gcovr Documentation

Release 5.1

the gcovr authors

Mar 26, 2022

Contents:

Gcovr provides a utility for managing the use of the GNU gcov utility and generating summarized code coverage results. This command is inspired by the Python coverage.py package, which provides a similar utility for Python.

The `gcovr` command can produce different kinds of coverage reports:

CLI Option	User Guide	Description
default, `--txt`	*Text Output*	compact human-readable summaries
`--html`	*HTML Output*	overview of all files
`--html-details`	*HTML Output*	annotated source files
`--cobertura`	*Cobertura XML Output*	machine readable XML reports in Cobertura format
`--sonarqube`	*Sonarqube XML Output*	machine readable XML reports in Sonarqube format
`--json`	*JSON Output*	JSON report with source file structure and coverage
`--json-summary`	*JSON Output*	JSON summary coverage report
`--csv`	*CSV Output*	CSV report summarizing the coverage of each file
`--coveralls`	*Coveralls JSON Output*	machine readable JSON report in Coveralls format

Thus, gcovr can be viewed as a command-line alternative to the lcov utility, which runs gcov and generates an HTML-formatted report. The development of gcovr was motivated by the need for text summaries and XML reports.

Quick Links

- Getting Help
  - Submit a ticket
  - Stack Overflow
  - Chat on Gitter
- Install from PyPI: `pip install gcovr`
- Source Code on GitHub
- *Change Log*

This documentation (https://gcovr.com/) describes gcovr 5.1.

CHAPTER 1

Installation

Gcovr is available as a Python package that can be installed via pip.

Install newest stable `gcovr` release from PyPI:

```
pip install gcovr
```

Install development version from GitHub:

```
pip install git+https://github.com/gcovr/gcovr.git
```

Which environments does `gcovr` support?

Python: 3.7+.

The automated tests run on CPython (versions 3.7, 3.8, 3.9, 3.10) and a compatible PyPy3. Gcovr will only run on Python versions with upstream support.

Last gcovr release for old Python versions:

Python	gcovr
2.6	3.4
2.7	4.2
3.4	4.1
3.5	4.2
3.6	5.0

Operating System: Linux, Windows, and macOS.

The automated tests run on Ubuntu 18.04 and 20.04 and Windows Server 2019.

Compiler: GCC and Clang.

The automated tests run on GCC 5, 6, and 8.

CHAPTER 2

Getting Started

The `gcovr` command provides a summary of the lines that have been executed in a program. Code coverage statistics help you discover untested parts of a program, which is particularly important when assessing code quality. Well-tested code is a characteristic of high quality code, and software developers often assess code coverage statistics when deciding if software is ready for a release.

GCC can instrument the executables to emit coverage data. You need to recompile your code with the following flags:

```
--coverage -g -O0
```

Next, run your test suite. This will generate raw coverage files.

Finally, invoke gcovr. This will print a tabular report on the console.

```
gcovr
```

You can also generate detailed HTML reports:

```
gcovr --html-details coverage.html
```

Gcovr will create one HTML report per source file next to the coverage.html summary.

You should run gcovr from the build directory. The `-r` option should point to the root of your project. This only matters if you have a separate build directory. For example:

```
cd build; gcovr -r ..
```

2.1 What to read next

The *User Guide* explains how to use the features of gcovr. In particular:

- *Compiling for Coverage*
- *Output Formats*

- *Using Filters*

The *Command Line Reference* provides an overview of all options.

Specific problems might be addressed in the *Cookbook* or the *Frequently Asked Questions*.

CHAPTER 3

User Guide

The user guide describes the various features of gcovr. It assumes you have read the *Getting Started* guide.

This User Guide provides the following sections:

3.1 Compiling for Coverage

In order to collect coverage data, your software must be "instrumented" by the compiler. That means, you must re-compile your software with special compiler options.

The general workflow is:

1. compile your software to enable coverage profiling
2. execute your software to collect coverage profiles
3. run `gcovr` to create reports from the collected coverage profiling data

This document explains how you can use GCC or Clang to compile with coverage instrumentation.

If you cannot compile your software with coverage flags, you cannot use gcovr. However, other tools like kcov might help.

3.1.1 Example Code

The following `example.cpp` program is used to illustrate the compilation process:

```
1 // example.cpp
2
3 int foo(int param)
4 {
5     if (param)
6     {
7         return 1;
8     }
```

```
 9     else
10     {
11         return 0;
12     }
13 }
14
15 int main(int argc, char* argv[])
16 {
17     foo(0);
18
19     return 0;
20 }
```

This code executes several subroutines in this program, but some lines in the program are not executed.

3.1.2 Compiler Options

We compile `example.cpp` with the GCC compiler as follows:

```
g++ -fprofile-arcs -ftest-coverage -fPIC -O0 example.cpp -o program
```

What do these compiler flags mean?

- We compile without optimization (`-O0`), because optimizations may merge lines of code or otherwise change the flow of execution in the program. This can change the measured coverage.

  On the other hand, enabling basic optimizations with `-O1` can sometimes produce "better" coverage reports, especially for C++. This is a matter of personal preference, just make sure to avoid comparing coverage metrics across optimization levels.

  If you are having problems with lots of uncovered branches, see: *Why does C++ code have so many uncovered branches?*

- Either `--coverage` or `-fprofile-arcs -ftest-coverage` are needed so that the compiler produces the information necessary to gather coverage data.

  With these options, the compiler adds logic to the output program that counts how often which part of the code was executed. The compiler will also create a `example.gcno` file with metadata. The name of the gcno file matches the compilation unit (see below).

Optional compiler flags:

- You can use other flags like `-g` or `-fPIC` as required by your tests. These don't affect the coverage results.
- Using `-fprofile-abs-path` (available since GCC 8) can avoid some problems with interpreting the coverage data correctly. By default, the additional coverage files generated by GCC contain relative paths from the working directory to the source files. If there are multiple potential working directories from which you might have run the compiler, gcovr can get confused. Adding this option is more robust.

This examples uses the `g++` compiler for C++ code, but any GCC or Clang-based compiler should work.

If you are using CMake, see *Out-of-Source Builds with CMake* for information on configuring that build system to compile your software with coverage enabled.

3.1.3 Running the Program

The above compiler invocation generated a `program` executable. Now, we have to execute this command:

```
./program
```

This will run whatever you designed this program to do. Often, such a program would contain unit tests to exercise your code.

As a side effect, this will create an `example.gcda` file with the coverage data for our compilation unit. This is a binary file so it needs to be processed first. Together, the `.gcda` and `.gcno` files can be used to create coverage reports.

3.1.4 Processing Coverage

Your compiler ships with tools to analyze the coverage data files. For GCC, this is `gcov`. For Clang, this is `llvm-cov`. You don't have to call these programs yourself – gcovr will do that for you.

So let's invoke gcovr:

```
gcovr
```

This will search for all your `.gcno` and `.gcda` files, run the compiler's gcov tool, and summarize the code coverage statistics into a report. By default, we get a text summary on the command line that shows aggregate statistics for each line:

```
------------------------------------------------------------------------------
                           GCC Code Coverage Report
Directory: .
------------------------------------------------------------------------------
File                                       Lines    Exec  Cover   Missing
------------------------------------------------------------------------------
example.cpp                                    7       6    85%   7
------------------------------------------------------------------------------
TOTAL                                          7       6    85%
------------------------------------------------------------------------------
```

Gcovr supports many different *Output Formats* that you can generate instead.

3.1.5 Choosing the Right Gcov Executable

If you have multiple compilers installed or if you are using Clang, you will likely need to tell gcovr which gcov executable to use. By default, gcovr just uses the program named `gcov`. This is fine for the default GCC compiler, e.g. `gcc` or `g++`. Otherwise, you must use the *`--gcov-executable`* to tell gcovr what to use.

If you have used a specific GCC version (e.g. `gcc-8` or `g++-8`), then you must name the gcov tool with the corresponding version. For example:

```
gcovr --gcov-executable gcov-8
```

If you have used Clang, then you can use its gcov emulation mode. For example:

```
gcovr --gcov-executable "llvm-cov gcov"
```

Again, the `llvm-cov` name may have to include your compiler version.

3.1.6 Working with Multiple Object Files

Code coverage instrumentation works on a per object file basis, which means you have to re-compile your entire project to collect coverage data.

The C/C++ model has a concept of "compilation units". A large project is typically not compiled in one go, but in separate steps. The result of compiling a compilation unit is a `.o` object file with the machine code. The object code from multiple compilation units is later linked into the final executable or library. The previous example only had a single compilation unit, so no explicit linking step was necessary.

Because each compilation unit is compiled independently, every one has to be instrumented with coverage counters separately. A common mistake is to add the compiler flags for coverage (e.g. in the CFLAGS or CXXFLAGS variables) but then forgetting to force a re-compile. Depending on the build system, it may be necessary to clear out the old object files that weren't compiled with coverage, e.g. with a `make clean` command. Other build systems use a separate build directory when compiling with coverage so that incremental compilation works as expected.

Each object file will have an associated `.gcno` and `.gcda` file in the same directory as the `.o` object file. For example, consider the following compilation process:

```
# (1) compile to object code
g++ --coverage -c -o a.o a.cpp
g++ --coverage -c -o b.o b.cpp

# (2) link the object files in the program
g++ --coverage -o the-program a.o b.o

# (3) run the program
./the-program
```

1. Compiling the object code creates the `a.o` and `b.o` object files, but also corresponding `a.gcno` and `b.gcno` notes files, one for each compilation unit. The `-c` option is used to only compile but to not link the code.
2. Linking the object code produces the final program. This has no effect on coverage processing, except that the `--coverage` flag makes sure that a compiler-internal gcov support library is linked.
3. Running the program will increment the in-memory coverage counters for all executed lines. At the end, the counters are written into gcov data files, one for each compilation unit. Here, we would get `a.gcda` and `b.gcda` files.

