui/test.cpp,31,error,memleak,Memory leak: b
gui/test.cpp,16,error,mismatchAllocDealloc,Mismatching allocation and deallocation: k
```

The following format specifiers are supported:

callstack

: callstack - if available

file

: filename

id

: message id

line
: line number

message
: verbose message text

severity
: severity

The escape sequences \b (backspace), \n (newline), \r (formfeed) and \t (horizontal tab) are supported.

Chapter 6. Suppressions

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

6.1. Suppressing a certain error type

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 *--xml* command line flag. Copy and paste the *id* string from the XML output. This may be * to suppress all warnings (for a specified file or files).

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.

6.1.1. 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/
```

6.1.2. Listing 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 suppressions.txt src/
```

6.2. 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
Checking test.c...
[test.c:3]: (error) Array 'arr[5]' index 10 out of bounds
```

To suppress the error message, a comment can be added:

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

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

Now the --inline-suppr flag can be used to suppress the warning. No error is reported when invoking cppcheck this way:

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

Chapter 7. Rules

You can define custom rules using regular expressions.

These rules can not perform sophisticated analysis of the code. But they give you an easy way to check for various simple patterns in the code.

To get started writing rules, see the related articles here:

http://sourceforge.net/projects/cppcheck/files/Articles/

The file format for rules is:

```
<?xml version="1.0"?>
<rule>
  <tokenlist>LIST</tokenlist>
  <pattern>PATTERN</pattern>
  <message>
    <id>ID</id>
    <severity>SEVERITY</severity>
    <summary>SUMMARY</summary>
  </message>
</rule>
```

7.1. <tokenlist>

The `<tokenlist>` element is optional. With this element you can control what tokens are checked. The `LIST` can be either `define`, `raw`, `normal` or `simple`.

define

used to check #define preprocessor statements.

raw

used to check the preprocessor output.

normal

used to check the `normal` token list. There are some simplifications.

simple

used to check the simple token list. All simplifications are used. Most Cppcheck checks use the simple token list.

If there is no <tokenlist> element then `simple` is used automatically.

7.2. <pattern>

The `PATTERN` is the `PCRE`-compatible regular expression that will be executed.

7.3. <id>

The ID specify the user-defined message id.

7.4. <severity>

The `SEVERITY` must be one of the `Cppcheck` severities: `information`, `performance`, `portability`, `style`, `warning`, or `error`.

7.5. <summary>

Optional. The summary for the message. If no summary is given, the matching tokens is written.

Chapter 8. Library configuration

`Cppcheck` has internal knowledge about how standard C/C++ functions work. There is no internal knowledge about how all libraries and environments work, and there can't be. `Cppcheck` can be told how libraries and environments work by using configuration files.

The idea is that users will be able to download library configuration files for all popular libraries and environments here:

http://cppcheck.sourceforge.net/archive

Ideally, all you need to do is choose and download the configuration files you need.

The archive is not complete however. If you can't find the configuration file you need in the archive, you can wait - maybe somebody else will write it and share it. Or you can write your own configuration file (and then it's possible to share your configuration file with others).

A minimal configuration file looks like this:

```
<?xml version="1.0"?>
<def>
</def>
```

This configuration file is filled up with various options:

```
<?xml version="1.0"?>
<def>
  <memory>
    <alloc>CreateFred</alloc>
    <dealloc>CloseFred</dealloc>

    <use>AppendFred</use>
  </memory>

  <memory>
    <alloc init="false">AllocWilma</alloc>
    <alloc init="true">CAllocWilma</alloc>
    <dealloc>CloseWilma</dealloc>
  </memory>

  <resource>
    <alloc>Lock</alloc>
    <dealloc>Unlock</dealloc>
  </resource>

  <function name="IsEqual">
```

```
    <leak-ignore/>
  </function>

  <function name="AssignFred">
    <noreturn>false</noreturn>
    <arg nr="1">
      <not-null/>
    </arg>
    <arg nr="2">
      <not-uninit/>
    </arg>
    <arg nr="3">
      <strz/>
    </arg>
    <arg nr="4">
      <formatstr/>
    </arg>
  </function>
</def>
```

In the `<memory>` and `<resource>` the allocation and deallocation functions are configured. Putting allocation and deallocation functions in different `<memory>` and `<resource>` blocks means they are mismatching - you'll get a warning message if you allocate memory with `CreateFred` and try to close it with `CloseWilma`.

The `<use>` and `<leak-ignore>` elements are used to control the leaks checking. If it should be ignored that a function is called, use `<leak-ignore>`. If there is no leak whenever the memory is passed to a function, use `<use>`.

In the `<function>` block some useful info is added about function behaviour. The `<noreturn>` tells `Cppcheck` if the function is a no return function. The `<arg>` is used to validate arguments. If it's invalid to pass NULL, use `<not-null>`. If it's invalid to pass uninitialized data, use `<not-uninit>`. If the argument is a zero-terminated string, use `<strz>`. If the argument is a formatstring, use `<formatstr>`.

8.1. Example: strcpy()

The strcpy() was chosen in this example for demonstration purposes because its behaviour is well-known.

The proper configuration for the standard strcpy() function would be:

```
  <function name="strcpy">
    <leak-ignore/>
    <noreturn>false</noreturn>
    <arg nr="1">
      <not-null/>
    </arg>
```

```
    <arg nr="2">
      <not-null/>
      <not-uninit/>
    </arg>
  </function>
```

The `<leak-ignore/>` is optional and it tells Cppcheck to ignore this function call in the leaks checking. Passing allocated memory to this function won't mean it will be deallocated.

The `<noreturn>` is optional. But it's recommended.

The first argument that the function takes is a pointer. It must not be a null pointer, a uninitialized pointer nor a dead pointer. It must point at some data, this data can be initialized but it is not wrong if it isn't. Using `<not-null>` is correct. `Cppcheck` will check by default that the pointer is not uninitialized nor dead.

The second argument the function takes is a pointer. It must not be null. And it must point at initialized data. Using `<not-null>` and `<not-uninit>` is correct.

Chapter 9. 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 10. Graphical user interface

10.1. Introduction

A Cppcheck GUI is available.

The main screen is shown immediately when the GUI is started.

10.2. Check source code

Use the Check menu.

10.3. Inspecting results

The results are shown in a list.

You can show/hide certain types of messages through the View menu.

Results can be saved to an XML file that can later be opened. See `Save results to file` and `Open XML`.

10.4. Settings

The language can be changed at any time by using the Language menu.

More settings are available in Edit⟶Preferences.

10.5. Project files

The project files are used to store project specific settings. These settings are:

- include folders
- preprocessor defines

As you can read in chapter 3 in this manual the default is that Cppcheck checks all configurations. So only provide preprocessor defines if you want to limit the checking.