# Introduction

## Types of Analysis

There’re two types of analysis:

* **Static analysis**: The process of detecting software defects via automated software tools WITHOUT executing the code.
* **Dynamic analysis**: The process of detecting software defects via automated software tools BY executing the code.

In this tutorial, we’ll discover some automated software tools for static and dynamic analysis.

## Important Points to Consider

**Work-in-Progress**

The analyzer is a continuous work-in-progress. There are many planned enhancements to improve both the precision and scope of its analysis algorithms, as well as the kinds of bugs it will find. While there are fundamental limitations to what static analysis can do, we have a long way to go before hitting that wall.

**Slower than Compilation**

Operationally, using static analysis to automatically find deep program bugs is about trading CPU time for the hardening of code. Static analysis can be much slower than compilation. Some of the algorithms needed to find bugs require in the worst case exponential time.

**False Positives**

Static analysis is not perfect. It can falsely flag bugs in a program where the code behaves correctly. Because some code checks require more analysis precision than others, the frequency of false positives can vary widely between different checks. The long-term goal is to have the analyzer have a low false positive rate for most code on all checks.

**More Checks**

Static analysis is not magic; a static analyzer can only find bugs that it has been specifically engineered to find.

# CppCheck

CppCheck is a free cross-platform **static analysis tool** for C/C++ code with a great community.

CppCheck guides:

[https://CppCheck.sourceforge.io/manual.html](https://cppcheck.sourceforge.io/manual.html)

[https://manpages.ubuntu.com/manpages/jammy/man1/CppCheck.1.html](https://manpages.ubuntu.com/manpages/jammy/man1/cppcheck.1.html)

You might not know

CppCheck is available both as **open-source** and as **CppCheck Premium** with extended functionality and support.

## Bug Types

Common **types of bugs** CppCheck can detect: uninitialized/unused variables and functions, out of bounds, exception safety, memory leaks, obsolete functions, invalid usage of STL, invalid conversions, null pointer dereferences, etc.

For a full checklist, see [https://sourceforge.net/p/CppCheck/wiki/ListOfChecks/](https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/)

## Coding Standards

CppCheck is designed to be able to analyze C/C++ code even if it has **non-standard syntax** (common in embedded projects).

|  |  |  |
| --- | --- | --- |
| **Coding standard** | **CppCheck plan** | |
| [**Free**](http://cppcheck.net/) | [**Premium**](https://www.cppcheck.com/) |
| Misra C 2012 - original rules | Yes | Yes |
| Misra C 2012 - amendment #1 | Yes | Yes |
| Misra C 2012 - amendment #2 | Yes | Yes |
| Misra C 2012 - amendment #3 |  | Yes |
| Misra C 2012 - amendment #4 |  | Yes |
| Misra C 2012 - Compliance report |  | Yes |
| Misra C 2012 - Rule texts | User provided | Yes |
| Misra C 2023 |  | Yes |
| Misra C++ 2008 |  | Yes |
| Misra C++ 2023 |  | Yes |
| Cert C |  | Yes |
| Cert C++ |  | Yes |
| Autosar |  | [Partial](https://files.cppchecksolutions.com/autosar.html) |

## Clients and Plugins

CppCheck is integrated with many popular development tools. For example:

* Buildbot - [integrated](https://docs.buildbot.net/latest/manual/configuration/steps/cppcheck.html)
* CLion - [CppCheck plugin](https://plugins.jetbrains.com/plugin/8143)
* Code::Blocks - integrated
* CodeDX (software assurance tool) - [integrated](http://codedx.com/code-dx-standard/)
* CodeLite - integrated
* CppDepend 5 - [integrated](http://www.cppdepend.com/CppDependV5.aspx)
* Eclipse - [Cppcheclipse](https://github.com/kwin/cppcheclipse/wiki/Installation)
* gedit - [gedit plugin](http://github.com/odamite/gedit-cppcheck)
* github - [Codacy](https://www.codacy.com/) and [SoftaCheck](http://www.softacheck.com/)
* Hudson - [CppCheck Plugin](http://wiki.hudson-ci.org/display/HUDSON/Cppcheck+Plugin)
* Jenkins - [CppCheck Plugin](http://wiki.jenkins-ci.org/display/JENKINS/Cppcheck+Plugin)
* KDevelop - [integrated since v5.1](https://kdevelop.org/)
* Mercurial (Linux) - [pre-commit hook](http://sourceforge.net/p/cppcheck/wiki/mercurialhook/) - Check for new errors on commit (requires interactive terminal)
* QtCreator - [Qt Project Tool (qpt)](https://sourceforge.net/projects/qtprojecttool/files)
* Tortoise SVN - [Adding a pre-commit hook script](http://omerez.com/automatic-static-code-analysis/)
* Visual Studio - [Visual Studio plugin](https://github.com/VioletGiraffe/cppcheck-vs-addin/releases/latest)
* Vim - [Vim Compiler](https://vimhelp.org/quickfix.txt.html#compiler-cppcheck)
* Visual Studio - [Visual Studio plugin](https://github.com/VioletGiraffe/cppcheck-vs-addin/releases/latest)
* VScode - [VScode plugin](https://marketplace.visualstudio.com/items?itemName=NathanJ.cppcheck-plugin)