If you only want coverage data for certain source files, it is sufficient to only compile those compilation units with coverage enabled that contain these source files. But this can be tricky to do correctly. For example, header files are often part of multiple compilation units.

3.2 Output Formats

Gcovr supports a variety of output formats that are documented on the following pages.

3.2.1 Text Output

The text output format summarizes coverage in a plain-text table. This is the default output format if no other format is selected. This output format can also be explicitly selected with the *`gcovr --txt`* option.

New in version 5.0: Added explicit *`--txt`* option.

Example output:

```
------------------------------------------------------------------------------
                           GCC Code Coverage Report
Directory: .
------------------------------------------------------------------------------
File                                       Lines    Exec  Cover   Missing
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------------------------
example.cpp                                 7       6    85%   7
------------------------------------------------------------------------------
TOTAL                                       7       6    85%
------------------------------------------------------------------------------
```

Line Coverage

Running gcovr without any explicit output formats ...

```
gcovr
```

generates a text summary of the lines executed:

```
------------------------------------------------------------------------------
                           GCC Code Coverage Report
Directory: .
------------------------------------------------------------------------------
File                                       Lines    Exec  Cover   Missing
------------------------------------------------------------------------------
example.cpp                                 7       6    85%   7
------------------------------------------------------------------------------
TOTAL                                       7       6    85%
------------------------------------------------------------------------------
```

The same result can be achieved when explicit *--txt* option is set. For example:

```
gcovr --txt
```

generates the same text summary.

Each line of this output includes a summary for a given source file, including the number of lines instrumented, the number of lines executed, the percentage of lines executed, and a summary of the line numbers that were not executed. To improve clarity, gcovr uses an aggressive approach to grouping uncovered lines and will combine uncovered lines separated by "non-code" lines (blank, freestanding braces, and single-line comments) into a single region. As a result, the number of lines listed in the "Missing" list may be greater than the difference of the "Lines" and "Exec" columns.

Note that `gcov` accumulates statistics by line. Consequently, it works best with a programming style that places only one statement on each line.

Branch Coverage

The `gcovr` command can also summarize branch coverage using the *-b/--branches* option:

```
gcovr --branches
```

This generates a tabular output that summarizes the number of branches, the number of branches taken and the branches that were not completely covered:

```
------------------------------------------------------------------------------
                           GCC Code Coverage Report
Directory: .
------------------------------------------------------------------------------
File                                    Branches   Taken  Cover   Missing
------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
example.cpp                                                2        1     50%   5
------------------------------------------------------------------------------
TOTAL                                                      2        1     50%
------------------------------------------------------------------------------
```

The same result can be achieved when explicit `--txt` option is set. For example:

```
gcovr --branches --txt
```

prints the same tabular output.

3.2.2 HTML Output

The `gcovr` command can also generate a simple HTML output using the `--html` option:

```
gcovr --html
```

This generates a HTML summary of the lines executed. In this example, the file `example1.html` is generated, which has the following output:

GCC Code Coverage Report

Directory: ./
Date: 2022-03-25 22:40:35
Legend: low: >= 0% medium: >= 75.0% high: >= 90.0%

	Exec	Total	Coverage
Lines:	29	43	67.4%
Functions:	7	10	70.0%
Branches:	6	14	42.9%

List of functions

File	Lines			Functions		Branches	
A/C/D/file6.cpp		75.0%	3 / 4	100.0%	1 / 1	50.0%	1 / 2
A/C/file5.cpp		75.0%	3 / 4	100.0%	1 / 1	50.0%	1 / 2
A/file1.cpp		75.0%	3 / 4	100.0%	1 / 1	50.0%	1 / 2
A/file2.cpp		57.1%	4 / 7	50.0%	1 / 2	-%	0 / 0
A/file3.cpp		44.4%	4 / 9	50.0%	1 / 2	0.0%	0 / 2
A/file4.cpp		75.0%	3 / 4	100.0%	1 / 1	50.0%	1 / 2
A/file7.cpp		0.0%	0 / 2	0.0%	0 / 1	-%	0 / 0
B/main.cpp		100.0%	9 / 9	100.0%	1 / 1	50.0%	2 / 4

Generated by: GCOVR (Version 5.1)

The default behavior of the `--html` option is to generate HTML for a single webpage that summarizes the coverage for all files. The HTML is printed to standard output, but the `-o/--output` option is used to specify a file that stores the HTML output.

The `--html-details` option is used to create a separate web page for each file. Each of these web pages includes the contents of file with annotations that summarize code coverage. Consider the following command:

```
gcovr --html-details example_html.details.html
```

This generates the following HTML page for the file `example1.cpp`:

GCC Code Coverage Report

		Exec	Total	Coverage
Directory:	./			
File:	A/file4.cpp			
Date:	2022-03-25 22:40:35			
	Lines:	3	4	75.0%
	Functions:	1	1	100.0%
	Branches:	1	2	50.0%

▸ List of functions

Line	Branch	Exec	Source
1		1	int foobar(int param)
2			{
3	▸ 1/2	1	if (param) {
4		1	return 1;
5			} else {
6		✗	return 0;
7			}
8			}
9			

Generated by: GCOVR (Version 5.1)

Note that the *--html-details* option needs a named output, e.g. via the the *-o/--output* option. For example, if the output is named `coverage.html`, then the web pages generated for each file will have names of the form `coverage.<filename>.html`.

The *--html-self-contained* option controls whether assets like CSS styles are bundled into the HTML file. The *--html* report defaults to self-contained mode. but *--html-details* defaults to *--no-html-self-contained* in order to avoid problems with the Content Security Policy of some servers, especially Jenkins.

New in version 5.0: Added *--html-self-contained* and *--no-html-self-contained*.

Changed in version 5.0: Default to external CSS file for *--html-details*.

3.2.3 Cobertura XML Output

The default output format for `gcovr` is to generate a tabular summary in plain text. The `gcovr` command can also generate a Cobertura XML output using the *--cobertura* and *--cobertura-pretty* options:

```
gcovr --cobertura-pretty
```

This generates an XML summary of the lines executed:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE coverage SYSTEM 'http://cobertura.sourceforge.net/xml/coverage-04.dtd'>
<coverage line-rate="0.8571428571428571" branch-rate="0.5" lines-covered="6" lines-
↪valid="7" branches-covered="1" branches-valid="2" function-rate="1.0" functions-
↪covered="2" functions-valid="2" complexity="0.0" timestamp="1573053861" version=
↪"gcovr 5.1">
  <sources>
    <source>.</source>
  </sources>
  <packages>
    <package name="" line-rate="0.8571428571428571" branch-rate="0.5" function-rate=
↪"1.0" complexity="0.0">
      <classes>
        <class name="example_cpp" filename="example.cpp" line-rate="0.8571428571428571
↪" branch-rate="0.5" complexity="0.0">
```

(continues on next page)

(continued from previous page)

```
          <methods/>
          <lines>
            <line number="3" hits="1" branch="false"/>
            <line number="5" hits="1" branch="true" condition-coverage="50% (1/2)">
              <conditions>
                <condition number="0" type="jump" coverage="50%"/>
              </conditions>
            </line>
            <line number="7" hits="0" branch="false"/>
            <line number="11" hits="1" branch="false"/>
            <line number="15" hits="1" branch="false"/>
            <line number="17" hits="1" branch="false"/>
            <line number="19" hits="1" branch="false"/>
          </lines>
        </class>
      </classes>
    </package>
  </packages>
</coverage>
```

This XML format is in the Cobertura XML format suitable for import and display within the Jenkins and Hudson continuous integration servers using the Cobertura Plugin. Gcovr also supports a *Sonarqube XML Output*.

The `--cobertura` option generates a denser XML output, and the `--cobertura-pretty` option generates an indented XML output that is easier to read. Note that the XML output contains more information than the tabular summary. The tabular summary shows the percentage of covered lines, while the XML output includes branch statistics and the number of times that each line was covered. Consequently, XML output can be used to support performance optimization in the same manner that `gcov` does.

New in version 5.1: The `--cobertura` and `--cobertura-pretty` options were added as an alias for `-x`/`--xml` and `--xml-pretty`, respectively. This avoids confusion with other XML output formats like *Sonarqube XML Output*. The old options remain available for backwards compatibility.

3.2.4 Sonarqube XML Output

If you are using Sonarqube, you can get a coverage report in a suitable XML format via the `--sonarqube` option:

```
gcovr --sonarqube coverage.xml
```

The Sonarqube XML format is documented at https://docs.sonarqube.org/latest/analysis/generic-test/.

3.2.5 JSON Output

The `gcovr` command can also generate a JSON output using the `--json` and `--json-pretty` options:

```
gcovr --json coverage.json
```

The `--json-pretty` option generates an indented JSON output that is easier to read.

If you just need a summary of the coverage information, similar to the tabulated text based output, you can use `--json-summary` instead (see *JSON Summary Output*).

Multiple JSON files can be merged into the coverage data with sum of lines and branches execution, see *Merging Coverage Data*.

JSON Format Reference

Structure of file is based on gcov JSON intermediate format with additional key names specific to gcovr.

Structure of the JSON is following:

```
{
    "gcovr/format_version": gcovr_json_version
    "files": [file]
}
```

gcovr_json_version: version of gcovr JSON format. This is independently versioned from gcovr itself.

Each *file* has the following form:

```
{
    "file": file
    "lines": [line]
}
```

file: path to source code file, relative to gcovr root directory.

Each **line** has the following form:

```
{
    "branches": [branch]
    "count": count
    "line_number": line_number
    "gcovr/noncode": gcovr_noncode
}
```

gcovr_noncode: if True coverage info on this line should be ignored

Each **branch** has the following form:

```
{
  "count": count
  "fallthrough": fallthrough
  "throw": throw
}
```

file, *line* and *branch* have the structure defined in gcov intermediate format. This format is documented at https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html#Invoking-Gcov.

JSON Summary Output

The *--json-summary* option output coverage summary in a machine-readable format for additional post processing. The format is identical to JSON output *--json* option without detailed `lines` information. The *--json-summary-pretty* option generates an indented JSON summary output that is easier to read. Consider the following command:

