gcovr Documentation

Release 4.2

the gcovr authors

Feb 16, 2020

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:

- default: compact human-readable summaries
- `--html`: HTML summaries
- `--html-details`: HTML report with annotated source files
- `--xml`: machine readable XML reports in Cobertura format
- `--sonarqube`: machine readable XML reports in Sonarqube format
- `--json`: JSON report with source files structure and coverage

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*

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: 3.5+.

The automated tests run on CPython 3.5, CPython 3.7, and PyPy 3.5. 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

Operating System: Linux, Windows, and macOS.

The automated tests run on Ubuntu 16.04 and Windows Server 2012.

Compiler: GCC and Clang.

The automated tests run on GCC 5.

CHAPTER 2

Gcovr User Guide

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:

- default: compact human-readable summaries
- `--html`: HTML summaries
- `--html-details`: HTML report with annotated source files
- `--xml`: machine readable XML reports in Cobertura format
- `--sonarqube`: machine readable XML reports in Sonarqube format
- `--json`: JSON report with source files structure and coverage

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.

The Gcovr Home Page is http://gcovr.com. Automated test results are available through Travis CI and Appveyor. Gcovr is available under the BSD license.

This documentation describes Gcovr 4.2.

This User Guide provides the following sections:

Related documents:

- *Installation*
- *Contributing* (includes instructions for bug reports)
- *Gcovr Cookbook*
- *Frequently Asked Questions*
- *Change Log*
- *License*

2.1 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.

The `gcovr` command can be used to analyze programs compiled with GCC. The following sections illustrate the application of `gcovr` to test coverage of the following program:

```
 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.

2.1.1 Tabular Output of Code Coverage

We compile `example1.cpp` with the GCC compiler as follows:

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

(If you are using CMake, also see *Out-of-Source Builds with CMake*.)

Note that we compile this program without optimization, because optimization may combine lines of code and otherwise change the flow of execution in the program. Additionally, we compile with the `-fprofile-arcs -ftest-coverage -fPIC` compiler options, which add logic to generate output files that can be processed by the `gcov` command.

The compiler generates the `program` executable. When we execute this command:

```
./program
```

the files `example1.gcno` and `example1.gcda` are generated. These files are processed with by `gcov` to generate code coverage statistics. The `gcovr` command calls `gcov` and summarizes these code coverage statistics in various formats. For example:

```
gcovr -r .
```

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%
------------------------------------------------------------------------------
```

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.

The `-r` option specifies the root directory for the files that are being analyzed. This allows `gcovr` to generate a simpler report (without absolute path names), and it allows system header files to be excluded from the analysis.

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

2.1.2 Tabular Output of Branch Coverage

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

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

2.1.3 Cobertura XML Output

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

```
gcovr -r . --xml-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" complexity="0.0" timestamp=
↪"1573053861" version="gcovr 4.2">
  <sources>
    <source>.</source>
  </sources>
  <packages>
    <package name="" line-rate="0.8571428571428571" branch-rate="0.5" complexity="0.0
↪">
      <classes>
        <class name="example_cpp" filename="example.cpp" line-rate="0.8571428571428571
↪" branch-rate="0.5" complexity="0.0">
          <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*

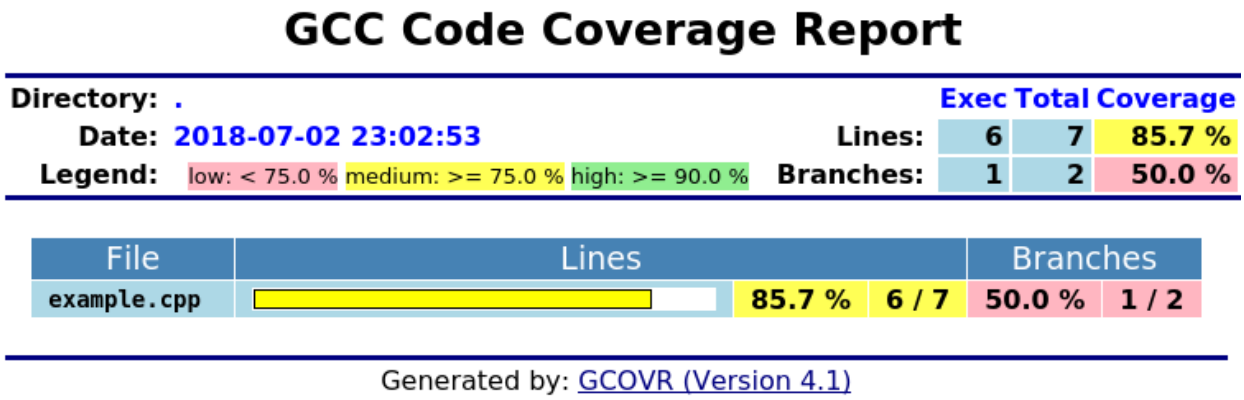
The `--xml` option generates a denser XML output, and the `--xml-pretty` option generates an indented XML output that is easier to read. Note that the XML output contains more information than the tabular summary. The tabular summary shows the percentage of covered lines, while the XML output includes 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.

2.1.4 HTML Output

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