Ref: [http://CppCheck.net/](http://cppcheck.net/)

## Usage

### Check one file

In command line, run:

$ cppcheck <path-to-file> # You can input multiple files, each must be separated with a space

Suppose you have file1.c with following content:

int main() {

    char a[10];

    a[10] = 0;

    return 0;

}

In command line, run:

$ cppcheck file1.c

Output:

Checking file1.c...

[file1.c:4]: (error) Array 'a[10]' index 10 out of bounds

### Check all files in a folder

In command line, run:

$ cppcheck <path-to-folder> # You can input multiple folders, each must be separated with a space

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 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.

### Exclude files from checking

To exclude a file or folder, use -i to specifies the files/paths to ignore.

For example, to ignores the src/a folder, run:

$ cppcheck -isrc/a src # You can input multiple folders, each must be separated with a space

This option is only valid when supplying an input directory which is src in this case.

### Choose severities

To switch severities (or additional checks), use --enable=<severity> option.

The available severity IDs are: all, warning, style, performance, portability, information, unusedFunction, missingInclude.

**Notes**:

* By default none of the severities are enabled.
* Several IDs can be given if you separate them with **commas**, e.g. --enable=style,unusedFunction.

**Details**:

* **All**: Enable all below severities.
* **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 behaviors**.
* **Style**: **Stylistic issues**, like unused functions, redundant code, constness, operator precedence, possible mistakes.

All messages with the severities 'style', 'performance' and 'portability' are enabled.

* **Performance**: Run-time performance suggestions based on common knowledge, though it’s 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.
* **Unused Functions**: Check for unused functions. It's **recommend to only enable this when the whole program is scanned**
* **Missing Include**: Warn if there are missing includes. For detailed information use --check-config.

### Use Clang parser (experimental)

By default, CppCheck uses an internal C/C++ parser. However, there is an experimental option to use the Clang parser instead.

Of course, you need to 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.

### Import project files

You can import project files which contain build configurations (C/C++ standard, included header files, preprocessors, compiliation flags, etc.) into CppCheck using the --project= option. With them, CppCheck can deliver a more precise (how much???) checking result.

Depending on the build system of your project, project files will be different. Below are some common build systems:

#### CMake

Generate a compile database (compile\_commands.json) in the current folder:

$ cmake -DCMAKE\_EXPORT\_COMPILE\_COMMANDS=ON .

Now run CppCheck like this:

$ cppcheck --project=compile\_commands.json

To ignore analysis of source files in certain folders, use -i:

$ cppcheck --project=compile\_commands.json -ifoo

#### Visual Studio

Run CppCheck on an entire solution:

$ cppcheck --project=foobar.sln

Running CppCheck on a project:

$ cppcheck --project=foobar.vcxproj

You can limit on a single configuration:

$ cppcheck --project=foobar.sln "--project-configuration=Release|Win32"

To ignore analysis of source files in certain folders, use -i:

$ cppcheck --project=foobar.vcxproj -ifoo

#### C++ Builder 6

Run CppCheck on a project:

$ cppcheck --project=foobar.bpr

To ignore analysis of source files in certain folders, use -i:

$ cppcheck --project=foobar.bpr -ifoo

#### Others

If you can generate a compile database yourself, then it's possible to import that in CppCheck.

In Linux, you can use the bear (build ear) utility to generate a compile database from arbitrary build tools:

$ bear make

### Import configuration file

CppCheck has its own configuration file in .CppCheck extension. You can **create it using CppCheck GUI** (*File* -> *New project file*) (for more details, see the [CppCheck GUI session](#_Cppcheck_GUI)). You can also import it like importing project files:

$ cppcheck --project=foobar.CppCheck

The best benefit of a config file is eliminate writing lengthy commands again and again because you can put all of your necessary settings for CppCheck in a single file.