```
gcovr --json-summary-pretty --json-summary
```

This generates an indented JSON summary:

```
{
    "branch_covered": 1,
    "branch_percent": 50.0,
```

(continues on next page)

(continued from previous page)


```
    "branch_total": 2,
    "files": [
        {
            "branch_covered": 1,
            "branch_percent": 50.0,
            "branch_total": 2,
            "filename": "example.cpp",
            "function_covered": 2,
            "function_percent": 100.0,
            "function_total": 2,
            "line_covered": 6,
            "line_percent": 85.7,
            "line_total": 7
        }
    ],
    "function_covered": 2,
    "function_percent": 100.0,
    "function_total": 2,
    "gcovr/summary_format_version": "0.5",
    "line_covered": 6,
    "line_percent": 85.7,
    "line_total": 7,
    "root": "."
}
```


New in version 5.0: Added *--json-summary* and *--json-summary-pretty*.

3.2.6 CSV Output

The *--csv* option output comma-separated values summarizing the coverage of each file. Consider the following command:

```
gcovr --csv
```

This generates an CSV:

```
filename,line_total,line_covered,line_percent,branch_total,branch_covered,branch_
↪percent,function_total,function_covered,function_percent
example.cpp,7,6,0.857,2,1,0.5,2,2,1.0
```

New in version 5.0: Added *--csv*.

3.2.7 Coveralls JSON Output

If you are using Coveralls, you can get a coverage report in a suitable JSON format via the *--coveralls* option:

```
gcovr --coveralls coverage.json
```

The *--coveralls-pretty* option generates an indented JSON output that is easier to read.

Keep in mind that the output contains the checksums of the source files. If you are using different OSes, the line endings shall be the same.

If available, environment variable COVERALLS_REPO_TOKEN will be consumed and baked into the JSON output.

If running in a CI additional variables are used:

- In Travis CI:
  - TRAVIS_JOB_ID
  - TRAVIS_BUILD_NUMBER
  - TRAVIS_PULL_REQUEST
  - TRAVIS_COMMIT
  - TRAVIS_BRANCH
- In Appveyor:
  - APPVEYOR_JOB_ID
  - APPVEYOR_JOB_NUMBER
  - APPVEYOR_PULL_REQUEST_NUMBER
  - APPVEYOR_REPO_COMMIT
  - APPVEYOR_REPO_BRANCH
- In Jenkins CI:
  - JOB_NAME
  - BUILD_ID
  - CHANGE_ID
  - GIT_COMMIT (if available)
  - BRANCH_NAME
- In GitHub Actions:
  - GITHUB_WORKFLOW
  - GITHUB_RUN_ID
  - GITHUB_SHA
  - GITHUB_HEAD_REF (if available)
  - GITHUB_REF

The Coveralls JSON format is documented at https://docs.coveralls.io/api-introduction.

New in version 5.0: Added `--coveralls` and `--coveralls-pretty`.

You can use *Multiple Output Formats* at the same time.

3.3 Multiple Output Formats

You can write multiple report formats with one gcovr invocation by passing the output filename directly to the report format flag. If no filename is specified for the format, the value from `-o/--output` is used by default, which itself defaults to stdout.

The following report format flags can take an optional output file name:

- `gcovr --csv`
- `gcovr --txt`
- `gcovr --cobertura`

- *gcovr --html*
- *gcovr --html-details*
- *gcovr --sonarqube*
- *gcovr --json*
- *gcovr --json-summary*
- *gcovr --coveralls*

If the value given to the output option ends with a path seperator (/ or \) it is used a directory which is created first and a default filename depending on the format is used.

Note that *--html-details* overrides any value of *--html* if it is present.

3.4 Merging Coverage Data

You can merge coverage data from multiple runs with *-a/--add-tracefile*.

For each run, generate *JSON output*:

```
...  # compile and run first test case
gcovr ... --json run-1.json
...  # compile and run second test case
gcovr ... --json run-2.json
```

Next, merge the json files and generate the desired report:

```
gcovr --add-tracefile run-1.json --add-tracefile run-2.json --html-details coverage.
↪html
```

You can also use unix style wildcards to merge the json files without duplicating *-a/--add-tracefile*. With this option you have to place your pathnames with wildcards in double quotation marks:

```
gcovr --add-tracefile "run-*.json" --html-details coverage.html
```

3.5 Using Filters

Gcovr tries to only report coverage for files within your project, not for your libraries. This is influenced by the following options:

- *-r*, *--root*
- *-f*, *--filter*
- *-e*, *--exclude*
- *--gcov-filter*
- *--gcov-exclude*
- *--exclude-directories*
- (the current working directory where gcovr is invoked)

These options take filters. A filter is a regular expression that matches a file path. Because filters are regexes, you will have to escape "special" characters with a backslash `\`.

Always use forward slashes `/` as path separators, even on Windows:

- wrong: `--filter C:\project\src\`
- correct: `--filter C:/project/src/`

If the filter looks like an absolute path, it is matched against an absolute path. Otherwise, the filter is matched against a relative path, where that path is relative to the current directory or if defined in a configuration file to the directory of the file.

Examples of relative filters:

- `--filter subdir/` matches only that subdirectory
- `--filter '\.\./src/'` matches a sibling directory `../src`. But because a dot `.` matches any character in a regex, we have to escape it. You have to use additional shell escaping. This example uses single quotes for Bash or POSIX shell.
- `--filter '(.+/)?foo\.c$'` matches only files called `foo.c`. The regex must match from the start of the relative path, so we ignore any leading directory parts with `(.+/)?`. The `$` at the end ensures that the path ends here.

If no *-f/--filter* is provided, the *-r/--root* is turned into a default filter. Therefore, files outside of the *-r/--root* directory are excluded.

To be included in a report, the source file must match any *-f/--filter*, and must not match any *-e/--exclude* filter.

The *--gcov-filter* and *--gcov-exclude* filters apply to the `.gcov` files created by `gcov`. This is useful mostly when running gcov yourself, and then invoking gcovr with *-g/--use-gcov-files*. But these filters also apply when gcov is launched by gcovr.

3.5.1 Speeding up coverage data search

The *--exclude-directories* filter is used while searching for raw coverage data (or for existing `.gcov` files when *-g/--use-gcov-files* is active). This filter is matched against directory paths, not file paths. If a directory matches, all its contents (files and subdirectories) will be excluded from the search. For example, consider this build directory:

```
build/
├─ main.o
├─ main.gcda
├─ main.gcno
├─ a/
│  ├─ awesome_code.o
│  ├─ awesome_code.gcda
│  └─ awesome_code.gcno
└─ b/
   ├─ better_code.o
   ├─ better_code.gcda
   └─ better_code.gcno
```

If we run `gcovr --exclude-directories 'build/a$'`, this will exclude anything in the `build/a` directory but will use the coverage data for `better_code.o` and `main.o`.

This can speed up gcovr when you have a complicated build directory structure. Consider also using the *search_paths* or *--object-directory* arguments to specify where gcovr starts searching. If you are unsure which directories are being searched, run gcovr in *-v/--verbose* mode.

For each found coverage data file gcovr will invoke the `gcov` tool. This is typically the slowest part, and other filters can only be applied *after* this step. In some cases, parallel execution with the *-j* option might be helpful to speed up processing.

3.5.2 Filters for symlinks

Gcovr matches filters against *real paths* that have all their symlinks resolved. E.g. consider this project layout:

```
/home/you/
├─ project/  (pwd)
│  ├─ src/
│  ├─ relevant-library/ -> ../external-library/
│  └─ ignore-this/
└─ external-library/
   └─ src/
```

Here, the `relevant-library` has the real path `/home/you/external-library`.

To write a filter that includes both `src/` and `relevant-library/src/`, we cannot use `--filter relevant-library/src/` because that contains a symlink. Instead, we have to use an absolute path to the real name:

```
gcovr --filter src/ --filter /home/you/external-library/src/
```

or a relative path to the real path:

```
gcovr --filter src/ --filter '\.\./external-library/src/'
```

New in version 5.1: gcovr also supports symlinks/junctions/drive substitutions on Windows.

3.6 Configuration Files

Warning: Config files are an experimental feature and may be subject to change without prior notice.

Defaults for the command line options can be set in a configuration file. Example:

```
filter = src/
html-details = yes  # info about each source file
output = build/coverage.html
```

How the configuration file is found: If a *--config* option is provided, that file is used. Otherwise, a `gcovr.cfg` file in the *-r/--root* directory is used, if that file exists.

Each line contains a `key = value` pair. Space around the = is optional. The value may be empty. Comments start with a hash # and ignore the rest of the line, but cannot start within a word. Empty lines are also ignored.

The available config keys correspond closely to the command line options, and are parsed similarly. In most cases, the name of a long command line option can be used as a config key. If not, this is documented in the option's help message. For example, *--gcov-executable* can be set via the `gcov-executable` config key. But *-b/--branches* is set via `txt-branch`.

Just like command line options, the config keys can be specified multiple times. Depending on the option the last one wins or a list will be built. For example, *-f/--filter* can be provided multiple times:

