Cppcheck 1.57

Cppcheck 1.57

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

portability

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

information

Informational messages that might be interesting. Ignore these messages unless you really agree.

* *The performance messages are based on 'common knowledge'. It is not certain that fixing performance messages will make any measurable difference in speed. Fixing performance messages generally doesn't make your code more readable.*

2.5. Enable messages

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

2.5.1. Stylistic issues

With *--enable=style* you enable most *warning*, *style* and *performance* messages.

Here is a simple code example:

```
void f(int x)
{
    int i;
    if (x == 0)
    {
        i = 0;
    }
}
```

There are no bugs in that code so Cppcheck won't report anything by default. To enable the stylistic messages, use the *--enable=style* command:

```
cppcheck --enable=style file3.c
```

The output from Cppcheck is now:

```
Checking file3.c...
[file3.c:3]: (style) Variable 'i' is assigned a value that is never used
[file3.c:3]: (style) The scope of the variable i can be reduced
```

2.5.2. Unused functions

This check will try to find unused functions. It is best to use this when the whole program is checked, so that all usages is seen by cppcheck.

```
cppcheck --enable=unusedFunction path
```

2.5.3. Enable all checks

To enable all checks your can use the *--enable=all* flag:

```
cppcheck --enable=all path
```

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 command line client can only use threads in posix environments. But it is the goal to be able to use threads on all platforms.

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). This is the recommended behaviour.

But if you want to manually limit the checking you can do so with `-D`.

Beware that only the macros, which are given here and the macros defined in source files and known header files are considered. That excludes all the macros defined in some system header files, which are by default not examined by Cppcheck.

The usage: if you, for example, want to limit the checking so the only configuration to check should be `DEBUG=1;__cplusplus` then something like this can be used:

```
cppcheck -DDEBUG=1 -D__cplusplus path
```

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
```

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. Leaks

Looking for memory leaks and resource leaks is a key feature of Cppcheck. Cppcheck can detect many common mistakes by default. But through some tweaking you can improve the checking.

7.1. User-defined allocation/deallocation functions

Cppcheck understands many common allocation and deallocation functions. But not all.

Here is example code that might leak memory or resources:

```
void foo(int x)
{
    void *f = CreateFred();
    if (x == 1)
        return;
    DestroyFred(f);
}
```

If you analyse that with Cppcheck it won't find any leaks:

```
cppcheck --enable=possibleError fred1.cpp
```

You can add some custom leaks checking by providing simple implementations for the allocation and deallocation functions. Write this in a separate file:

```
void *CreateFred()
{
    return malloc(100);
}

void DestroyFred(void *p)
{
    free(p);
}
```

When Cppcheck see this it understands that `CreateFred()` will return allocated memory and that `DestroyFred()` will deallocate memory.

Now, execute **cppcheck** this way:

```
cppcheck --append=fred.cpp fred1.cpp
```

The output from **cppcheck** is:

```
Checking fred1.cpp...
[fred1.cpp:5]: (error) Memory leak: f
```

Chapter 8. Exception safety

Cppcheck has a few checks that ensure that you don't break the basic guarantee of exception safety. It doesn't have any checks for the strong guarantee yet.

Example:

```
Fred::Fred() : a(new int[20]), b(new int[20])
{
}
```

By default Cppcheck will not detect any problems in that code.

To enable the exception safety checking you can use *--enable*:

```
cppcheck --enable=exceptNew --enable=exceptRealloc fred.cpp
```

The output will be:

```
[fred.cpp:3]: (style) Upon exception there is memory leak: a
```

If an exception occurs when `b` is allocated, `a` will leak.

Here is another example:

```
int *p;

int a(int sz)
{
    delete [] p;
    if (sz <= 0)
        throw std::runtime_error("size <= 0");
    p = new int[sz];
}
```

Check that with Cppcheck:

```
cppcheck --enable=exceptNew --enable=exceptRealloc except2.cpp
```

The output from Cppcheck is:

```
[except2.cpp:7]: (error) Throwing exception in invalid state, p points at deallocated memor
```

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

It isn't recommended to provide the paths to the standard C/C++ headers - Cppcheck has internal knowledge about ANSI C/C++ and it isn't recommended that this known functionality is redefined. But feel free to try it.

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.