**Notes**:

* The config file doesn’t contain all options. So, you still have to pass some options (like STD standard, forced language, etc.) to the command line.
* Options in the **config file will overwrite** options passed to command line.
* The exact syntax of the config file is undocumented, but it's basically an XML file. For example:

<?xml version="1.0" encoding="UTF-8"?>

<project version="1">

    <root name="."/>

    <builddir>cppcheck-build-dir</builddir>

    <platform>native</platform>

    <analyze-all-vs-configs>false</analyze-all-vs-configs>

    <check-headers>true</check-headers>

    <check-unused-templates>false</check-unused-templates>

    <max-ctu-depth>10</max-ctu-depth>

    <includedir>

        <dir name="service/"/>

    </includedir>

    <defines>

        <define name="DEBUG"/>

    </defines>

    <undefines>

        <undefine>RELEASE</undefine>

    </undefines>

    <paths>

        <dir name="service"/>

    </paths>

    <exclude>

        <path name="test/"/>

    </exclude>

    <libraries>

        <library>posix</library>

        <library>tinyxml2</library>

    </libraries>

    <suppressions>

        <suppression>AssignmentAddressToInteger</suppression>

        <suppression fileName="abc.cpp" lineNumber="12" symbolName="abc">AssignmentIntegerToAddress</suppression>

    </suppressions>

    <addons>

        <addon>threadsafety</addon>

        <addon>cert</addon>

    </addons>

</project>

### Configure libraries

When external libraries are used (WinAPI, POSIX, gtk, Qt, etc.), CppCheck doesn’t know how they behave. It then fails to detect various problems like memory leaks, buffer overflows, possible null pointer dereferences, etc.

This can be fixed with configuration files!

CppCheck already contains configurations for several libraries. Each config file is a .cfg file. You can create your own cfg file for your project. Use --check-library to get hints about what you should configure.

Note: Configuration file for the standard libraries of C (c11) and C++ (c++20) is std.cfg which is always loaded by CppCheck. It’s documented [here](https://cppcheck.sourceforge.io/reference-cfg-format.pdf).

### Configure preprocessors

If you import project files (with --project=), then CppCheck will automatically use the preprocessor settings in the imported project file. So, you don’t have to do anything extra.

Otherwise, a bit of manual preprocessor configuration is required:

* The flag -D tells CppCheck that a name is defined. There will be no analysis without this define.
* The flag -U tells CppCheck that a name is undefined. There will be no 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.

For example, here is a code that has 3 bugs:

#ifdef A

x=100/0; // bug 1

#ifdef B

y=100/0; // bug 2

#endif

#else

z=100/0; // bug 3

#endif

#ifndef C

#error C must be defined

Copy it from Cppcheck manual, but cannot understand

#endif

Running CppCheck in different ways gives different results:

$ cppcheck test.c # Test all configs => all bugs are found

$ cppcheck -DA test.c # Only test config A => No bug is found (#error)

$ cppcheck -DA -DC test.c # Only test config A and C => The first bug is found

$ cppcheck -UA test.c # Only test config C => The last bug is found

$ cppcheck --force -DA test.c # Test all config with "-DA" => The two first bugs are found

### Suppress certain checks

If you want to filter out certain errors/warnings from being generated, then it is possible to suppress these. For more details, check CppCheck manual guide.

The format for an warning suppression is one of:

[error id] # Suppress cerrain error/warning

[error id]:[filename] # Suprress certain error/warning in certain file

[error id]:[filename]:[line] # Suprress certain error/warning in certain file at certain line

There are some ways to suppress checks:

#### Command Line

The --suppress= option is used to specify suppressions on the command line. Example:

# Suppress memleak warning in the file src/file1.cpp

$ cppcheck --suppress=memleak:src/file1.cpp src/

**Tips**:

* You can get list of checks using $ CppCheck --doc. The output will look like [this](https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/).
* You can get list of check IDs and messages using $ CppCheck --errorlist. For a even more detailed explanation for each check, see [this](https://github.com/wcventure/Static-Analysis-Rules/tree/master/Summary%20of%20static%20analysis%20in%20C%20%26%20C%2B%2B).

#### In-File

You can create a suppressions file and then pass its path to command line. This file can be either in TXT format or XML format.