```
# Only show coverage for files in src/, lib/foo, or for main.cpp files.
filter = src/
filter = lib/foo/
filter = *./main\.cpp
```

Note that relative filters specified in config files will be interpreted relative to the location of the config file itself.

Option arguments are parsed with the following precedence:

- First the config file is parsed, if any.
- Then, all command line arguments are added.
- Finally, if an option was specified neither in a config file nor on the command line, its documented default value is used.

Therefore, it doesn't matter whether a value is provided in the config file or the command line.

Boolean flags are treated specially. When their config value is "yes" they are enabled, as if the flag had been provided on the command line. When their value is "no", they are explicitly disabled by assigning their default value. The *-j* flag is special as it takes an optional argument. In the config file, `gcov-parallel = yes` would refer to the no-argument form, whereas `gcov-parallel = 4` would provide an explicit argument.

Some config file syntax is explicitly reserved for future extensions: Semicolon comments, INI-style sections, multi-line values, quoted values, variable substitutions, alternative key–value separators, ...

3.7 Exclusion Markers

You can exclude parts of your code from coverage metrics.

- If `GCOVR_EXCL_LINE` appears within a line, that line is ignored.
- If `GCOVR_EXCL_START` appears within a line, all following lines (including the current line) are ignored until a `GCOVR_EXCL_STOP` marker is encountered.

Instead of `GCOVR_*`, the markers may also start with `GCOV_*` or `LCOV_*`. However, start and stop markers must use the same style. The prefix is configurable with the option *--exclude-pattern-prefix*.

In the excluded regions, *any* coverage is excluded. It is not currently possible to exclude only branch coverage in that region. In particular, lcov's EXCL_BR markers are not supported (see issue #121).

3.8 Reproducible Timestamps

In some cases, it may be desirable to list a specific timestamp in the report. Timestamps are shown in the *HTML Output*, *Coveralls JSON Output*, and the *Cobertura XML Output*. This can be achieved via the *--timestamp* option. This option does not affect the modification times or other filesystem metadata.

New in version 5.1: The *gcovr --timestamp* option.

3.8.1 Timestamp Syntax

The timestamp option understands different formats: Unix timestamps and RFC-3339 timestamps.

Unix timestamps (also known as Posix time or Epoch) are the number of seconds since 1 Jan 1970. These timestamps are always resolved in the UTC timezone. Example usage:

```
gcovr --timestamp 1640606727
```

RFC 3339 specifies a reasonable subset of ISO-8601 timestamps. This is the `YYYY-MM-DDThh:mm:ss` format, optionally followed by a timezone offset (`+hh:mm`, or `Z` for UTC). Example usage without a timezone:

```
gcovr --timestamp '2021-12-27 13:05:27'
```

Example usages that show equivalent specifications for UTC timestamps:

```
gcovr --timestamp '2021-12-27T13:05:27Z'
gcovr --timestamp '2021-12-27T13:05:27+00:00'
gcovr --timestamp '2021-12-27T13:05:27-00:00'
```

Differences and clarifications with respect to RFC-3339:

- the time zone may be omitted
- the date and time parts may be separated by a space character instead of the `T`
- the date is parsed in a case insensitive manner
- sub-second accuracy is not currently supported

Additional formats may be added in the future. To ensure that timestamps are handled in the expected manner, it is possible to select a particular timestamp syntax with a prefix.

- Epoch timestamps can be selected with a `@` or `epoch:` prefix.
- RFC-3339 timestamps can be selected with a `rfc3339:` prefix.

Examples of prefixes:

```
gcovr --timestamp @1640606727
gcovr --timestamp epoch:1640606727
gcovr --timestamp 'rfc3339:2021-12-27 13:05:27'
```

3.8.2 Using timestamps from Git commits

As an example of using the timestamp feature, we might want to attribute a coverage report to the time when a Git commit was created. Git lets us extract the commit date from a commit with the git show command. For the current HEAD commit:

```
git show --no-patch --format=%cI HEAD
```

This can be combined into a Bash one-liner like this:

```
gcovr --timestamp="$(git show --no-patch --format=%cI HEAD)"
```

Each Git commit has two dates, the author date and the committer date. This information can be extracted with various format codes, e.g. `%aI` for the author date and `%cI` for the committer date. These format codes are also available in different formats. The supported Git formats are:

- Unix timestamps: `%at`, `%ct`
- "Strict ISO" format: `%aI`, `%cI`
- depending on the `--date` option: `%ad`, `%cd`

Git's `--date` option is documented in git log. The supported settings are:

- Unix timestamps: `--date=unix`

- "Strict ISO" format: `--date=iso-strict`, `--date=iso8601-strict`, `--date=iso-strict-local`, `--date=iso8601-strict-local`

Related documents:

- *Installation*
- *Getting Started*
- *Command Line Reference*
- *Cookbook*
- *Frequently Asked Questions*
- *Contributing* (includes instructions for bug reports)
- *Change Log*
- *License*

CHAPTER 4

Command Line Reference

The `gcovr` command recursively searches a directory tree to find `gcov` coverage files, and generates a text summary of the code coverage. The *`-h/--help`* option generates the following summary of the `gcovr` command line options:

4.1 gcovr

A utility to run gcov and summarize the coverage in simple reports.

```
usage: gcovr [options] [search_paths...]
```

See <http://gcovr.com/> for the full manual.

4.1.1 Options

`search_paths`
: Search these directories for coverage files. Defaults to –root and –object-directory. Config key: search-path.

`-h, --help`
: Show this help message, then exit.

`--version`
: Print the version number, then exit.

`-v, --verbose`
: Print progress messages. Please include this output in bug reports.

`-r` `<root>`**`, --root`** `<root>`
: The root directory of your source files. Defaults to '.', the current directory. File names are reported relative to this root. The –root is the default –filter.

`-a` `<add_tracefile>`**`, --add-tracefile`** `<add_tracefile>`
: Combine the coverage data from JSON files. Coverage files contains source files structure relative to root directory. Those structures are combined in the output relative to the current root directory. Unix style wildcards

can be used to add the pathnames matching a specified pattern. In this case pattern must be set in double quotation marks. Option can be specified multiple times. When option is used gcov is not run to collect the new coverage data.

--config <config>
Load that configuration file. Defaults to gcovr.cfg in the –root directory.

--no-markers
Turn off exclusion markers. Any exclusion markers specified in source files will be ignored.

--fail-under-line <min>
Exit with a status of 2 if the total line coverage is less than MIN. Can be ORed with exit status of '–fail-under-branch' option.

--fail-under-branch <min>
Exit with a status of 4 if the total branch coverage is less than MIN. Can be ORed with exit status of '–fail-under-line' option.

--source-encoding <source_encoding>
Select the source file encoding. Defaults to the system default encoding (UTF-8).

--exclude-lines-by-pattern <exclude_lines_by_pattern>
Exclude lines that match this regex.

--exclude-pattern-prefix <exclude_pattern_prefix>
Define the regex prefix used in markers / line exclusions (i.e ..._EXCL_START, ..._EXCL_START, ..._EXCL_STOP)

4.1.2 Output Options

Gcovr prints a text report by default, but can switch to XML or HTML.

-o <output>, **--output** <output>
Print output to this filename. Defaults to stdout. Individual output formats can override this.

-b, --branches
Report the branch coverage instead of the line coverage. For text report only. Config key: txt-branch.

--decisions
Report the decision coverage. For HTML and JSON report.

-u, --sort-uncovered
Sort entries by increasing number of uncovered lines. For text and HTML report.

-p, --sort-percentage
Sort entries by increasing percentage of uncovered lines. For text and HTML report.

--txt <output>
Generate a text report. OUTPUT is optional and defaults to –output.

--cobertura <output>, **-x** <output>, **--xml** <output>
Generate a Cobertura XML report. OUTPUT is optional and defaults to –output.

--cobertura-pretty, --xml-pretty
Pretty-print the Cobertura XML report. Implies –cobertura. Default: False.

--html <output>
Generate a HTML report. OUTPUT is optional and defaults to –output.

--html-details <output>
Add annotated source code reports to the HTML report. Implies –html. OUTPUT is optional and defaults to –output.

--html-details-syntax-highlighting
Use syntax highlighting in HTML details page. Enabled by default. Negation: –no-html-details-syntax-highlighting.

--html-theme {green,blue}
Override the default color theme for the HTML report. Default is green.

--html-css <css>
Override the default style sheet for the HTML report.

--html-title <title>
Use TITLE as title for the HTML report. Default is 'GCC Code Coverage Report'.

--html-medium-threshold <medium>
If the coverage is below MEDIUM, the value is marked as low coverage in the HTML report. MEDIUM has to be lower than or equal to value of –html-high-threshold and greater than 0. If MEDIUM is equal to value of –html-high-threshold the report has only high and low coverage. Default is 75.0.

--html-high-threshold <high>
If the coverage is below HIGH, the value is marked as medium coverage in the HTML report. HIGH has to be greater than or equal to value of –html-medium-threshold. If HIGH is equal to value of –html-medium-threshold the report has only high and low coverage. Default is 90.0.

--html-tab-size <html_tab_size>
Used spaces for a tab in a source file. Default is 4

--html-absolute-paths
Use absolute paths to link the –html-details reports. Defaults to relative links.

--html-encoding <html_encoding>
Override the declared HTML report encoding. Defaults to UTF-8. See also –source-encoding.

--html-self-contained
Control whether the HTML report bundles resources like CSS styles. Self-contained reports can be sent via email, but conflict with the Content Security Policy of some web servers. Defaults to self-contained reports unless –html-details is used. Negation: –no-html-self-contained.

-s, --print-summary
Print a small report to stdout with line & function & branch percentage coverage. This is in addition to other reports. Default: False.

--sonarqube <output>
Generate sonarqube generic coverage report in this file name. OUTPUT is optional and defaults to –output.

--json <output>
Generate a JSON report. OUTPUT is optional and defaults to –output.

--json-pretty
Pretty-print the JSON report. Implies –json. Default: False.

--json-summary <output>
Generate a JSON summary report. OUTPUT is optional and defaults to –output.

--json-summary-pretty
Pretty-print the JSON SUMMARY report.Implies –json-summary. Default: False.

--csv <output>
Generate a CSV summary report. OUTPUT is optional and defaults to –output.

--coveralls <output>
Generate Coveralls API coverage report in this file name. OUTPUT is optional and defaults to –output.

--coveralls-pretty
Pretty-print the coveralls report. Implies –coveralls. Default: False.

--timestamp <timestamp>
Override current time for reproducible reports. Can use *YYYY-MM-DD hh:mm:ss* or epoch notation. Used by HTML, Coveralls, and Cobertura reports. Default: current time.

4.1.3 Filter Options

Filters decide which files are included in the report. Any filter must match, and no exclude filter must match. A filter is a regular expression that matches a path. Filter paths use forward slashes, even on Windows. If the filter looks like an absolute path it is matched against an absolute path. Otherwise, the filter is matched against a relative path, where that path is relative to the current directory or if defined in a configuration file to the directory of the file.

-f <filter>, **--filter** <filter>
Keep only source files that match this filter. Can be specified multiple times. Relative filters are relative to the current working directory or if defined in a configuration file. If no filters are provided, defaults to –root.

-e <exclude>, **--exclude** <exclude>
Exclude source files that match this filter. Can be specified multiple times.

--gcov-filter <gcov_filter>
Keep only gcov data files that match this filter. Can be specified multiple times.

--gcov-exclude <gcov_exclude>
Exclude gcov data files that match this filter. Can be specified multiple times.

--exclude-directories <exclude_dirs>
Exclude directories that match this regex while searching raw coverage files. Can be specified multiple times.

4.1.4 GCOV Options

The 'gcov' tool turns raw coverage files (.gcda and .gcno) into .gcov files that are then processed by gcovr. The gcno files are generated by the compiler. The gcda files are generated when the instrumented program is executed.

--gcov-executable <gcov_cmd>
Use a particular gcov executable. Must match the compiler you are using, e.g. 'llvm-cov gcov' for Clang. Can include additional arguments. Defaults to the GCOV environment variable, or 'gcov': 'gcov'.

--include-internal-functions
Include function coverage of compiler internal functions (starting with '__' or '_GLOBAL__sub_I_').

--exclude-unreachable-branches
Exclude branch coverage from lines without useful source code (often, compiler-generated "dead" code). Default: False.

--exclude-function-lines
Exclude coverage from lines defining a function Default: False.

--exclude-throw-branches
For branch coverage, exclude branches that the compiler generates for exception handling. This often leads to more "sensible" coverage reports. Default: False.

-g, --use-gcov-files
Use existing gcov files for analysis. Default: False.

--gcov-ignore-parse-errors
Skip lines with parse errors in GCOV files instead of exiting with an error. A report will be shown on stderr. Default: False.

--object-directory <objdir>
: Override normal working directory detection. Gcovr needs to identify the path between gcda files and the directory where the compiler was originally run. Normally, gcovr can guess correctly. This option specifies either the path from gcc to the gcda file (i.e. gcc's '-o' option), or the path from the gcda file to gcc's working directory.

-k, --keep
: Keep gcov files after processing. This applies both to files that were generated by gcovr, or were supplied via the –use-gcov-files option. Default: False. Config key: keep-gcov-files.

-d, --delete
: Delete gcda files after processing. Default: False. Config key: delete-gcov-files.

-j <gcov_parallel>
: Set the number of threads to use in parallel. Config key: gcov-parallel.

For guide-level explanation on using these options, see the *User Guide*.

CHAPTER 5

Cookbook

This section contains how-to guides on creating code coverage reports for various purposes. For an introduction on using gcovr, see the *User Guide* instead.

Recipes in the cookbook:

5.1 How to collect coverage for C extensions in Python

Collecting code coverage data on the C code that makes up a Python extension module is not quite as straightforward as with a regular C program.

As with a normal C project, we have to compile our code with coverage instrumentation. Here, we `export CFLAGS="--coverage"` and then run `python3 setup.py build_ext`.

Unfortunately, `build_ext` can rebuild a source file even if the current object file is up to date. If multiple extension modules share the same source code file, gcov will get confused by the different timestamps and report inaccurate coverage. It is nontrivial to adapt the `build_ext` process to avoid this.

Instead, we can use the `ccache` utility to make the compilation lazy (works best on Unix systems). Before we invoke the `build_ext` step, we first `export CC="ccache gcc"`. Ccache works well but isn't absolutely perfect, see the ccache manual for caveats.

A shell session might look like this:

