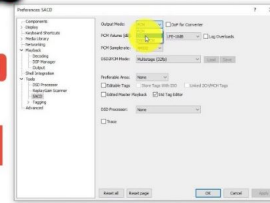


**foobar 2000**

**DSD Setup**

**Tutorial**



## Cppcheck 2000 Playing back DSD audio using foobar User Manual

[Home](#) » [Cppcheck](#) » Cppcheck 2000 Playing back DSD audio using foobar User Manual 

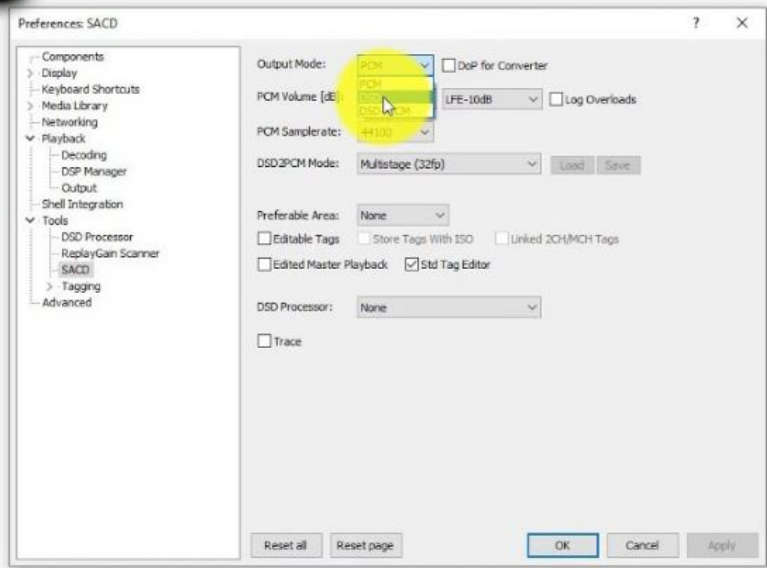
### Contents

- [1 Cppcheck 2000 Playing back DSD audio using foobar](#)
- [2 Introduction](#)
- [3 About static analysis](#)
- [4 Command line](#)
- [5 Clang parser \(experimental\)](#)
- [6 Cppcheck build folder](#)
- [7 Preprocessor Settings](#)
- [8 Symbol name](#)
- [9 Reformatting the text output](#)
- [10 Library configuration](#)
- [11 Documents / Resources](#)
  - [11.1 References](#)
- [12 Related Posts](#)

# Cppcheck

# foobar 2000

## DSD Setup Tutorial



### 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

## 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

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 pathIf “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 `cpp check src/a src/b`

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: 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 template 'a<101>'. You might want to limit Cppcheck recursion. [templateRecursion]a<i+1>(); ^`

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

- add template specialization
- configure Cppcheck, which can be done in the GUI project file dialog example code with template specialization:
- `template <int i>void a()aa<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`.
- 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 the 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 compiled 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). 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.

### Include paths

To add an include path, use `-I`, followed by the path. Cppcheck's preprocessor basically handles includes like any other preprocessor. However, while other preprocessors stop working when they encounter a missing header, Cppcheck will just print an information message and continues parsing the code. The purpose of this behaviour is that Cppcheck is meant to work without necessarily seeing the entire code. Actually, it is recommended to not give all include paths. While it is useful for Cppcheck to see the declaration of a class when checking the implementation of its members, passing standard library headers is discouraged, because the analysis will not work fully and lead to a longer checking time. For such cases, `.cfg` files are the preferred way to provide information about the implementation of functions and types to Cppcheck, see below for more information.

### Platform

You should use a platform configuration that matches your target environment. By default Cppcheck uses native platform configuration that works well if your code is compiled and executed locally. Cppcheck has built-in 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.

### 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 `cp-pcheck` build dir. For instance, the unused function 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 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: `Univar // suppress all uninitvar errors in all files// suppress memleak and exceptNew errors in the file src/file1.cpp`

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

### 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: Using 1 suppression comment before the code: Suppression comment on the same line as the code:

### Symbol name

You can specify that the inline suppression only applies to a specific symbol:// cppcheck-suppress aaaa  
symbolName=arr// 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

### **The <error> element**

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

#### **id**

id of error, and which are valid symbol names

#### **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` To get gcc compatible output you can use `–template=gcc: cppcheck –template=gcc samples/arrayIndexOutOfBounds/bad.c`

## 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: The output will then look like this: `samples/arrayIndexOutOfBounds/bad.c:6: error: Array 'a[2]' accessed at index 2, which is out of bounds.` A comma-separated format: `cppcheck –template="{file},{line},{severity},{id},{message}" samples/arrayIndexOutOfBounds/bad.c`

## User-defined output format (multi-line)

Many warnings have multiple locations. Example code: 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: 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.

## 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.

### 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 the Cppcheck home page. This will launch all Cppcheck checks and additionally calls specific checks provided by selected addon. Some add-ons need extra arguments. You can configure how you want to execute an addon in a json file. For example put this in `misra.json`: And then the configuration can be executed on the Cppcheck command line: 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.

## 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 `HTML report` that contains a script that transforms a Cppcheck XML file into HTML output.

This command generates the help screen: `cppcheck gui/test.cpp –xml 2> err.xml htmlreport/cppcheck-htmlreport –file=err.xml –report-dir=test1V –source-dir=.`

## Documents / Resources

	<a href="#">Cppcheck 2000 Playing back DSD audio using foobar</a> [pdf] User Manual 2000 Playing back DSD audio using foobar, Playing back DSD audio using foobar, DSD audio u sing foobar, using foobar
---	--

References

-  [Welcome! — Pygments](#)
-  [Secure Development | Software Engineering Institute](#)
-  [Cppcheck - MISRA C 2012 Compliance](#)
-  [Year 2038 problem - Wikipedia](#)
-  [GitHub - 3adev/y2038: A sandbox for Y2038 development on GLIBC](#)
-  [cppcheck/y2038.txt at main · danmar/cppcheck · GitHub](#)
-  [cppcheck/misra.py at main · danmar/cppcheck · GitHub](#)
-  [cppcheck/addons/test/misra at main · danmar/cppcheck · GitHub](#)
-  [cppcheck/threadsafety.py at main · danmar/cppcheck · GitHub](#)
-  [cppcheck/y2038.py at main · danmar/cppcheck · GitHub](#)
-  [MISRA](#)