**XML Format**

For example:

// Suppress memleak and exceptNew warnings in the file src/file1.cpp

memleak:src/file1.cpp

exceptNew:src/file1.cpp

// Suppress all uninitvar errors in all files

uninitvar

// You can 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.

Now, pass it to the command with the --suppressions-list= option:

$ cppcheck --suppressions-list=suppressions.txt src/

**XML Format**

For example:

<?xml version="1.0"?>

<suppressions>

  <suppress>

    <id>memleak</id>

    <fileName>src/file1.c</fileName>

  </suppress>

  <suppress>

    <id>exceptNew</id>

    <fileName>src/file1.c</fileName>

  </suppress>

  <suppress>

    <id>uninitvar</id>

  </suppress>

</suppressions>

Now, pass it to the command with the --suppress-xml= option:

$ cppcheck --suppress-xml=suppressions.xml src/

**Tips**:

* You can put a <suppressions> session into your XML [configuration file](#_Import_configuration_file) (.CppCheck).

#### In-Line

Suppressions can also be added directly in the code by adding comments that contain special keywords.

For more details, check CppCheck manual guide.

To activate inline suppressions, add option --inline-suppr:

$ CppCheck --inline-suppr test.c

### Choose language

By default, CppCheck **automatically detects language to run analysis**. But if you want, you can **force** CppCheck to check code in the given language with --language=<language> option. Valid values are: c, c++

### Choose standard

To specify C/C++ standard, use --std=<id> option. The available options are:

* posix: POSIX compatible code
* 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)

For example:

$ cppcheck --std=c99 --std=posix file.cpp

## Tips

### CppCheck Build Folder

Using a CppCheck build folder is not mandatory, but it is **recommended**.

When CppCheck saves analyzer information in that folder, there are some benefits:

* It speeds up the analysis as it makes incremental analysis possible. **Only changed files are analyzed when rechecking**.
* Whole program analysis also when multiple threads are used.

To use the build folder, use --CppCheck-build-dir=<path>. Example:

$ mkdir build

$ cppcheck --CppCheck-build-dir=build src # All files are analyzed

$ cppcheck --CppCheck-build-dir=build src # Now only changed files are analyzed

### CppCheck GUI

CppCheck has a GUI, not just command line.

In fact, **CppCheck GUI has a few options that are not available in the command line directly**. The command line tool usage is kept intentionally simple; therefore, the options are limited.

If you want to have these limited options in the command line, **a workaroud way is to first create a CppCheck GUI project with them activated, then import the GUI project file on the command line** by running:

$ cppcheck --project=foobar.CppCheck

### Check Level

* **Normal**: The "normal" check level is chosen by default. It provides an effective checking in "reasonable" time.
* **Exhaustive**: When you can wait longer for the results, you can enable the "exhaustive" checking, by using the option --check-level=exhaustive.

### Addons

Addons are Python **scripts** that analyse **CppCheck dump files** to check compatibility with secure **coding standards**.

#### List of Addons

CppCheck is distributed with a few addons:

**misra.py**

The [misra.py](https://github.com/danmar/cppcheck/blob/main/addons/misra.py) is used to verify compliance with **MISRA C 2012**.

Because this standard is proprietary, 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 by referring to <https://www.misra.org.uk>.

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](https://cppcheck.sourceforge.io/misra.php) home page.

**y2038.py**

The [y2038.py](https://github.com/danmar/cppcheck/blob/main/addons/y2038.py) checks Linux systems for [year 2038 problem](https://en.wikipedia.org/wiki/Year_2038_problem) safety. This required [modified environment](https://github.com/3adev/y2038). See complete description [here](https://github.com/danmar/cppcheck/blob/main/addons/doc/y2038.txt).

**threadsafety.py**

The [threadsafety.py](https://github.com/danmar/cppcheck/blob/main/addons/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 with --addon option:

$ 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"

]

}

Then the configuration can be executed by:

$ 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.

### XML Output

By using --xml option, CppCheck can generate output in XML format:

For example, output errors in XML:

$ 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>

For more details, check CppCheck manual guide.

### HTML Report

You can convert the XML output from CppCheck into a HTML report. You’ll need Python and the [pygments](http://pygments.org) module 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

Example usage:

$ ./cppcheck gui/test.cpp --xml 2> err.xml

$ htmlreport/CppCheck-htmlreport --file=err.xml --report-dir=test1 --source-dir=.