```
# Set required env vars
export CFLAGS="--coverage"
export CC="ccache gcc"
```

(continues on next page)

(continued from previous page)

```
# clear out build files so we get a fresh compile
rm -rf build/temp.*  # contains old .gcda, .gcno files
rm -rf build/lib.*

# rebuild extensions
python3 setup.py build_ext --inplace  # possibly --force

# run test command i.e. pytest

# run gcovr
rm -rf coverage; mkdir coverage
gcovr --filter src/ --print-summary --html-details coverage/index.html
```

5.2 Out-of-Source Builds with CMake

Tools such as `cmake` encourage the use of out-of-source builds, where the code is compiled in a directory other than the one which contains the sources. This is an extra complication for `gcov`. In order to pass the correct compiler and linker flags, the following commands need to be in `CMakeLists.txt`:

```
# This flags are used if cmake is called with -DCMAKE_BUILD_TYPE=PROFILE
set(CMAKE_C_FLAGS_PROFILE --coverage)
set(CMAKE_CXX_FLAGS_PROFILE --coverage)

add_executable(program example.cpp)
```

The `--coverage` compiler flag is an alternative to `-fprofile-arcs -ftest-coverage` for recent version of gcc. In versions 3.13 and later of `cmake`, the `target_link_libraries` command can be removed and `add_link_options("--coverage")` added after the `add_compile_options` command.

We then follow a normal `cmake` build process:

```
cd $BLD_DIR
cmake -DCMAKE_BUILD_TYPE=PROFILE $SRC_DIR
make VERBOSE=1
```

and run the program:

```
cd $BLD_DIR
./program
```

However, invocation of `gcovr` itself has to change. The assorted `.gcno` and `.gcda` files will appear under the `CMakeFiles` directory in `BLD_DIR`, rather than next to the sources. Since `gcovr` requires both, the command we need to run is:

```
cd $BLD_DIR
gcovr -r $SRC_DIR .
```

CHAPTER 6

Frequently Asked Questions

6.1 What is the difference between lcov and gcovr?

Both lcov and gcovr are tools to create coverage reports.

Gcovr was originally created as a simple script to provide a convenient command line interface to gcov that produced more easily digestible output similar to Python's coverage utilities.

Later, we added XML output that could be used with the Cobertura plugin of the Jenkins continuous integration server. This gave us nice coverage reports for C/C++ code in Jenkins.

HTML output was added much later. If all you need is HTML, pick whichever one produces the output you like better or integrates easier with your existing workflow.

Lcov is a far older project that is part of the Linux Test Project. It provides some features that gcovr does not have: For example, lcov has explicit support for capturing Linux kernel coverage. Lcov also supports various trace file manipulation functions such as merging trace files from different test runs. You can learn more at the lcov website or the lcov GitHub repository.

6.2 Why does C++ code have so many uncovered branches?

Gcovr's branch coverage reports are based on GCC's `-profile-arcs` feature, which uses the compiler's control flow graph (CFG) of each function to determine branches. This is a very low-level view: to understand the branches in a given function, it can help to view the function's assembly, e.g. via the Godbolt Compiler Explorer.

What gcovr calls a *branch* is in fact an *arc* between basic blocks in the CFG. This means gcovr's reports have many branches that are not caused by `if` statements! For example:

- Arcs are caused by C/C++ branching operators: `for`, `if`, `while`, `switch`/`case`, `&&`, `||`, `? :`. Note that switches are often compiled as a decision tree which introduces extra arcs, not just one per case.
- (Arcs into another function are not shown.)

- Arcs are caused when a function that may throw returns: one arc to the next block or statement for normal returns, and one arc to an exception handler for exceptions, if this function contains an exception handler. Every local variable with a destructor is an exception handler as well.
- Compiler-generated code that deals with exceptions often needs extra branches: `throw` statements, `catch` clauses, and destructors.
- Extra arcs are created for `static` initialization and destruction.
- Arcs may be added or removed by compiler optimizations. If you compile without optimizations, some arcs may even be unreachable!

Gcovr is not able to *remove* any "unwanted" branches because GCC's gcov tool does not make the necessary information available, and because different projects are interested in different kinds of branches. However, gcovr has the following options to *reduce* unwanted branches:

With the *`gcovr --exclude-unreachable-branches`* option, gcovr parses the *source code* to see whether that line even contains any code. If the line is empty or only contains curly braces, this could be an indication of compiler-generated code that was mis-attributed to that line (such as that for static destruction) and branch coverage will be ignored on that line.

With the *`gcovr --exclude-throw-branches`* option, exception-only branches will be ignored. These are typically arcs from a function call into an exception handler.

With the *`gcovr --decisions`* option, gcovr parses the source code to extract a ISO 26262 compliant metric for decision coverage. This metric can be interpreted as the branch coverage on C/C++-Level. While the feature is not always able to detect the decisions reliabily when the code is written very compact (uncheckable decisions will be marked), it provides a reliable tool for (i.e. MISRA-compliant) code in security-relevant situations.

Compiling with optimizations will typically remove unreachable branches and remove superfluous branches, but makes the coverage report less exact. For example, branching operators might be optimized away. Decision coverage analysis will be very buggy when compiling with optimizations. See also: Gcov and Optimization in the GCC documentation.

Despite these approaches, 100% branch coverage will be impossible for most programs.

6.3 Why are uncovered files not reported?

Gcovr does report files that have zero coverage, even when no .gcda file is available for that compilation unit.

However, the gcov tool in some versions of GCC refuses to generate output for uncovered files.

To fix this, upgrade GCC to:

- version 5.5 or later,
- version 6.2 or later, or
- any version since 7.

Note that the compiler may ignore `inline` functions that are never used.

6.4 Which options are used for calling gcov?

The options used for calling `gcov` depends on the version of gcov.

Following options are always used:

- `--branch-counts`

- `--branch-probabilities`
- `--object-directory`

Following options are only used if available:

- `--demangled-names`: Not available for LLVM based `gcov`.
- `--hash-filenames`: Available since GCC 7, as fallback the option `--preserve-paths` is used.

CHAPTER 7

Contributing

This document contains:

- our *guidelines for bug reports*
- *general contribution guidelines*
- a *checklist for pull requests*
- a developer guide that explains the *development environment*, *project structure*, and *test suite*

7.1 How to report bugs

When reporting a bug, first search our issues to avoid duplicates. In your bug report, please describe what you expected gcovr to do, and what it actually did. Also try to include the following details:

- how you invoked gcovr, i.e. the exact flags and from which directory
- your project layout
- your gcovr version
- your compiler version
- your operating system
- and any other relevant details.

Ideally, you can provide a short script and the smallest possible source file to reproduce the problem.

7.2 How to help

If you would like to help out, please take a look at our open issues and pull requests. The issues labeled help wanted and needs review would have the greatest impact.

There are many ways how you can help:

- assist other users with their problems
- share your perspective as a gcovr user in discussions
- test proposed changes in your real-world projects
- improve our documentation
- submit pull requests with bug fixes and enhancements

7.3 How to submit a Pull Request

Thank you for helping with gcovr development! Please follow this checklist for your pull request:

- **Is this a good approach?** Fixing open issues is always welcome! If you want to implement an enhancement, please discuss it first as a GitHub issue.
- **Does it work?** Please run the tests locally:

