

Cppcheck 1.69

Cppcheck 1.69

Table of Contents

| | |
|-----------------------------------------------------|-----------|
| 1. Introduction..... | 1 |
| 2. Getting started | 2 |
| 2.1. First test..... | 2 |
| 2.2. Checking all files in a folder | 2 |
| 2.3. Excluding a file or folder from checking | 2 |
| 2.4. Severities | 3 |
| 2.5. Enable messages | 3 |
| 2.5.1. Inconclusive checks..... | 4 |
| 2.6. Saving results in file | 4 |
| 2.7. Multithreaded checking..... | 4 |
| 3. Preprocessor configurations..... | 6 |
| 4. XML output..... | 7 |
| 4.1. The <error> element..... | 7 |
| 4.2. The <location> element | 8 |
| 5. Reformatting the output..... | 9 |
| 6. Suppressions | 11 |
| 6.1. Suppressing a certain error type..... | 11 |
| 6.1.1. Command line suppression..... | 11 |
| 6.1.2. Listing suppressions in a file | 11 |
| 6.2. Inline suppressions | 12 |
| 7. Library configuration..... | 13 |
| 7.1. Using your own custom .cfg file | 13 |
| 7.2. Memory/resource leaks | 13 |
| 7.2.1. alloc and dealloc | 13 |
| 7.2.2. leak-ignore and use..... | 14 |
| 7.3. Function argument: Uninitialized memory | 15 |
| 7.4. Function Argument: Null pointers | 16 |
| 7.5. Function Argument: Format string..... | 16 |
| 7.6. Function Argument: Value range | 17 |
| 7.7. Function Argument: minsize..... | 18 |
| 7.8. noreturn | 19 |
| 7.9. use-retval | 20 |
| 7.10. define | 21 |
| 7.11. podtype | 21 |
| 7.12. container..... | 22 |
| 7.13. Example configuration for strcpy() | 22 |
| 7.14. Specifications for all arguments | 23 |
| 8. Rules..... | 24 |
| 8.1. <tokenlist> | 24 |
| 8.2. <pattern> | 25 |
| 8.3. <id> | 25 |
| 8.4. <severity>..... | 25 |
| 8.5. <summary> | 25 |

| | |
|------------------------------------------------|-----------|
| 9. Cppcheck extensions with Python..... | 26 |
| 9.1. Simple checker: Division by zero | 26 |
| 9.2. Licensing..... | 26 |
| 10. HTML report | 27 |
| 11. Graphical user interface..... | 28 |
| 11.1. Introduction..... | 28 |
| 11.2. Check source code | 28 |
| 11.3. Inspecting results..... | 28 |
| 11.4. Settings..... | 28 |
| 11.5. Project files..... | 28 |

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="1.66">
    <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. Library configuration

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

If you create a configuration file for a popular library, we would appreciate if you upload it to us.

7.1. Using your own custom .cfg file

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

The command line cppcheck will try to load custom .cfg files from the working path - execute cppcheck from the path where the .cfg files are.

The cppcheck GUI will try to load custom .cfg files from the project file path. The custom .cfg files should be shown in the `Edit Project File` dialog that you open from the `File` menu.

7.2. Memory/resource leaks

Cppcheck has configurable checking for leaks.

7.2.1. alloc and dealloc

Here is an example program:

```
void test()
{
    HPEN pen = CreatePen(PS_SOLID, 1, RGB(255,0,0));
}
```

The code example above has a resource leak - `CreatePen()` is a windows function that creates a pen. However Cppcheck doesn't assume that return values from functions must be freed. There is no error message:

```
# cppcheck pen1.c
Checking pen1.c...
```

If you provide a windows configuration file then Cppcheck detects the bug:

```
# cppcheck --library=windows.cfg pen1.c
Checking pen1.c...
[pen1.c:3]: (error) Resource leak: pen
```

Here is a minimal windows.cfg file:

```
<?xml version="1.0"?>
<def>
  <resource>
    <alloc>CreatePen</alloc>
    <dealloc>DeleteObject</dealloc>
  </resource>
</def>
```

7.2.2. leak-ignore and use

Often the allocated pointer is passed to functions. Example:

```
void test()
{
    char *p = malloc(100);
    dostuff(p);
}
```

If Cppcheck doesn't know what `dostuff` does, without configuration it will assume that `dostuff` takes care of the memory so there is no memory leak.