### Online Demo

You don’t even need to install CppCheck to use it. An online demo is available here: [https://CppCheck.sourceforge.io/demo/](https://cppcheck.sourceforge.io/demo/)

### VS Code Extension

#### [cpp-check-lint](https://marketplace.visualstudio.com/items?itemName=QiuMingGe.cpp-check-lint)

**Sample of VSCode's settings.json**:

///////////// General ////////////

"cpp-check-lint.--enable": true,                    // Enable or disable the cpp-check-lint

"cpp-check-lint.--log": 0,                          // Log level to output (0: OFF, 1: Error, 2: Warning, 3: Info, 4: Debug)

///////////// CppCheck ////////////

"cpp-check-lint.cppcheck.--enable": true,           // Enable or disable CppCheck

"cpp-check-lint.cppcheck.--executable": "CppCheck", // Path to CppCheck executable

"cpp-check-lint.cppcheck.--onsave": true,           // Whether CppCheck is run on save

"cpp-check-lint.cppcheck.--customargs=": "--project=/home/worker/vcm/nad/LGE/tiger-src/services/v2xmgr-service/config.CppCheck",    // Additional arguments to pass to CppCheck command. 1> Path to .CppCheck config file

"cpp-check-lint.cppcheck.--enable=": "all",         // Choose severities to check. The available are: all, warning, style, performance, portability, information, unusedFunction, missingInclude

"cpp-check-lint.cppcheck.--platform=": "native",

"cpp-check-lint.cppcheck.--std\_c++=": "c++17",

"cpp-check-lint.cppcheck.--std\_c=": "c11",

"cpp-check-lint.cppcheck.--suppress=": [],          // We don't put suppressions here, but put them all in .CppCheck config file

# Cpplint

Cpplint is a open-source command-line tool to check C/C++ files for style issues following [Google's C++ style guide](http://google.github.io/styleguide/cppguide.html). It’s developed and maintained by Google Inc. at [google/styleguide](https://github.com/google/styleguide).

Cpplint guide:

<https://github.com/cpplint/cpplint>

https://manpages.ubuntu.com/manpages/jammy/man1/cpplint.1.html

## Bug Types

Following is the full checklist:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| build/class | readability/alt\_tokens | runtime/arrays | whitespace/blank\_line | legal/copyright |
| build/c++11 | readability/braces | runtime/casting | whitespace/braces |  |
| build/c++14 | readability/casting | runtime/explicit | whitespace/comma |  |
| build/c++tr1 | readability/check | runtime/int | whitespace/comments |  |
| build/deprecated | readability/constructors | runtime/init | whitespace/empty\_conditional\_body |  |
| build/endif\_comment | readability/fn\_size | runtime/invalid\_increment | whitespace/empty\_if\_body |  |
| build/explicit\_make\_pair | readability/inheritance | runtime/member\_string\_references | whitespace/empty\_loop\_body |  |
| build/forward\_decl | readability/multiline\_comment | runtime/memset | whitespace/end\_of\_line |  |
| build/header\_guard | readability/multiline\_string | runtime/indentation\_namespace | whitespace/ending\_newline |  |
| build/include | readability/namespace | runtime/operator | whitespace/forcolon |  |
| build/include\_subdir | readability/nolint | runtime/printf | whitespace/indent |  |
| build/include\_alpha | readability/nul | runtime/printf\_format | whitespace/line\_length |  |
| build/include\_order | readability/strings | runtime/references | whitespace/newline |  |
| build/include\_what\_you\_use | readability/todo | runtime/string | whitespace/operators |  |
| build/namespaces\_headers | readability/utf8 | runtime/threadsafe\_fn | whitespace/parens |  |
| build/namespaces\_literals |  | runtime/vlog | whitespace/semicolon |  |
| build/namespaces |  |  | whitespace/tab |  |
| build/printf\_format |  |  | whitespace/todo |  |
| build/storage\_class |  |  |  |  |

## Coding Standards

## Clients and Plugins

## Usage

### Check one file

In the command line, run:

cpplint --clint <path-to-file>

### Check all files in a folder

cpplint --clint <path-to-folder>/\*

### Exclude files from checking

Exclude the given path from the list of files to be checked. For example:

--exclude=one.cc --exclude=test/\*.cc --exclude=mock/\*.cc

### Choose severities

### Import configuration file

If you try of writing lengthy commands again and again, you can create a config file called CPPLINT.cfg. Cpplint will try to find it in the closest parent directory of the source file.