  ```
  python3 -m nox
  ```

  (see also: *Test suite*)

  In any case, the tests will run automatically when you open the pull request. But please prevent unnecessary build failures and run the tests yourself first. If you cannot run the tests locally, you can activate GitHub for your fork, or run the tests with Docker. If there are differences the updated files will be available for download from the CI system (one ZIP for each test environment).

  If you add new features, please try to add a test case.

- **Does it conform to the style guide?** The source code should conform to the **PEP 8** standard. Please check your code:

  ```
  python3 -m nox --session lint

  # or:

  python3 -m flake8 doc gcovr
  ```

  The command `python3 -m nox` will run the linter, run the tests, and check that the docs can be built.

- **Add yourself as an author.** If this is your first contribution to gcovr, please add yourself to the `AUTHORS.txt` file.
- **One change at a time.** Please keep your commits and your whole pull request fairly small, so that the changes are easy to review. Each commit should only contain one kind of change, e.g. refactoring *or* new functionality.
- **Why is this change necessary?** When you open the PR, please explain why we need this change and what your PR does. If this PR fixes an open issue, reference that issue in the pull request description. Add a reference to the issue in the `CHANGELOG.rst`, if the change should not be visible in the changelog (minor or not of interest), add the following string to a single line in the PR body:

  [no changelog]

Once you submit the PR, it will be automatically tested on Windows and Linux, and code coverage will be collected. Your code will be reviewed. This can take a week. Please fix any issues that are discovered during this process. Feel free to force-push your updates to the pull request branch.

If you need assistance for your pull request, you can

- chat in our Gitter room

- discuss your problem in an issue
- open an unfinished pull request as a work in progress (WIP), and explain what you've like to get reviewed

7.4 How to set up a development environment

For working on gcovr, you will need a supported version of Python 3, GCC version 5, 6 or 8 (other GCC versions are supported by gcovr, but will cause spurious test failures) or clang version 10, `make`, `cmake` and `ninja`. Please make sure that the tools are in the system `PATH`. On **Windows**, you will need to install a GCC toolchain as the tests expect a Unix-like environment. You can use MinGW-W64 or MinGW. An easier way is to *run tests with Docker*, on **Windows** a Pro license or the WSL (Windows subsystem for Linux) is needed.

- Check your GCC installation, the binary directory must be added to the PATH environment. If the command gcc-5/g++-5/gcov-5, gcc-6/g++-6/gcov-6, gcc-8/g++-8/gcov-8, gcc-9/g++-9/gcov-9, clang-10/clang++-10/llvm-cov or clang-13/clang++-13/llvm-cov are available everything is OK. For gcc-6, gcc-8, gcc-9, clang-10 and clang-13 you should use the option `CC=...` see *run and filter tests*. If this isn't OK you can set the reference data to use by setting the environment `CC_REFERENCE=gcc-8` or you have to create symlinks for the gcc executables with the following steps. You can check the GCC version with gcc –version. If the output says version 8, you should also be able to run gcc-8 –version. Your Linux distribution should have set all of this up already. If you don't have an alias like gcc-8, perform the following steps to create an alias for gcc, this should also work in the MSYS shell under Windows:
  1. Create a directory somewhere, e.g. in your home directory: `mkdir ~/bin`
  2. Create a symlink in that directory which points to GCC: `ln -s $(which gcc) ~/bin/gcc-8`
  3. Add this directory to your PATH: `export PATH="$HOME/bin:$PATH"`
  4. Re-test `gcc-8 --version` to ensure everything worked.
  5. Create additional symlinks for g++ -> g++-8 and gcov -> gcov-8.
- (Optional) Fork the project on GitHub.
- Clone the git repository.
- (Optional) Set up a virtualenv (e.g. with `python3 -m venv my-venv`)
- Install gcovr in development mode, and install nox:

```
pip install -e .
pip install nox
```

  You can then run gcovr as `gcovr` or `python3 -m gcovr`.

  Run the tests to verify that everything works (see *Test suite*).
- (Optional) Activate GitHub Actions for your forked repository, so that the cross-platform compatibility tests get run whenever you push your work to your repository. These tests will also be run when you open a pull request to the main gcovr repository.

Tip: If you have problems getting everything set up, consider looking at these files:

- for Linux: `.github/workflows/test.yml` and `admin/Dockerfile.qa`
- for Windows: `.github/workflows/test.yml`

7.5 Project Structure

Path	Description
/	project root
/gcovr/	the gcovr source code (Python module)
/gcovr/__main__.py	command line interface + top-level behaviour
/gcovr/templates/	HTML report templates
/gcovr/tests/	unit tests + integration test corpus
/noxfile.py	Definition of tests tasks
/setup.py	Python package configuration
/doc/	documentation
/doc/sources/	user guide + website
/doc/examples/	runnable examples for the user guide

The program entrypoint and command line interface is in `gcovr/__main__.py`. The coverage data is parsed in the `gcovr.gcov` module. The HTML, XML, text, and summary reports are in `gcovr.generator.html` and respective modules.

7.6 Test suite

The QA process (`python3 -m nox`) consists of multiple parts:

- linting and checking format(`python3 -m nox --session lint`)
- tests (`python3 -m nox --session tests`)
  - unit tests in `gcovr/tests`
  - integration tests in `gcovr/tests`
  - documentation examples in `doc/examples`
- documentation build (`python3 -m nox --session doc`)

The tests are in the `gcovr/tests` directory. You can run the tests with `python3 -m nox --session tests` for the default GCC version (specified via `CC` environment variable, defaults to gcc-5). You can also select the gcc version if you run the tests with e.g. `python3 -m nox --session 'tests_compiler(gcc-8)'`.

There are unit tests for some parts of gcovr, and a comprehensive corpus of example projects that are executed as the `test_gcovr.py` integration test. Each `gcovr/tests/*` directory is one such example project.

You can format files with `python3 -m nox --session black FileToFormat`)

To get a list of all available sessions run `python3 -m nox -l`.

The next sections discuss the *structure of integration tests*, how to *run and filter tests*, and how to *run tests with Docker*.

7.6.1 Structure of integration tests

Each project in the corpus contains a `Makefile` and a `reference` directory:

```
gcovr/tests/sometest/
  reference/
  Makefile
```

(continues on next page)

(continued from previous page)

```
  README
  example.cpp
```

The Makefile controls how the project is built, and how gcovr should be invoked. The reference directory contains baseline files against which the gcovr output is compared. Tests can be executed even without baseline files.

Each Makefile contains the following targets:

- `all`: builds the example project. Can be shared between gcovr invocations.
- `run`: lists available targets which must be a subset of the available output formats.
- `clean`: remove any generated files after all tests of the scenario have finished.
- output formats (txt, html, json, sonarqube, ...): invoke gcovr to produce output files of the correct format. The test runner automatically finds the generated files (if any) and compares them to the baseline files in the reference directory. All formats are optional, but using at least JSON is recommended.
- `clean-each`: if provided, will be invoked by the test runner after testing each format.

7.6.2 Run and filter tests

To run all tests, use `python3 -m nox`. The tests currently assume that you are using GCC 5 and have set up a *development environment*. You can select a different GCC version by setting the CC environment variable. Supported versions are `CC=gcc-5`, `CC=gcc-6`, `CC=gcc-8`, `CC=gcc-9`, `clang-10` and `clang-13`.

You can run the tests with additional options by adding `--` and then the options to the test invocation. Run all tests after each change is a bit slow, therefore you can limit the tests to a specific test file, example project, or output format. For example:

```
# run only XML tests
python3 -m nox --session tests -- -k 'xml'

# run the simple1 tests
python3 -m nox --session tests -- -k 'simple1'

# run the simple1 tests only for XML
python3 -m nox --session tests -- -k 'xml and simple1'
```

To see which tests would be run, add the `--collect-only` option:

```
#see which tests would be run
python3 -m nox --session tests -- --collect-only
```

Sometimes during development you need to create reference files for new test or update the current reference files. To do this you have to add `--generate_reference` or `--update-reference` option to the test invocation. By default generated output files are automatically removed after test run. To skip this process you can add `--skip_clean` option the test invocation. For example:

```
# run tests and generate references for simple1 example
python3 -m nox --session tests -- -k 'simple1' --generate_reference

# run tests and update xml references for simple1 example
python3 -m nox --session tests -- -k 'xml and simple1' --update_reference

# run only XML tests and do not remove generated files
python3 -m nox --session tests -- -k 'xml' --skip_clean
```

To update the refernce data for all compiler in one call see *run tests with Docker*.

When the currently generated output reports differ to the reference files you can create a ZIP archive named `diff.zip` in the tests directory by using `--archive_differences` option. Currently in gcovr it is used by GitHub CI to create a ZIP file with the differences as an artifact.