```
gcovr -r . --html -o example-html.html
```

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

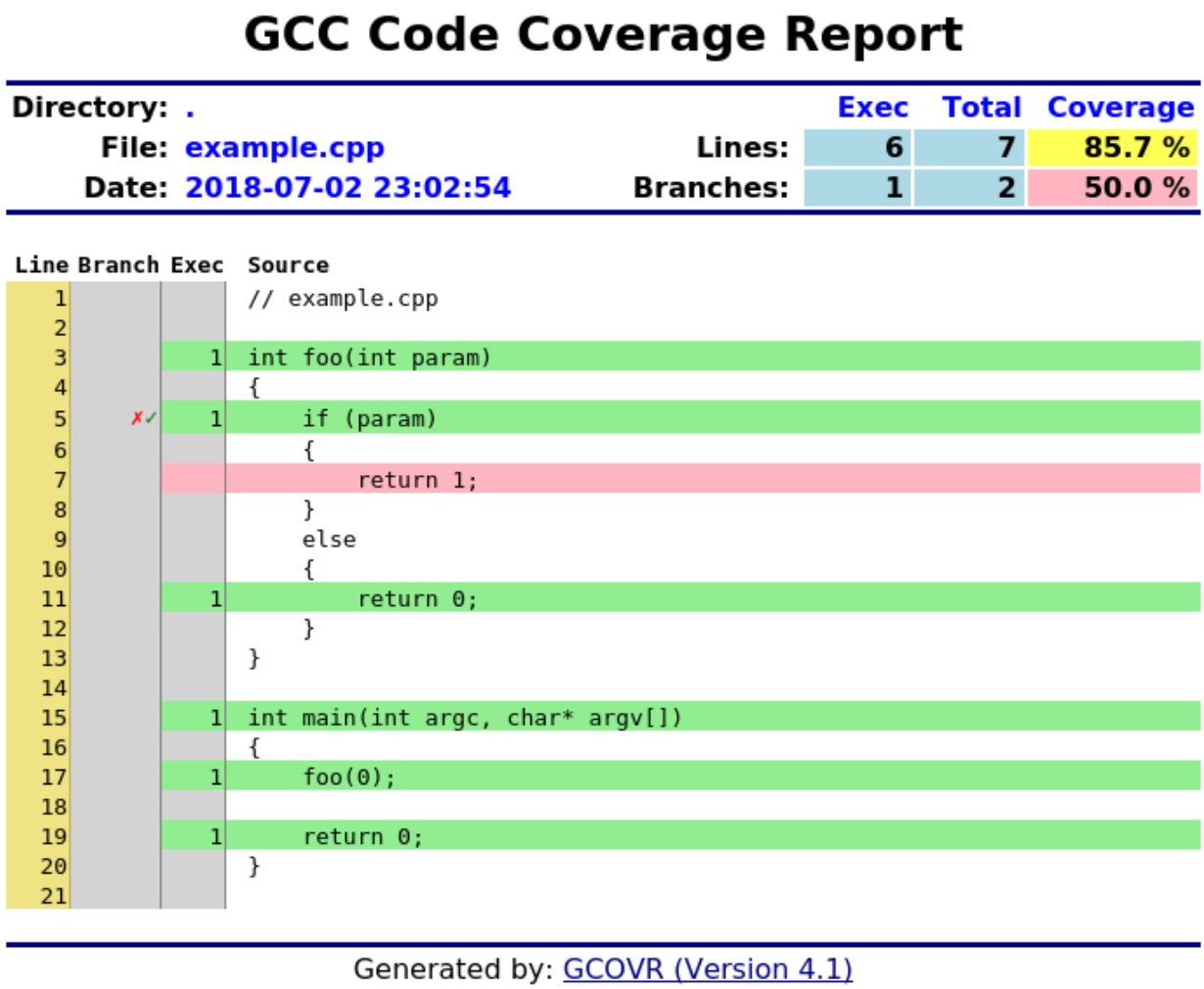


The default behavior of the `--html` option is to generate HTML for a single webpage that summarizes the coverage for all files. The HTML is printed to standard output, but the `-o` (`--output`) option is used to specify a file that stores the HTML output.

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 -r . --html --html-details -o example-html-details.html
```