Following is a sample of CPPLINT.cfg file:

# Don't search for additional CPPLINT.cfg in parent directories.

set noparent

# Use 'ART\_' as the cpp header guard prefix (e.g. #ifndef ART\_PATH\_TO\_FILE\_H\_).

root=..

# Limit line length.

linelength=100

# Ignore the following categories of errors, as specified by the filter:

# (the filter settings are concatenated together)

filter=-build/c++11

filter=-build/include

filter=-readability/function,-readability/streams,-readability/todo

filter=-runtime/printf,-runtime/references,-runtime/sizeof,-runtime/threadsafe\_fn

filter=-whitespace/line\_length

**Notes**:

* Options in CPPLINT.cfg file will overwrite options passed to command line.

### Suppress certain checks

#### Command Line

To tell Cpplint which errors/warnings will be printed, you can use the --filter=+<error-type> option. To tell Cpplint which errors will be suppressed, you can use the --filter=-<error-type> option. Each error type is spearated by a comma.

For example, if you want to suppress "whitespace" error, but keep "whitespace/braces" error, you can use:

--filter=-whitespace,+whitespace/braces

**Tips:**

* Error/warning types are printed with the message and at the end of the message between "[]". For example, "[whitespace/indent]".
* To see a list of all the checks, run: $ cpplint --filter=

Another way is to read the source code [cpplint.py](https://github.com/cpplint/cpplint/blob/develop/cpplint.py) (search for \_ERROR\_CATEGORIES variable).

#### In-File

You can put filters to your [CPPLINT.cfg file](#_Import_configuration_file_1).

### VS Code Extension

#### [cpp-check-lint](https://marketplace.visualstudio.com/items?itemName=QiuMingGe.cpp-check-lint)

**Sample of VSCode's settings.json**:

///////////// General ////////////

"cpp-check-lint.--enable": true,                    // Enable or disable the cpp-check-lint

"cpp-check-lint.--log": 0,                          // Log level to output (0: OFF, 1: Error, 2: Warning, 3: Info, 4: Debug)

///////////// CppLint ////////////

"cpp-check-lint.cpplint.--enable": true,            // Enable or disable the cpplint

"cpp-check-lint.cpplint.--executable": "cpplint",   // Path to Cpplint executable

"cpp-check-lint.cpplint.--onsave": true,            // Whether Cpplint is run on save

"cpp-check-lint.cpplint.--filter=": "",             // We don't put suppressions here, but put them all in CPPLINT.cfg config file

"cpp-check-lint.cpplint.--linelength=": 120,        // Allowed line length for the project

"cpp-check-lint.cpplint.--output=": "eclipse",      // Can be one of following: emacs, eclipse, vs7, junit, sed, gsed

"cpp-check-lint.cpplint.--verbose=": 1,             // Specify a number 0-5 to restrict errors to certain verbosity levels.

"cpp-check-lint.cpplint.--customargs=": "",         // Addition arguments to pass to cpplint command

# Clang-Tidy

Check *Clang/Clang.docx.*

# PC-lint and Pc-lint Plus

PC-lint and PC-link Plus are commerccial software linting tools produced by Gimpel Software (formerly Gimp Suit Software Ltd.) for C/C++.

They come with a comprehensive ruCle set covering widely recognized coding standards such as MISRA, CERT-C and AUTOSAR.

<https://pclintplus.com/>

# Flawfinder

<https://dwheeler.com/flawfinder/>

# Valgrind

## Introduction

Valgrind is an **open-source dynamic analysis framework** that helps developers detect and diagnose various programming errors.

Here are some important notes you need to know about Valgrind before using it:

* **Only for Unix-like OS**: That means Valgrind is designed for Linux, macOS, and other Unix-like OS. It’s not natively supported on Windows.
* **Only for C and C++**: Valgrind cannot work with other languages except C and C++.
* **Compile with debugging symbols**: To get more accurate and informative results from Valgrind, it’s recommended to compile your code with debugging symbols enabled. This can be done by including the -g flag when compiling with GCC or Clang.
* **Start with small test cases**: It’s helpful to start with smaller test cases to reduce the amount of output. Once you fix the issues, you can gradually increase the complexity of the test cases.
* **Learn to use Valgrind**: It’s not an easy tool to master, so check the [Valgrind’s documentations](https://valgrind.org/docs/).
* **Consider using suppression files**: Valgrind allows the use of suppression files to filter out known or irrelevant errors. Suppression files can help reduce noise and focus on the relevant issues during analysis.
* **Use Valgrind tools selectively**: Valgrind provides several tools, each designed to detect specific types of errors. Here are some of the commonly used:
  + **Memcheck**: It’s the default. It detects **memory-related errors** including memory errors and memory leaks.