```
# run tests and generate a ZIP archive when there were differences
python3 -m nox --session tests -- --archive_differences
```

Changed in version 5.1: Change how to start test from `make test` to `python3 -m nox --session tests`

New in version 5.0: Added test options *–generate_reference*, *–update_reference*, *–skip_clean*, '–archive_differences' and changed way to call tests only by `make test`.

7.6.3 Run tests with Docker

If you can't set up a toolchain locally, you can run the QA process via Docker. First, build the container image:

```
python3 -m nox --session docker_qa_build
```

Then, run the container, which executes `python3 -m nox` within the container:

```
python3 -m nox --session docker_qa_run
```

Or to build and run the container in one step:

```
python3 -m nox --session docker_qa
```

You can select the gcc version to use inside the docker by setting the environment variable CC to gcc-5 (default), gcc-6, gcc-8, gcc-9, clang-10 or clang-13 or you can build and run the container with:

```
python3 -m nox --session 'docker_qa_compiler(gcc-9)'
```

You can also use the compiler 'all' to run the tests for all compiler versions. This is usefull to update the all reference files:

```
python3 -m nox --session 'docker_qa_compiler(all)' -- --update_reference
```

7.7 Become a gcovr developer

After you've contributed a bit (whether with discussions, documentation, or code), consider becoming a gcovr developer. As a developer, you can:

- manage issues and pull requests (label and close them)
- review pull requests (a developer must approve each PR before it can be merged)
- participate in votes

Just open an issue that you're interested, and we'll have a quick vote.

CHAPTER 8

Change Log

`gcovr` Release History and Change Log

8.1 5.1 (26 March 2022)

Breaking changes:

- Dropped support for Python 3.6 (#550)
- Changed `xml` configuration key to `cobertura` (#552)
- JSON summary output: all percentages are now reported from 0 to 100 (#570)

New features and notable changes:

- Report function coverage (#362, #515, #554)
- Consistent support for symlinks across operating systems
  - Support for Windows junctions (#535)
  - Symlinks are only resolved for *evaluating filters* (#565)
- Show error message on STDERR when `--fail-under-line` or `--fail-under-branch` fails (#502)
- Can report decision coverage with `--decisions` option (reasonably formatted C/C++ source files only, HTML and JSON output) (#350)
- Can create reproducible reports with the `--timestamp` option (#546)
- Improvements to *Exclusion Markers* (LINE/START/STOP)
  - Can ignore markers in code with `--no-markers` option (#361)
  - Can customize patterns with `--exclude-pattern-prefix` option (#561)
- Can use `--cobertura` as a less ambiguous alias for `--xml`. (#552)

Bug fixes and small improvements:

- Gcov is invoked without localization by setting LC_ALL=C (#513)
- Gcov is invoked without temporary directories (#525)
- Gcov: solved problems with file name limitations. (#528)
- Fixed "root" path in JSON summary report. (#548)
- Correctly resolve relative filters in configuration files. (#568)
- HTML output: indicate lines with excluded coverage (#503)
- HTML output: fixed sanity check to support empty files (#571)
- HTML output: support `jinja2 >= 3.1` (#576)

Documentation:

- Split documentation into smaller pages (#552)
- Document used options for `gcov` (#528)

Internal changes:

- Replaced own logger with Python's logging module. (#540)
- New parser for `.gcov` file format, should be more robust. (#512)
- New tests
  - more compilers: clang-10 (#484), clang-13 (#527), gcc-9 (#527)
  - `-fprofile-abs-path` compiler option (#521)
  - enabled symlink tests for Windows (#539)
- Improvements to the test suite
  - Use Nox instead of Makefiles to manage QA checks (#516, #555)
  - Can run tests for all compiler versions in one go (#514)
  - More linter checks (#566) and code style enforcement with black (#579)
  - Better XML diffing with yaxmldiff (#495, #509)
  - Share test reference data between compiler versions where possible (#556)
  - Better environment variable handling (#493, #541)
  - Fixed glob patterns for collecting reference files (#533)
  - Add timeout for each single test. (#572)
- Improvements and fixes to the release process (#494, #537)
- Normalize shell scripts to Unix line endings (#538, #547)

8.2 5.0 (11 June 2021)

Breaking changes:

- Dropped support for Python 2 and Python 3.5. From now on, gcovr will only support Python versions that enjoy upstream support.

Improvements and new features:

- Handles spaces in `gcov` path. (#385)

- Early fail when output cannot be created. (#382)
- Add `--txt` for text output. (#387)
- Add `--csv` for CSV output. (#376)
- Add `--exclude-lines-by-pattern` to filter out source lines by arbitrary regex. (#356)
- Add `--json-summary` to generate a *JSON Summary* report. (#366)
- Add `--coveralls` to generate a *Coveralls* compatible JSON report. (#328)
- Add support for output directories. If the output ends with a `/` or `\` it is used as a directory. (#416)
- Compare paths case insensitive if file system of working directory is case insensitive. (#329)
- Add wildcard pattern to json `--add-tracefile`. (#351)
- Enable `--filter` and `--exclude` for *Merging coverage*. (#373)
- Only output 100.0% in text and HTML output if really 100.0%, else use 99.9%. (#389)
- Support relative source location for shadow builds. (#410)
- Incorrect path for header now can still generate html-details reports (#271)
- Change format version in JSON output from number to string and update it to "0.2". (#418, #463)
- Only remove `--root` path at the start of file paths. (#452)
- Fix coverage report for cmake ninja builds with given in-source object-directory. (#453)
- Add issue templates. (#461)
- Add `--exclude-function-lines` to exclude the line of the function definition in the coverage report. (#430)
- Changes for HTML output format:
  - Redesign HTML generation. Add `--html-self-contained` to control external or internal CSS. (#367)
  - Change legend for threshold in html report. (#371)
  - Use HTML title also for report heading. Default value for `--html-title` changed. (#378)
  - Add `--html-tab-size` to configure tab size in HTML details. (#377)
  - Add option `--html-css` for user defined styling. (#380)
  - Create details html filename independent from OS. (#375)
  - Add `--html-theme` to change the color theme. (#393)
  - Add linkable lines in HTML details. (#401)
  - Add syntax highlighting in the details HTML report. This can be turned off with `--no-html-details-syntax-highlighting`. (#402, #415)

Documentation:

- Cookbook: *Out-of-Source Builds with CMake* (#340, #341)

Internal changes:

- Add makefile + dockerfile for simpler testing.
- Add .gitbugtraq to link comments to issue tracker in GUIs. (#429)
- Add GitHub actions to test PRs and master branch. (#404)

- Remove Travis CI. (#419)
- Remove Appveyor CI and upload coverage report from Windows and Ubuntu from the GitHub actions. (#455)
- Add check if commit is mentioned in the CHANGELOG.rst. (#457)
- Move flake8 config to setup.cfg and add black code formatter. (#444)
- Fix filter/exclude relative path issue in Windows. (#320, #479)
- Extend test framework for CI:
  - Set make variable TEST_OPTS as environment variable inside docker. (#372)
  - Add make variable USE_COVERAGE to extend flags for coverage report in GitHub actions. (#404)
  - Extend tests to use an unified diff in the assert. Add test options *–generate_reference*, *–update_reference* and *–skip_clean*. (#379)
  - Support multiple output patterns in integration tests. (#383)
  - New option *–archive_differences* to save the different files as ZIP. Use this ZIP as artifact in AppVeyor. (#392)
  - Add support for gcc-8 to test suite and docker tests. (#423)
  - Run as limited user inside docker container and add test with read only directory. (#445)

8.3 4.2 (6 November 2019)

Breaking changes:

- Dropped support for Python 3.4.
- Format flag parameters like `--xml` or `--html` now take an optional output file name. This potentially changes the interpretation of search paths. In `gcovr --xml foo`, previous gcovr versions would search the `foo` directory for coverage data. Now, gcovr will try to write the Cobertura report to the `foo` file. To keep the old meaning, separate positional arguments like `gcovr --xml -- foo`.

Improvements and new features:

- *Configuration file* support (experimental). (#167, #229, #279, #281, #293, #300, #304)
- *JSON output*. (#301, #321, #326)
- *Merging coverage* with `gcovr --add-tracefile`. (#10, #326)
- *SonarQube XML Output*. (#308)
- Handle cyclic symlinks correctly during coverage data search. (#284)
- Simplification of `--object-directory` heuristics. (#18, #273, #280)
- Exception-only code like a `catch` clause is now shown as uncovered. (#283)
- New `--exclude-throw-branches` option to exclude exception handler branches. (#283)
- Support `--root ..` style invocation, which might fix some CMake-related problems. (#294)
- Fix wrong names in report when source and build directories have similar names. (#299)
- Stricter argument handling. (#267)
- Reduce XML memory usage by moving to lxml. (#1, #118, #307)

- Can write *multiple reports* at the same time by giving the output file name to the report format parameter. Now, `gcovr --html -o cov.html` and `gcovr --html cov.html` are equivalent. (#291)
- Override gcov locale properly. (#334)
- Make gcov parser more robust when used with GCC 8. (#315)

Known issues:

- The `--keep` option only works when using existing gcov files with `-g`/`--use-gcov-files`. (#285, #286)
- Gcovr may get confused when header files in different directories have the same name. (#271)
- Gcovr may not work when no en_US locale is available. (#166)

Documentation:

- *Exclusion marker* documentation.
- FAQ: *Why does C++ code have so many uncovered branches?* (#283)
- FAQ: *Why are uncovered files not reported?* (#33, #100, #154, #290, #298)

Internal changes:

- More tests. (#269, #268, #269)
- Refactoring and removal of dead code. (#280)
- New internal data model.

8.4 4.1 (2 July 2018)

- Fixed/improved –exclude-directories option. (#266)
- New "Cookbook" section in the documentation. (#265)

8.5 4.0 (17 June 2018)

Breaking changes:

- This release drops support for Python 2.6. (#250)
- PIP is the only supported installation method.
- No longer encoding-agnostic under Python 2.7. If your source files do not use the system encoding (probably UTF-8), you will have to specify a –source-encoding. (#148, #156, #256)
- Filters now use forward slashes as path separators, even on Windows. (#191, #257)
- Filters are no longer normalized into pseudo-paths. This could change the interpretation of filters in some edge cases.

Improvements and new features:

- Improved –help output. (#236)
- Parse the GCC 8 gcov format. (#226, #228)
- New –source-encoding option, which fixes decoding under Python 3. (#256)
- New –gcov-ignore-parse-errors flag. By default, gcovr will now abort upon parse errors. (#228)
- Detect the error when gcov cannot create its output files (#243, #244)

- Add -j flag to run gcov processes in parallel. (#3, #36, #239)
- The –html-details flag now implies –html. (#93, #211)
- The –html output can now be used without an –output filename (#223)
- The docs are now managed with Sphinx. (#235, #248, #249, #252, #253)
- New –html-title option to change the title of the HTML report. (#261, #263)
- New options –html-medium-threshold and –html-high-threshold to customize the color legend. (#261, #264)

Internal changes:

- Huge refactoring. (#214, #215, #221 #225, #228, #237, #246)
- Various testing improvements. (#213, #214, #216, #217, #218, #222, #223, #224, #227, #240, #241, #245)
- HTML reports are now rendered with Jinja2 templates. (#234)
- New contributing guide. (#253)

8.6 3.4 (12 February 2018)

- Added –html-encoding command line option (#139).
- Added –fail-under-line and –fail-under-branch options, which will error under a given minimum coverage. (#173, #116)
- Better pathname resolution heuristics for –use-gcov-file. (#146)
- The –root option defaults to current directory '.'.
- Improved reports for "(", ")", ";" lines.
- HTML reports show full timestamp, not just date. (#165)
- HTML reports treat 0/0 coverage as NaN, not 100% or 0%. (#105, #149, #196)
- Add support for coverage-04.dtd Cobertura XML format (#164, #186)
- Only Python 2.6+ is supported, with 2.7+ or 3.4+ recommended. (#195)
- Added CI testing for Windows using Appveyor. (#189, #200)
- Reports use forward slashes in paths, even on Windows. (#200)
- Fix to support filtering with absolute paths.
- Fix HTML generation with Python 3. (#168, #182, #163)
- Fix –html-details under Windows. (#157)
- Fix filters under Windows. (#158)
- Fix verbose output when using existing gcov files (#143, #144)

8.7 3.3 (6 August 2016)

- Added CI testing using TravisCI
- Added more tests for out of source builds and other nested builds
- Avoid common file prefixes in HTML output (#103)

- Added the –execlude-directories argument to exclude directories from the search for symlinks (#87)
- Added branches taken/not taken to HTML (#75)
- Use –object-directory to scan for gcov data files (#72)
- Improved logic for nested makefiles (#135)
- Fixed unexpected semantics with –root argument (#108)
- More careful checks for covered lines (#109)

8.8 3.2 (5 July 2014)

- Adding a test for out of source builds
- Using the starting directory when processing gcov filenames. (#42)
- Making relative paths the default in html output.
- Simplify html bar with coverage is zero.
- Add option for using existing gcov files (#35)
- Fixing –root argument processing (#27)
- Adding logic to cover branches that are ignored (#28)

8.9 3.1 (6 December 2013)

- Change to make the -r/–root options define the root directory for source files.
- Fix to apply the -p option when the –html option is used.
- Adding new option, '–exclude-unreachable-branches' that will exclude branches in certain lines from coverage report.
- Simplifying and standardizing the processing of linked files.
- Adding tests for deeply nested code, and symbolic links.
- Add support for multiple —filter options in same manner as —exclude option.

8.10 3.0 (10 August 2013)

- Adding the '–gcov-executable' option to specify the name/location of the gcov executable. The command line option overrides the environment variable, which overrides the default 'gcov'.
- Adding an empty "<methods/>" block to <classes/> in the XML output: this makes out XML complient with the Cobertura DTD. (#3951)
- Allow the GCOV environment variable to override the default 'gcov' executable. The default is to search the PATH for 'gcov' if the GCOV environment variable is not set. (#3950)
- Adding support for LCOV-style flags for excluding certain lines from coverage analysis. (#3942)
- Setup additional logic to test with Python 2.5.
- Added the –html and –html-details options to generate HTML.

- Sort output for XML to facilitate baseline tests.
- Added error when the –object-directory option specifies a bad directory.
- Added more flexible XML testing, which can ignore XML elements that frequently change (e.g. timestamps).
- Added the '—xml-pretty' option, which is used to generate pretty XML output for the user manual.
- Many documentation updates

8.11 2.4 (13 April 2012)

- New approach to walking the directory tree that is more robust to symbolic links (#3908)
- Normalize all reported path names
  - Normalize using the full absolute path (#3921)
  - Attempt to resolve files referenced through symlinks to a common project-relative path
- Process `gcno` files when there is no corresponding `gcda` file to provide coverage information for unexecuted modules (#3887)
- Windows compatibility fixes
  - Fix for how we parse `source:` file names (#3913)
  - Better handling od EOL indicators (#3920)
- Fix so that gcovr cleans up all `.gcov` files, even those filtered by command line arguments
- Added compatibility with GCC 4.8 (#3918)
- Added a check to warn users who specify an empty `--root` option (see #3917)
- Force `gcov` to run with en_US localization, so the gcovr parser runs correctly on systems with non-English locales (#3898, #3902).
- Segregate warning/error information onto the stderr stream (#3924)
- Miscellaneous (Python 3.x) portability fixes
- Added the master svn revision number as part of the verson identifier

8.12 2.3.1 (6 January 2012)

- Adding support for Python 3.x

8.13 2.3 (11 December 2011)

- Adding the `--gcov-filter` and `--gcov-exclude` options.

8.14 2.2 (10 December 2011)

- Added a test driver for gcovr.
- Improved estimation of the `<sources>` element when using gcovr with filters.
- Added revision and date keywords to gcovr so it is easier to identify what version of the script users are using (especially when they are running a snapshot from trunk).
- Addressed special case mentioned in [comment:ticket:3884:1]: do not truncate the reported file name if the filter does not start matching at the beginning of the string.
- Overhaul of the `--root` / `--filter` logic. This should resolve the issue raised in #3884, along with the more general filter issue raised in [comment:ticket:3884:1]
- Overhaul of gcovr's logic for determining gcc/g++'s original working directory. This resolves issues introduced in the original implementation of `--object-directory` (#3872, #3883).
- Bugfix: gcovr was only including a `<sources>` element in the XML report if the user specified `-r` (#3869)
- Adding timestamp and version attributes to the gcovr XML report (see #3877). It looks like the standard Cobertura output reports number of seconds since the epoch for the timestamp and a doted decimal version string. Now, gcovr reports seconds since the epoch and "`gcovr ``"+``__version__` (e.g. "gcovr 2.2") to differentiate it from a pure Cobertura report.

8.15 2.1 (26 November 2010)

- Added the `--object-directory` option, which allows for a flexible specification of the directory that contains the objects generated by gcov.
- Adding fix to compare the absolute path of a filename to an exclusion pattern.
- Adding error checking when no coverage results are found. The line and branch counts can be zero.
- Adding logic to process the `-o`/`--output` option (#3870).
- Adding patch to scan for lines that look like:

