Invoking Gcov (Using the GNU Compiler Collection (GCC))

10.2 Invoking `gcov` ¶

`gcov` accepts the following options:

`-a`
`--all-blocks`

> Write individual execution counts for every basic block. Normally gcov outputs execution counts only for the main blocks of a line. With this option you can determine if blocks within a single line are not being executed.

`-b`
`--branch-probabilities`

> Write branch frequencies to the output file, and write branch summary info to the standard output. This option allows you to see how often each branch in your program was taken. Unconditional branches will not be shown, unless the `-u` option is given.

`-c`
`--branch-counts`

> Write branch frequencies as the number of branches taken, rather than the percentage of branches taken.

`-g`
`--conditions`

> Write condition coverage to the output file, and write condition summary info to the standard output. This option allows you to see if the conditions in your program at least once had an independent effect on the outcome of the boolean expression (modified condition/decision coverage). This requires you to compile the source with `-fcondition-coverage`.

`-d`
`--display-progress`

> Display the progress on the standard output.

`-f`
`--function-summaries`

> Output summaries for each function in addition to the file level summary.

`-h`
`--help`

> Display help about using `gcov` (on the standard output), and exit without doing any further processing.

`-j`

`--json-format`

Output gcov file in an easy-to-parse JSON intermediate format which does not require source code for generation. The JSON file is compressed with gzip compression algorithm and the files have `.gcov.json.gz` extension.

Structure of the JSON is following:

```
{
  "current_working_directory": "foo/bar",
  "data_file": "a.out",
  "format_version": "2",
  "gcc_version": "11.1.1 20210510"
  "files": ["$file"]
}
```

Fields of the root element have following semantics:

- *current_working_directory*: working directory where a compilation unit was compiled
- *data_file*: name of the data file (GCDA)
- *format_version*: semantic version of the format

  Changes in version *2*:

  - *calls*: information about function calls is added
- *gcc_version*: version of the GCC compiler

Each *file* has the following form:

```
{
  "file": "a.c",
  "functions": ["$function"],
  "lines": ["$line"]
}
```

Fields of the *file* element have following semantics:

- *file_name*: name of the source file

Each *function* has the following form:

```
{
  "blocks": 2,
  "blocks_executed": 2,
  "demangled_name": "foo",
  "end_column": 1,
  "end_line": 4,
  "execution_count": 1,
  "name": "foo",
  "start_column": 5,
  "start_line": 1
}
```

Fields of the *function* element have following semantics:

- *blocks*: number of blocks that are in the function
- *blocks_executed*: number of executed blocks of the function
- *demangled_name*: demangled name of the function
- *end_column*: column in the source file where the function ends
- *end_line*: line in the source file where the function ends

- *execution_count*: number of executions of the function
- *name*: name of the function
- *start_column*: column in the source file where the function begins
- *start_line*: line in the source file where the function begins

Note that line numbers and column numbers number from 1. In the current implementation, *start_line* and *start_column* do not include any template parameters and the leading return type but that this is likely to be fixed in the future.

Each *line* has the following form:

```
{
  "block_ids": ["$block_id"],
  "branches": ["$branch"],
  "calls": ["$call"],
  "count": 2,
  "conditions": ["$condition"],
  "line_number": 15,
  "unexecuted_block": false,
  "function_name": "foo",
}
```

Branches and calls are present only with *-b* option. Fields of the *line* element have following semantics:

- *block_ids*: IDs of basic blocks that belong to the line
- *count*: number of executions of the line
- *line_number*: line number
- *unexecuted_block*: flag whether the line contains an unexecuted block (not all statements on the line are executed)
- *function_name*: a name of a function this *line* belongs to (for a line with an inlined statements can be not set)

Each *branch* has the following form:

```
{
  "count": 11,
  "destination_block_id": 17,
  "fallthrough": true,
  "source_block_id": 13,
  "throw": false
}
```

Fields of the *branch* element have following semantics:

- *count*: number of executions of the branch
- *fallthrough*: true when the branch is a fall through branch
- *throw*: true when the branch is an exceptional branch
- *isource_block_id*: ID of the basic block where this branch happens
- *destination_block_id*: ID of the basic block this branch jumps to