Note: When you run the valgrind command without specifying a specific tool, it defaults to running the Memcheck tool. You have to explicitly specify a different tool if you want to focus on a specific type of analysis. For example, you can run valgrind --tool=cachegrind ./myprogram to use the Cachegrind tool.

* + **Cachegrind**: It’s a tool for **analyzing cache performance**. It simulates the behavior of the CPU cache hierarchy and provides information on cache misses, cache line utilization, and cache-related performance bottlenecks.
  + **Helgrind**: It’s a tool for **detecting synchronization errors in multithreaded programs**. It can identify issues like data races (concurrent access to shared data without proper synchronization), deadlocks, and thread misuses.
  + **DRD (DynamoRIO-based Detector)**: It’s a tool for **detecting data races in multithreaded programs**. It uses dynamic binary instrumentation to analyze memory accesses and identify potential data races. DRD can be used as an alternative to Helgrind for data race detection.
  + **Massif**: It’s a tool for **measuring heap memory usage** over time. It provides insights into memory allocation patterns, peak memory usage, and memory growth. Massif can help identify memory leaks, inefficient memory usage, and areas of high memory consumption.
  + **Callgrind**: It’s a tool for **collecting information about function calls** and their associated costs, including CPU time and cache misses. It generates a detailed profile that can be visualized using tools like KCachegrind or QCacheGrind. It’s useful for understanding program performance and identifying hotspots that may benefit from optimization.

## Memcheck

Valgrind will print a summary of its memory usage. If all goes well, it'll look something like this:

==4649== ERROR SUMMARY: 0 errors from 0 contexts

==4649== malloc/free: in use at exit: 0 bytes in 0 blocks.

==4649== malloc/free: 10 allocs, 10 frees, 2640 bytes allocated.

==4649== For counts of detected errors, rerun with: -v

==4649== All heap blocks were freed -- no leaks are possible.

### Detect Memory Error

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #include <stdio.h>  #include <stdlib.h>  #include <string.h>  int main()  {      char buffer[] = "Hello, world!";      // Allocate memory for copy      char\* copy = malloc(strlen(buffer));      // Copy the contents of buffer to copy      strcpy(copy, buffer);      printf("Copied string: %s\n", copy);      free(copy);      return 0;  } |

Analyzing the above program with Valgrind gives following output:

==4651== Invalid write of size 1

==4651==    at 0x80486A4: main (myprogram.c:13)

==4651==  Address 0x4449054 is not stack'd, malloc'd or (recently) free'd

==4651==

==4651== ERROR SUMMARY: 1 errors from 1 contexts

==4651== malloc/free: in use at exit: 0 bytes in 0 blocks.

==4651== malloc/free: 1 allocs, 1 frees, 10 bytes allocated.

==4651== For counts of detected errors, rerun with: -v

==4651== All heap blocks were freed -- no leaks are possible.

Here is the fix:

char\* copy = malloc(strlen(buffer) + 1);

Why memory error:

The string literal Hello, world! has 13 characters. That’s mean its string length is 13 (characters), but it actually occupies 14 (bytes) (including the \0 terminator). To make thing correct, we need to allocate 14 bytes in memory with malloc().

There are different kinds of memory errors that you may see in Valgrind reports. The most common are:

* Invalid read/write of size X: The program was observed to read/write X bytes of memory that was invalid. Common causes: accessing beyond the end of a heap block, accessing memory has been freed, or accessing into an unallocated region like from use of a uninitialized pointer.
* Use of uninitialised value or Conditional jump or move depends on uninitialised value(s): The program read the value of a memory location that was not previously written to, i.e. uses random junk. The second more specifically indicates the read occurred in the test expression in an if/for/while. Make sure to initialize all of your variables!

Note that Valgrind will silently allow a program to propagate an uninitialized value along from variable to variable; the complaint will only come when it eventually uses the value which may be far removed from the root of the error. When tracking down an uninitialized value, run Valgrind with the additional flag --track-origins=yes and it will report the entire history of the value back to the origin which can be very helpful.

* Source and destination overlap in memcpy(): The program attempted to copy data from one location to another and the range to be read intersects with the range to be written. Transferring data between overlapping regions using memcpy can be a problem; memmove is the correct function to use in such a situation.
* Invalid free(): The program attempted to free a non-heap address or free the same block more than once.