To specify that `dostuff` doesn't take care of the memory in any way, use `leak-ignore`:

```
<?xml version="1.0"?>
<def>
  <function name="dostuff">
    <leak-ignore/>
    <arg nr="1"/>
    <arg nr="2"/>
  </function>
</def>
```

If instead `dostuff` takes care of the memory then this can be configured with:

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

```

<def>
  <memory>
    <alloc>malloc</alloc>
    <dealloc>free</dealloc>
    <use>dostuff</use>
  </memory>
</def>

```

The `<use>` configuration has no logical purpose. You will get the same warnings without it. Use it to silence `--check-library` information messages.

7.3. Function argument: Uninitialized memory

Here is an example program:

```

void test()
{
    char buffer1[1024];
    char buffer2[1024];
    CopyMemory(buffer1, buffer2, 1024);
}

```

The bug here is that `buffer2` is uninitialized. The second argument for `CopyMemory` needs to be initialized. However `Cppcheck` assumes that it is fine to pass uninitialized variables to functions:

```

# cppcheck unittest.c
Checking unittest.c...

```

If you provide a windows configuration file then `Cppcheck` detects the bug:

```

# cppcheck --library=windows.cfg unittest.c
Checking unittest.c...
[unittest.c:5]: (error) Uninitialized variable: buffer2

```

Here is the minimal `windows.cfg`:

```

<?xml version="1.0"?>
<def>
  <function name="CopyMemory">
    <arg nr="1"/>
    <arg nr="2">
      <not-uninit/>
    </arg>
    <arg nr="3"/>
  </function>

```

```
</def>
```

7.4. Function Argument: Null pointers

Cppcheck assumes it's ok to pass NULL pointers to functions. Here is an example program:

```
void test()
{
    CopyMemory(NULL, NULL, 1024);
}
```

The MSDN documentation is not clear if that is ok or not. But let's assume it's bad. Cppcheck assumes that it's ok to pass NULL to functions so no error is reported:

```
# cppcheck null.c
Checking null.c...
```

If you provide a windows configuration file then Cppcheck detects the bug:

```
cppcheck --library=windows.cfg null.c
Checking null.c...
[null.c:3]: (error) Null pointer dereference
```

Here is a minimal windows.cfg file:

```
<?xml version="1.0"?>
<def>
  <function name="CopyMemory">
    <arg nr="1">
      <not-null/>
    </arg>
    <arg nr="2"/>
    <arg nr="3"/>
  </function>
</def>
```

7.5. Function Argument: Format string

You can define that a function takes a format string. Example:

```
void test()
{
    do_something("%i %i\n", 1024);
}
```

```
}
```

No error is reported for that:

```
# cppcheck formatstring.c
Checking formatstring.c...
```

A configuration file can be created that says that the string is a format string. For instance:

```
<?xml version="1.0"?>
<def>
  <function name="do_something">
    <formatstr type="printf"/>
    <arg nr="1">
      <formatstr/>
    </arg>
  </function>
</def>
```

Now Cppcheck will report an error:

```
cppcheck --library=test.cfg formatstring.c
Checking formatstring.c...
[formatstring.c:3]: (error) do_something format string requires 2 parameters but only 1 is
```

The type attribute can be either:

- printf - format string follows the printf rules
- scanf - format string follows the scanf rules

7.6. Function Argument: Value range

The valid values can be defined. Imagine:

```
void test()
{
    do_something(1024);
}
```

No error is reported for that:

```
# cppcheck valuerange.c
Checking valuerange.c...
```

A configuration file can be created that says that 1024 is out of bounds. For instance:

```
<?xml version="1.0"?>
<def>
  <function name="do_something">
    <arg nr="1">
      <valid>0:1023</valid>
    </arg>
  </function>
</def>
```

Now Cppcheck will report an error:

```
cppcheck --library=test.cfg range.c
Checking range.c...
[range.c:3]: (error) Invalid do_something() argument nr 1. The value is 1024 but the valid
```