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



GCC Code Coverage Report

		Exec	Total	Coverage
Directory:	.			
File:	example.cpp	Lines:	6	7	85.7 %
Date:	2018-07-02 23:02:54	Branches:	1	2	50.0 %

Line	Branch	Exec	Source
1			// example.cpp
2			
3		1	int foo(int param)
4			{
5	✗✓	1	if (param)
6			{
7			return 1;
8			}
9			else
10			{
11		1	return 0;
12			}
13			}
14			
15		1	int main(int argc, char* argv[])
16			{
17		1	foo(0);
18			
19		1	return 0;
20			}
21			

Generated by: GCOVR (Version 4.1)

Note that the `--html-details` option can only be used with the `-o` (`--output`) option. For example, if the `--output` option specifies the output file `coverage.html`, then the web pages generated for each file will have names of the form `coverage.<filename>.html`.

2.1.5 Sonarqube XML Output

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

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

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

2.1.6 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.

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

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.

Multiple JSON files can be merged into the coverage data with sum of lines and branches execution

2.1.7 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 --xml*
- *gcovr --html*
- *gcovr --html-details*
- *gcovr --sonarqube*

- *gcovr --json*

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

2.1.8 Combining Tracefiles

You can merge coverage data from multiple runs with *gcovr --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 *gcovr --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
```

2.2 The gcovr Command

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

2.2.1 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.

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.

--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).

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.

-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.

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

--xml-pretty
Pretty-print the XML report. Implies –xml. 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-title <title>
Use TITLE as title for the HTML report. Default is Head.

--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. 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-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.

-s, --print-summary
Print a small report to stdout with line & 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.

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.

-f <filter>, **--filter** <filter>
Keep only source files that match this filter. Can be specified multiple times. 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.

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'.

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

The above *Getting Started* guide illustrates the use of some command line options. *Using Filters* is discussed below.

2.3 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. 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 *--filter* is provided, the *--root* is turned into a default filter. Therefore, files outside of the *--root* directory are excluded.

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

2.3.1 Speeding up coverage data search

The *--exclude-directories* filter is used while searching for raw coverage data (or for existing `.gcov` files when *--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 *--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.

2.3.2 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/
```

(continues on next page)

(continued from previous page)

```
│   └─ 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/'
```

Note: This section discusses symlinks on Unix systems. The behavior under Windows is unclear. If you have more insight, please update this section by submitting a pull request (see our *contributing guide*).

2.4 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 *--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 *--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, *--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, ...

2.5 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 markers are not configurable.

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).

2.6 Acknowledgements

Gcovr is maintained by:

William Hart, John Siirola, and Lukas Atkinson.

The following developers contributed to gcovr (ordered alphabetically):

alex43dm, Andrew Stone, Arvin Schnell, Attie Grande, Bernhard Breinbauer, Carlos Jenkins, Cezary Gapiński, Christian Taedcke, Dave George, Dom Postorivo, goriy, ja11sop, James Reynolds, Jeremy Fixemer, Jessica Levine, Joel Klinghed, John Siirola, Jörg Kreuzberger, Kai Blaschke, Kevin Cai, libPhipp, Lukas Atkinson, Luke Woydziak, Leon Ma, Marek Kurdej, Martin Mraz, Matsumoto Taichi, Matthew Stadelman, Matthias Schmieder, Matthieu Darbois, Michael Förderer, Michał Pszona, Mikael Salson, Mikk Leini, Nikolaj Schumacher, Piotr Dziwinski, Phil Clapham, Richard Kjerstadius, Reto Schneider, Robert Rosengren, Songmin Li, Steven Myint, Sylvestre Ledru, Tilo Wiedera, trapzero, Will Thompson, William Hart, 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.