Each *call* has the following form:

```
{
  "destination_block_id": 1,
  "returned": 11,
  "source_block_id": 13
}
```

Fields of the *call* element have following semantics:

- *returned*: number of times a function call returned (call count is equal to *line::count*)
- *isource_block_id*: ID of the basic block where this call happens
- *destination_block_id*: ID of the basic block this calls continues after return

Each *condition* has the following form:

```
{
  "count": 4,
  "covered": 2,
  "not_covered_false": [],
  "not_covered_true": [0, 1],
}
```

Fields of the *condition* element have following semantics:

- *count*: number of condition outcomes in this expression
- *covered*: number of covered condition outcomes in this expression
- *not_covered_true*: terms, by index, not seen as true in this expression
- *not_covered_false*: terms, by index, not seen as false in this expression

`-H`
`--human-readable`

Write counts in human readable format (like 24.6k).

`-k`
`--use-colors`

Use colors for lines of code that have zero coverage. We use red color for non-exceptional lines and cyan for exceptional. Same colors are used for basic blocks with `-a` option.

`-l`
`--long-file-names`

Create long file names for included source files. For example, if the header file `x.h` contains code, and was included in the file `a.c`, then running `gcov` on the file `a.c` will produce an output file called `a.c##x.h.gcov` instead of `x.h.gcov`. This can be useful if `x.h` is included in multiple source files and you want to see the individual contributions. If you use the '`-p`' option, both the including and included file names will be complete path names.

`-m`
`--demangled-names`

Display demangled function names in output. The default is to show mangled function names.

`-n`
`--no-output`

Do not create the `gcov` output file.

`-o` *directory|file*
`--object-directory` *directory*
`--object-file` *file*

Specify either the directory containing the gcov data files, or the object path name. The `.gcno`, and `.gcda` data files are searched for using this option. If a directory is specified, the data files are in that directory and named after the input file name, without its extension. If a file is specified here, the data files are named after that file, without its extension.

`-p`
`--preserve-paths`

Preserve complete path information in the names of generated `.gcov` files. Without this option, just the filename component is used. With this option, all directories are used, with '`/`' characters translated to '`#`' characters, `.` directory components removed and unremoveable `..` components renamed to '`^`'. This is useful if sourcefiles are in several different directories.

`-q`
`--use-hotness-colors`

Emit perf-like colored output for hot lines. Legend of the color scale is printed at the very beginning of the output file.

`-r`
`--relative-only`

Only output information about source files with a relative pathname (after source prefix elision). Absolute paths are usually system header files and coverage of any inline functions therein is normally uninteresting.

`-s` *`directory`*
`--source-prefix` *`directory`*

A prefix for source file names to remove when generating the output coverage files. This option is useful when building in a separate directory, and the pathname to the source directory is not wanted when determining the output file names. Note that this prefix detection is applied before determining whether the source file is absolute.

`-t`
`--stdout`

Output to standard output instead of output files.

`-u`
`--unconditional-branches`

When branch probabilities are given, include those of unconditional branches. Unconditional branches are normally not interesting.

`-v`
`--version`

Display the `gcov` version number (on the standard output), and exit without doing any further processing.

`-w`
`--verbose`

Print verbose informations related to basic blocks and arcs.

`-x`
`--hash-filenames`

When using *–preserve-paths*, gcov uses the full pathname of the source files to create an output filename. This can lead to long filenames that can overflow filesystem limits. This option creates names of the form *`source-file`*`##`*`md5`*`.gcov`, where the *source-file* component is

the final filename part and the *md5* component is calculated from the full mangled name that would have been used otherwise. The option is an alternative to the *–preserve-paths* on systems which have a filesystem limit.

`gcov` should be run with the current directory the same as that when you invoked the compiler. Otherwise it will not be able to locate the source files. `gcov` produces files called *`mangledname`*`.gcov` in the current directory. These contain the coverage information of the source file they correspond to. One `.gcov` file is produced for each source (or header) file containing code, which was compiled to produce the data files. The *mangledname* part of the output file name is usually simply the source file name, but can be something more complicated if the '`-l`' or '`-p`' options are given. Refer to those options for details.

If you invoke `gcov` with multiple input files, the contributions from each input file are summed. Typically you would invoke it with the same list of files as the final link of your executable.