Some example expressions you can use in the valid element:

```
0,3,5 => only values 0, 3 and 5 are valid
-10:20 => all values between -10 and 20 are valid
:0 => all values that are less or equal to 0 are valid
0: => all values that are greater or equal to 0 are valid
0,2:32 => the value 0 and all values between 2 and 32 are valid
```

7.7. Function Argument: minsize

Some function arguments take a buffer. With minsize you can configure the min size of the buffer (in bytes, not elements). Imagine:

```
void test()
{
    char str[5];
    do_something(str, "12345");
}
```

No error is reported for that:

```
# cppcheck minsize.c
Checking minsize.c...
```

A configuration file can for instance be created that says that the size of the buffer in argument 1 must be larger than the strlen of argument 2. For instance:

```
<?xml version="1.0"?>
<def>
  <function name="do_something">
    <arg nr="1">
```

```

    <minsize type="strlen" arg="2"/>
  </arg>
  <arg nr="2"/>
</function>
</def>

```

Now Cppcheck will report this error:

```

cppcheck --library=1.cfg minsize.c
Checking minsize.c...
[minsize.c:4]: (error) Buffer is accessed out of bounds: str

```

There are different types of minsizes:

strlen

buffer size must be larger than other arguments string length. Example: see strcpy configuration in std.cfg

argvalue

buffer size must be larger than value in other argument. Example: see memset configuration in std.cfg

sizeof

buffer size must be larger than other argument buffer size. Example: see strncpy configuration in std.cfg

mul

buffer size must be larger than multiplication result when multiplying values given in two other arguments. Typically one argument defines the element size and another element defines the number of elements. Example: see fread configuration in std.cfg

7.8. noreturn

Cppcheck doesn't assume that functions always return. Here is an example code:

```

void test(int x)
{
    int data, buffer[1024];
    if (x == 1)
        data = 123;
    else
        ZeroMemory(buffer, sizeof(buffer));
    buffer[0] = data; // <- error: data is uninitialized if x is not 1
}

```

In theory, if `ZeroMemory` terminates the program then there is no bug. Cppcheck therefore reports no error:

```
# cppcheck noreturn.c
Checking noreturn.c...
```

However if you use `--check-library` and `--enable=information` you'll get this:

```
# cppcheck --check-library --enable=information noreturn.c
Checking noreturn.c...
[noreturn.c:7]: (information) --check-library: Function ZeroMemory() should have <noreturn>
```

If a proper `windows.cfg` is provided, the bug is detected:

```
# cppcheck --library=windows.cfg noreturn.c
Checking noreturn.c...
[noreturn.c:8]: (error) Uninitialized variable: data
```

Here is a minimal `windows.cfg` file:

```
<?xml version="1.0"?>
<def>
  <function name="ZeroMemory">
    <noreturn>false</noreturn>
  </function>
</def>
```

7.9. use-retval

As long as nothing else is specified, cppcheck assumes that ignoring the return value of a function is ok:

```
bool test(const char* a, const char* b)
{
    strcmp(a, b); // <- bug: The call of strcmp does not have side-effects, but the return
    return true;
}
```

In case `strcmp` has side effects, such as assigning the result to one of the parameters passed to it, nothing bad would happen:

```
# cppcheck useretval.c
Checking useretval.c...
```

If a proper `lib.cfg` is provided, the bug is detected:

```
# cppcheck --library=lib.cfg --enable=warning useretval.c
```



```
Checking userretval.c...
[noreturn.c:3]: (warning) Return value of function strcmp() is not used.
```

Here is a minimal `lib.cfg` file:

```
<?xml version="1.0"?>
<def>
  <function name="strcmp">
    <use-retval/>
  </function>
</def>
```

7.10. define

Libraries can be used to define preprocessor macros as well. For example:

```
<?xml version="1.0"?>
<def>
  <define name="NULL_VALUE" value="0"/>
</def>
```

Each occurrence of `"NULL_VALUE"` in the code would then be replaced by `"0"` at preprocessor stage.