CHAPTER 3

Gcovr Cookbook

3.1 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"

# 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 -o coverage/index.html
```

3.2 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`:

```
add_compile_options("--coverage")

add_executable(program example.cpp)
target_link_libraries(program gcov)
```

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 $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 4

Frequently Asked Questions

4.1 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.

4.2 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.

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. See also: Gcov and Optimization in the GCC documentation.

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

4.3 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.

CHAPTER 5

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*

5.1 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.

5.2 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

5.3 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:

  ```
  make test
  ```

  (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 Travis CI or Appveyor for your fork, or run the tests with Docker.

  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:

  ```
  make lint

  # or:

  python3 -m flake8 doc gcovr --ignore E501,W503
  ```

  The command `make qa` 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.

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

5.4 How to set up a development environment

For working on gcovr, you will need a supported version of Python 3, and GCC version 5. Other GCC versions are supported by gcovr, but will cause spurious test failures.

- (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 the test requirements:

```
make setup-dev  # install all test + doc dependencies

# or:

pip install -e .
pip install -r requirements.txt
```

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

  Run the tests to verify that everything works (see *Test suite*).

- (Optional) Install documentation requirements:

```
# would be already done by `make setup-dev`
pip install -r doc/requirements.txt
```

  See `doc/README.txt` for details on working with the documentation.

- (Optional) Activate Travis and Appveyor for your forked GitHub 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: `.travis.yml` and `admin/Dockerfile.qa`
- for Windows: `appveyor.yml`

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. To run the tests, please make sure that the `make` and `cmake` from your MinGW distribution are in the system `PATH`.

If setting up a local toolchain is too complicated, you can also run the tests in a Docker container (see *Test suite*).

5.5 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
`/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.html_generator` and respective modules.

5.6 Test suite

The QA process (`make qa`) consists of multiple parts:

- linting (`make lint`)
- tests (`make test`)
  - unit tests in `gcovr/tests`
  - integration tests in `gcovr/tests`
  - documentation examples in `doc/examples`
- documentation build (`make doc`)

The tests are in the `gcovr/tests` directory. You can run the tests with `make test` or `python3 -m pytest gcovr`.

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.

Each project in the corpus contains a `Makefile` and a `reference` directory. 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. Each project is once per output format (txt, html, xml, json, sonarqube, ...).

Because the tests are a bit slow, you can limit the tests to a specific test file, example project, or output format. For example:

```
# run only XML tests
python3 -m pytest -k xml

# run the simple1 tests
python3 -m pytest -k simple1

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

(continues on next page)

(continued from previous page)

```
# set filters using the Makefile:
make test TEST_OPTS="-k 'xml and simple1'"
```

To see all tests, run pytest in `-v` verbose mode. To see which tests would be run, add the `--collect-only` option.

The tests currently assume that you are using GCC 5.

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

```
make docker-qa-build

# or:

docker build --tag gcovr-qa --file admin/Dockerfile.qa .
```

Then, run the container, which executes `make qa` within the container:

```
make docker-qa

# or:

docker run --rm -v `pwd`:/gcovr gcovr-qa
```

5.7 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 6

Change Log

`gcovr` Release History and Change Log

6.1 Next Release

Breaking changes:

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

Documentation:

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

6.2 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)
- *Combining tracefiles* 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.

6.3 4.1 (2 July 2018)

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

6.4 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)

6.5 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)

6.6 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)

6.7 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)

6.8 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.

6.9 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

6.10 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

6.11 2.3.1 (6 January 2012)

- Adding support for Python 3.x

6.12 2.3 (11 December 2011)

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

6.13 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.

6.14 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.

6.15 2.0 (22 August 2010)

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

CHAPTER 7

License

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

Index

Symbols

G

P

S