### Detect Memory Leaks

To check for leaks, you need to include the options leak-check=full and --show-leak-kinds=all in the Valgrind command.

Analyzing a program with Valgrind gives following output:

==5942== ERROR SUMMARY: 0 errors from 0 contexts

==5942== malloc/free: in use at exit: 12 bytes in 1 blocks.

==5942== malloc/free: 250 allocs, 249 frees, 12476 bytes allocated.

==5942== For counts of detected errors, rerun with: -v

==5942== searching for pointers to 1 not-freed blocks.

==5942== checked 51,452 bytes.

==5942==

==5942==

==5942== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1

==5942==    at 0x43BC3C0: malloc (vg\_replace\_malloc.c:149)

==5942==    by 0x804863D: main (myprogram.c:51)

==5942==

==5942== LEAK SUMMARY:

==5942==    definitely lost: 12 bytes in 1 blocks.

==5942==      possibly lost: 0 bytes in 0 blocks.

==5942==    still reachable: 0 bytes in 0 blocks.

==5942==         suppressed: 0 bytes in 0 blocks.

It's pretty easy to tell when there's a leak:

* The alloc/free counts don't match up.
* You get a LEAK SUMMARY section at the end.

Valgrind also gives a little data about each leak -- how many bytes, how many times it happened, and where in the code the original allocation was made.

Valgrind categorizes leaks using these terms:

* definitely lost: heap-allocated memory that was never freed to which the program no longer has a pointer. Valgrind knows that you once had the pointer, but have since lost track of it. This memory is definitely orphaned.
* indirectly lost: heap-allocated memory that was never freed to which the only pointers to it also are lost. For example, if you orphan a linked list, the first node would be definitely lost, the subsequent nodes would be indirectly lost.
* possibly lost: heap-allocated memory that was never freed to which valgrind cannot be sure whether there is a pointer or not.
* still reachable: heap-allocated memory that was never freed to which the program still has a pointer at exit (typically this means a global variable points to it).

### QAs

**Valgrind says I leaked memory because of a call to malloc() in main(), but I don't call malloc() in main()! What's going on?**

This report can also be a result of calling a **library function** in main() that itself calls malloc() internally.

**My valgrind report suggests to rerun with -v for "counts of suppressed errors". What are these? Should I worry about them?**

Pay no attention. Some library code does unusual things which can trigger reports from Valgrind even when operating correctly. Those errors/leaks are suppressed as they are known to be spurious. The -v flag causes valgrind to provide verbose commentary about its internal handling of these events. You can safely ignore all suppressed events; no need for you to wade through the verbose chatter.

# Visual Studio IDE

The Visual Studio IDE provides various built-in tools, including static and dynamic code analyzers for C/C++, both in the editor and from the command line.

Here are some of the key tools in Visual Studio:

* **AddressSanitizer (ASan)**: ASan was added to Visual Studio 2019 version 16.7 and later versions. It’s a dynamic memory error detection tool that detects memory issues, like buffer overflows, use-after-free bugs, double-free, etc.

You can enable ASan in your project settings and use it to catch memory errors during runtime. For more info, check its [official guide](https://learn.microsoft.com/en-us/cpp/sanitizers/asan?view=msvc-170).

* **Static Code Analysis**: VS includes a static code analysis feature that can detect potential memory errors, such as buffer overflows, null pointer dereferences, and resource leaks. It performs a thorough analysis of your code without actually running it.

For more info, check its [official guide](https://learn.microsoft.com/en-us/visualstudio/code-quality/?view=vs-2022).

* **Memory Usage Analysis**: It allows you to profile your application's memory usage during runtime. It provides insights into memory allocations, deallocations, and tracks memory leaks. You can use it to identify memory-related issues and optimize memory usage in your code.

For more info, check its [official guide](https://learn.microsoft.com/en-us/visualstudio/profiling/memory-usage?view=vs-2022).

# Other Tools

## Free

### Infer

Infer is a free **static analysis tool** for C/C++, Objective-C and Java. It’s developed by Facebook.

The tool targets null pointer, memory leaks, coding conventions and unavailable API's problems.

<https://fbinfer.com/>

## Commercial

### CppDepend

<https://www.cppdepend.com/>

### Synopsys Coverity Static Analysis

<https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html>

### Parasoft C/C++ Static Analysis

<https://www.parasoft.com/products/parasoft-c-ctest/c-c-static-analysis/>