7.11. podtype

Lots of code relies on typedefs providing platform independent types. "podtype"-tags can be used to provide necessary information to cppcheck to support them. Without further information, cppcheck does not understand the type `"uint16_t"` in the following example:

```
void test() {
    uint16_t a;
}
```

No message about variable `'a'` being unused is printed:

```
# cppcheck --enable=style unusedvar.cpp
Checking unusedvar.cpp...
```

If `uint16_t` is defined in a library as follows, the result improves:

```
<?xml version="1.0"?>
<def>
  <podtype name="uint16_t" sign="u" size="2"/>
</def>
```

The size of the type is specified in bytes. Possible values for the "sign" attribute are "s" (signed) and "u" (unsigned). Both attributes are optional. Using this library, cppcheck prints:

```
# cppcheck --library=lib.cfg --enable=style unusedvar.cpp
Checking unusedvar.cpp...
[unusedvar.cpp:2]: (style) Unused variable: a
```

7.12. container

A lot of C++ libraries, among those the STL itself, provide containers with very similar functionality. Libraries can be used to tell cppcheck about their behaviour. Each container needs a unique ID. It can optionally have a startPattern, which must be a valid Token::Match pattern and an endPattern that is compared to the linked token of the first token with such a link. The optional attribute "inherits" takes an ID from a previously defined container.

Inside the <container> tag, functions can be defined inside of the tags <size>, <access> and <other> (on your choice). Each of them can specify an action like "resize" and/or the result it yields, for example "end-iterator".

The following example provides a definition for std::vector, based on the definition of "stdContainer" (not shown):

```
<?xml version="1.0"?>
<def>
  <container id="stdVector" startPattern="std :: vector &lt;" inherits="stdContainer">
    <size>
      <function name="push_back" action="push"/>
      <function name="pop_back" action="pop"/>
    </size>
    <access indexOperator="array-like">
      <function name="at" yields="at_index"/>
      <function name="front" yields="item"/>
      <function name="back" yields="item"/>
    </access>
  </container>
</def>
```

7.13. Example configuration for strcpy()

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/>
  <strz/>
</arg>
</function>

```

The `<leak-ignore/>` 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>` tells Cppcheck if this function returns or not.

The first argument that the function takes is a pointer. It must not be a null pointer, therefore `<not-null>` is used.

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. Moreover it must point at a zero-terminated string so `<strz>` is also used.

7.14. Specifications for all arguments

Specifying `-1` as the argument number is going to apply a check to all arguments of that function. The specifications for individual arguments override this setting.

Chapter 8. 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>
```

CDATA can be used to include characters in a pattern that might interfere with XML:

```
<![CDATA[some<strange>pattern]]>
```

8.1. <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.

8.2. `<pattern>`

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

8.3. `<id>`

The ID specify the user-defined message id.

8.4. `<severity>`

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

8.5. `<summary>`

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

Chapter 9. Cppcheck extensions with Python

Using dump files it is possible to write Cppcheck extensions with for instance Python.

The `cppcheckdata.py` module (<http://github.com/danmar/cppcheck/blob/master/tools/cppcheckdata.py>) allows you to load such dump file. It contains `Token/Variable/ValueFlow.Value/Scope` classes that are similar to the C++ classes in Cppcheck-core. The doxygen information for the C++ classes should be somewhat useful for Python developers also.

9.1. Simple checker: Division by zero

Here is a simple checker:

```
import cppcheckdata

data = cppcheckdata.parsedump('1.c.dump')

for token in data.tokenlist:
    if token.str == '/' or token.str == '%':
        # Get denominator (2nd operand)
        den = token.astOperand2

        # Can denominator be zero?
        if den.getValue(0):
            print '[' + token.file + ':' + str(token.linenr) + '] Division by zero'
```

Example usage:

```
cppcheck --dump 1.c
python divzero.py
```

9.2. Licensing

The dump file is just a xml file, so it is an open interface without restrictions. You can use it in any way you need.

The `cppcheckdata.py` is also free to use. No matter if your project is open source or closed source. Use it for any purpose.

Chapter 10. 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 11. Graphical user interface

11.1. Introduction

A Cppcheck GUI is available.

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

11.2. Check source code

Use the **Check** menu.

11.3. 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`.

11.4. Settings

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

More settings are available in `Edit`→`Preferences`.

11.5. 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.