The `.gcov` files contain the ':' separated fields along with program source code. The format is

`execution_count``:`*`line_number`*`:`*`source line text`*

Additional block information may succeed each line, when requested by command line option. The *execution_count* is '`-`' for lines containing no code. Unexecuted lines are marked '`#####`' or '`=====`', depending on whether they are reachable by non-exceptional paths or only exceptional paths such as C++ exception handlers, respectively. Given the '`-a`' option, unexecuted blocks are marked '`$$$$$`' or '`%%%%%`', depending on whether a basic block is reachable via non-exceptional or exceptional paths. Executed basic blocks having a statement with zero *execution_count* end with '`*`' character and are colored with magenta color with the `-k` option. This functionality is not supported in Ada.

Note that GCC can completely remove the bodies of functions that are not needed – for instance if they are inlined everywhere. Such functions are marked with '`-`', which can be confusing. Use the `-fkeep-inline-functions` and `-fkeep-static-functions` options to retain these functions and allow gcov to properly show their *execution_count*.

Some lines of information at the start have *line_number* of zero. These preamble lines are of the form

The ordering and number of these preamble lines will be augmented as `gcov` development progresses — do not rely on them remaining unchanged. Use *tag* to locate a particular preamble line.

The additional block information is of the form

The *information* is human readable, but designed to be simple enough for machine parsing too.

When printing percentages, 0% and 100% are only printed when the values are *exactly* 0% and 100% respectively. Other values which would conventionally be rounded to 0% or 100% are instead printed as the nearest non-boundary value.

When using `gcov`, you must first compile your program with a special GCC option '`--coverage`'. This tells the compiler to generate additional information needed by gcov (basically a flow graph of the program) and also includes additional code in the object files for generating the extra profiling information needed by gcov. These additional files are placed in the directory where the object file is located.

Running the program will cause profile output to be generated. For each source file compiled with `-fprofile-arcs`, an accompanying `.gcda` file will be placed in the object file directory.

Running `gcov` with your program's source file names as arguments will now produce a listing of

the code along with frequency of execution for each line. For example, if your program is called `tmp.cpp`, this is what you see when you use the basic `gcov` facility:

```
$ g++ --coverage tmp.cpp -c
$ g++ --coverage tmp.o
$ a.out
$ gcov tmp.cpp -m
File 'tmp.cpp'
Lines executed:92.86% of 14
Creating 'tmp.cpp.gcov'
```

The file `tmp.cpp.gcov` contains output from `gcov`. Here is a sample:

```
        -:    0:Source:tmp.cpp
        -:    0:Working directory:/home/gcc/testcase
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:template<class T>
        -:    4:class Foo
        -:    5:{
        -:    6:  public:
       1*:    7:  Foo(): b (1000) {}
------------------
Foo<char>::Foo():
    #####:    7:  Foo(): b (1000) {}
------------------
Foo<int>::Foo():
        1:    7:  Foo(): b (1000) {}
------------------
       2*:    8:  void inc () { b++; }
------------------
Foo<char>::inc():
    #####:    8:  void inc () { b++; }
------------------
Foo<int>::inc():
        2:    8:  void inc () { b++; }
------------------
        -:    9:
        -:   10:  private:
        -:   11:  int b;
        -:   12:};
        -:   13:
        -:   14:template class Foo<int>;
        -:   15:template class Foo<char>;
        -:   16:
        -:   17:int
        1:   18:main (void)
        -:   19:{
        -:   20:  int i, total;
        1:   21:  Foo<int> counter;
        -:   22:
        1:   23:  counter.inc();
        1:   24:  counter.inc();
        1:   25:  total = 0;
        -:   26:
       11:   27:  for (i = 0; i < 10; i++)
       10:   28:    total += i;
        -:   29:
       1*:   30:  int v = total > 100 ? 1 : 2;
        -:   31:
        1:   32:  if (total != 45)
    #####:   33:    printf ("Failure\n");
```

```
        -:   34:  else
        1:   35:    printf ("Success\n");
        1:   36:  return 0;
        -:   37:}
```

Note that line 7 is shown in the report multiple times. First occurrence presents total number of execution of the line and the next two belong to instances of class Foo constructors. As you can also see, line 30 contains some unexecuted basic blocks and thus execution count has asterisk symbol.