```
creating `foo'
```

  as well as

```
creating 'foo'
```

- Changing the semantics for EOL to be portable for MS Windows.
- Add attributes to xml format so that it could be used by hudson/bamboo with cobertura plug-in.

8.16 2.0 (22 August 2010)

- Initial release as a separate package. Earlier versions of gcovr were managed within the 'fast' Python package.

CHAPTER 9

License

Copyright 2013-2021 the gcovr authors

Copyright 2013 Sandia Corporation. Under the terms of Contract DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains certain rights in this software.

Gcovr is available under the 3-clause BSD License. See LICENSE.txt for full details. See AUTHORS.txt for the full list of contributors.

Gcovr development moved to this repository in September, 2013 from Sandia National Laboratories.

9.1 License Terms

Gcovr is available under the terms of a BSD-3-clause license:

```
Copyright 2013-2021 the gcovr authors
Copyright 2013 Sandia Corporation. Under the terms of Contract DE-AC04-
94AL85000 with Sandia Corporation, the U.S. Government retains certain
rights in this software.

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

* Neither the name of the Sandia National Laboratories nor the names of
```

(continues on next page)

(continued from previous page)

```
its contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

9.2 Acknowledgements

Gcovr is maintained by:

William Hart, John Siirola, and Lukas Atkinson.

The following developers contributed to gcovr (ordered alphabetically):

alex43dm, Andrew Stone, Antonio Quarta, Arvin Schnell, Attie Grande, Bernhard Breinbauer, Carlos Jenkins, Cary Converse, Cezary Gapiński, Christian Taedcke, Dave George, Dom Postorivo, ebmmy, Elektrobit Automotive GmbH, Ensky Lin, Glenn Töws, Grégoire Roussel, goriy, Irfan Adilovic, ja11sop, James Reynolds, Jeremy Fixemer, Jessica Levine, Joachim Kuebart, Joel Klinghed, John Siirola, Jörg Kreuzberger, Jordi Sabater, Kai Blaschke, Kevin Broselge, Kevin Cai, Leon Ma, libPhipp, Lukas Atkinson, Luke Woydziak, Marek Kurdej, Martin Mraz, Matsumoto Taichi, Matthew Stadelman, Matthias Schmieder, Matthieu Darbois, Matthieu Eyraud, Michael Förderer, Michał Pszona, Mikael Salson, Mikk Leini, Nikolaj Schumacher, Oleksiy Pikalo, Phil Clapham, Piotr Dziwinski, Reto Schneider, Richard Kjerstadius, Robert Rosengren, Songmin Li, Steven Myint, Sylvestre Ledru, Tilo Wiedera, trapzero, Will Thompson, William Hart, Zachary J. Fields, Zachary P. Hensley, and possibly others.

The development of Gcovr has been partially supported by Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Index

Symbols

G

P

S