When you use the `-a` option, you will get individual block counts, and the output looks like this:

```
        -:    0:Source:tmp.cpp
        -:    0:Working directory:/home/gcc/testcase
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:template<class T>
        -:    4:class Foo
        -:    5:{
        -:    6:  public:
       1*:    7:  Foo(): b (1000) {}
------------------
Foo<char>::Foo():
    #####:    7:  Foo(): b (1000) {}
------------------
Foo<int>::Foo():
        1:    7:  Foo(): b (1000) {}
------------------
       2*:    8:  void inc () { b++; }
------------------
Foo<char>::inc():
    #####:    8:  void inc () { b++; }
------------------
Foo<int>::inc():
        2:    8:  void inc () { b++; }
------------------
        -:    9:
        -:   10:  private:
        -:   11:  int b;
        -:   12:};
        -:   13:
        -:   14:template class Foo<int>;
        -:   15:template class Foo<char>;
        -:   16:
        -:   17:int
        1:   18:main (void)
        -:   19:{
        -:   20:  int i, total;
        1:   21:  Foo<int> counter;
        1:   21-block  0
        -:   22:
        1:   23:  counter.inc();
        1:   23-block  0
        1:   24:  counter.inc();
        1:   24-block  0
        1:   25:  total = 0;
        -:   26:
       11:   27:  for (i = 0; i < 10; i++)
        1:   27-block  0
       11:   27-block  1
       10:   28:    total += i;
```

```
       10:   28-block  0
        -:   29:
       1*:   30:  int v = total > 100 ? 1 : 2;
        1:   30-block  0
    %%%%%:   30-block  1
        1:   30-block  2
        -:   31:
        1:   32:  if (total != 45)
        1:   32-block  0
    #####:   33:    printf ("Failure\n");
    %%%%%:   33-block  0
        -:   34:  else
        1:   35:    printf ("Success\n");
        1:   35-block  0
        1:   36:  return 0;
        1:   36-block  0
        -:   37:}
```

In this mode, each basic block is only shown on one line – the last line of the block. A multi-line block will only contribute to the execution count of that last line, and other lines will not be shown to contain code, unless previous blocks end on those lines. The total execution count of a line is shown and subsequent lines show the execution counts for individual blocks that end on that line. After each block, the branch and call counts of the block will be shown, if the `-b` option is given.

Because of the way GCC instruments calls, a call count can be shown after a line with no individual blocks. As you can see, line 33 contains a basic block that was not executed.

When you use the `-b` option, your output looks like this:

```
        -:    0:Source:tmp.cpp
        -:    0:Working directory:/home/gcc/testcase
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:template<class T>
        -:    4:class Foo
        -:    5:{
        -:    6:  public:
       1*:    7:  Foo(): b (1000) {}
------------------
Foo<char>::Foo():
function Foo<char>::Foo() called 0 returned 0% blocks executed 0%
    #####:    7:  Foo(): b (1000) {}
------------------
Foo<int>::Foo():
function Foo<int>::Foo() called 1 returned 100% blocks executed 100%
        1:    7:  Foo(): b (1000) {}
------------------
       2*:    8:  void inc () { b++; }
------------------
Foo<char>::inc():
function Foo<char>::inc() called 0 returned 0% blocks executed 0%
    #####:    8:  void inc () { b++; }
------------------
Foo<int>::inc():
function Foo<int>::inc() called 2 returned 100% blocks executed 100%
        2:    8:  void inc () { b++; }
------------------
        -:    9:
        -:   10:  private:
        -:   11:  int b;
```

```
        -:   12:};
        -:   13:
        -:   14:template class Foo<int>;
        -:   15:template class Foo<char>;
        -:   16:
        -:   17:int
function main called 1 returned 100% blocks executed 81%
        1:   18:main (void)
        -:   19:{
        -:   20:  int i, total;
        1:   21:  Foo<int> counter;
call    0 returned 100%
branch  1 taken 100% (fallthrough)
branch  2 taken 0% (throw)
        -:   22:
        1:   23:  counter.inc();
call    0 returned 100%
branch  1 taken 100% (fallthrough)
branch  2 taken 0% (throw)
        1:   24:  counter.inc();
call    0 returned 100%
branch  1 taken 100% (fallthrough)
branch  2 taken 0% (throw)
        1:   25:  total = 0;
        -:   26:
       11:   27:  for (i = 0; i < 10; i++)
branch  0 taken 91% (fallthrough)
branch  1 taken 9%
       10:   28:    total += i;
        -:   29:
       1*:   30:  int v = total > 100 ? 1 : 2;
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
        -:   31:
        1:   32:  if (total != 45)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:   33:    printf ("Failure\n");
call    0 never executed
branch  1 never executed
branch  2 never executed
        -:   34:  else
        1:   35:    printf ("Success\n");
call    0 returned 100%
branch  1 taken 100% (fallthrough)
branch  2 taken 0% (throw)
        1:   36:  return 0;
        -:   37:}
```

For each function, a line is printed showing how many times the function is called, how many times it returns and what percentage of the function's blocks were executed.

For each basic block, a line is printed after the last line of the basic block describing the branch or call that ends the basic block. There can be multiple branches and calls listed for a single source line if there are multiple basic blocks that end on that line. In this case, the branches and calls are each given a number. There is no simple way to map these branches and calls back to source constructs. In general, though, the lowest numbered branch or call will correspond to the leftmost construct on the source line.

For a branch, if it was executed at least once, then a percentage indicating the number of times the branch was taken divided by the number of times the branch was executed will be printed. Otherwise, the message "never executed" is printed.

For a call, if it was executed at least once, then a percentage indicating the number of times the

call returned divided by the number of times the call was executed will be printed. This will usually be 100%, but may be less for functions that call `exit` or `longjmp`, and thus may not return every time they are called.

When you use the `-g` option, your output looks like this:

```
$ gcov -t -m -g tmp
        -:    0:Source:tmp.cpp
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int
        1:    4:main (void)
        -:    5:{
        -:    6:  int i, total;
        1:    7:  total = 0;
        -:    8:
       11:    9:  for (i = 0; i < 10; i++)
condition outcomes covered 2/2
       10:   10:    total += i;
        -:   11:
       1*:   12:  int v = total > 100 ? 1 : 2;
condition outcomes covered 1/2
condition  0 not covered (true)
        -:   13:
       1*:   14:  if (total != 45 && v == 1)
condition outcomes covered 1/4
condition  0 not covered (true)
condition  1 not covered (true false)
    #####:   15:    printf ("Failure\n");
        -:   16:  else
        1:   17:    printf ("Success\n");
        1:   18:  return 0;
        -:   19:}
```

For every condition the number of taken and total outcomes are printed, and if there are uncovered outcomes a line will be printed for each condition showing the uncovered outcome in parentheses. Conditions are identified by their index – index 0 is the left-most condition. In `a || (b && c)`, *a* is condition 0, *b* condition 1, and *c* condition 2.

An outcome is considered covered if it has an independent effect on the decision, also known as masking MC/DC (Modified Condition/Decision Coverage). In this example the decision evaluates to true and *a* is evaluated, but not covered. This is because *a* cannot affect the decision independently – both *a* and *b* must change value for the decision to change.

```
$ gcov -t -m -g tmp
        -:    0:Source:tmp.c
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    1:#include <stdio.h>
        -:    2:
        1:    3:int main()
        -:    4:{
        1:    5:  int a = 1;
        1:    6:  int b = 0;
        -:    7:
        1:    8:  if (a && b)
condition outcomes covered 1/4
condition  0 not covered (true false)
condition  1 not covered (true)
    #####:    9:    printf ("Success!\n");
```

```
-:   10:  else
1:   11:    printf ("Failure!\n");
-:   12:}
```

The execution counts are cumulative. If the example program were executed again without removing the `.gcda` file, the count for the number of times each line in the source was executed would be added to the results of the previous run(s). This is potentially useful in several ways. For example, it could be used to accumulate data over a number of program runs as part of a test verification suite, or to provide more accurate long-term information over a large number of program runs.

The data in the `.gcda` files is saved immediately before the program exits. For each source file compiled with `-fprofile-arcs`, the profiling code first attempts to read in an existing `.gcda` file; if the file doesn't match the executable (differing number of basic block counts) it will ignore the contents of the file. It then adds in the new execution counts and finally writes the data to the file.