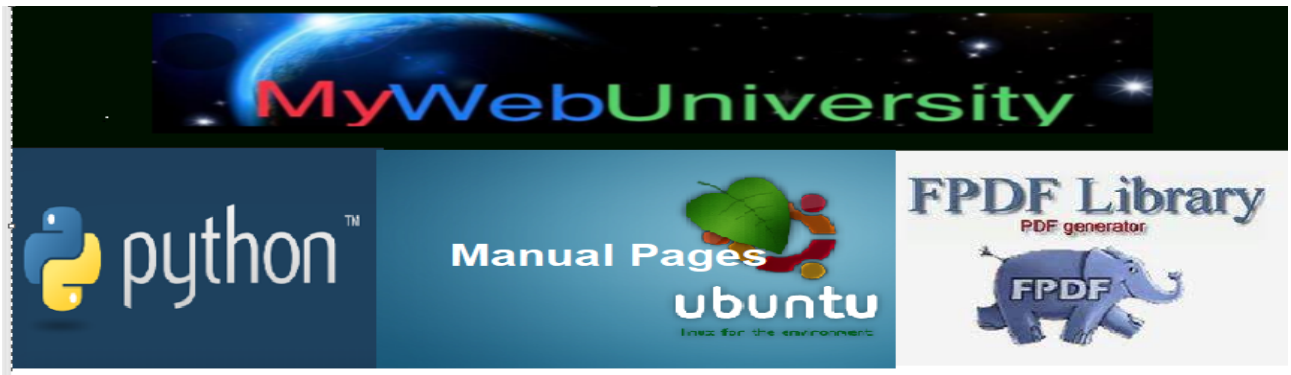

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'gcov.1'

$ man gcov.1

GCOV(1) GNU GCOV(1)

NAME

gcov - coverage testing tool

SYNOPSIS

gcov [-v|--version] [-h|--help]

[-a|--all-blocks]

[-b|--branch-probabilities]

[-c|--branch-counts]

[-g|--conditions]

[-e|--prime-paths]

[--prime-paths-lines[=type]]

[--prime-paths-source[=type]]

[-d|--display-progress]

[-f|--function-summaries]

[--include regex]

[--exclude regex]

[-j|--json-format]

[-H|--human-readable]

[-k|--use-colors]

[-l|--long-file-names]

[-m|--demangled-names]

[-M|--filter-on-demangled]

[-n|--no-output]

[-o|--object-directory directory|file]

[-p|--preserve-paths]

[-q|--use-hotness-colors]

[-r|--relative-only]

[-s|--source-prefix directory]

[-t|--stdout]

[-u|--unconditional-branches]

[-x|--hash-filenames]

files

DESCRIPTION

gcov is a test coverage program. Use it in concert with GCC to analyze your programs to help create more efficient, faster running code and to discover untested parts of your program. You can use gcov as a profiling tool to help discover where your optimization efforts will best affect your code. You can also use gcov along with the other profiling tool, gprof, to assess which parts of your code use the greatest amount of computing time.

Profiling tools help you analyze your code's performance. Using a profiler such as gcov or gprof, you can find out some basic performance statistics, such as:

* how often each line of code executes
* what lines of code are actually executed
* how much computing time each section of code uses

Once you know these things about how your code works when compiled, you can look at each module to see which modules should be optimized. gcov helps you determine where to work on optimization.

Software developers also use coverage testing in concert with testsuites, to make sure software is actually good enough for a release. Testsuites can verify that a program works as expected; a coverage program tests to see how much of the program is exercised by the testsuite. Developers can then determine what kinds of test cases need to be added to the testsuites to create both better testing and a better final product.

You should compile your code without optimization if you plan to use gcov because the optimization, by combining some lines of code into one function, may not give you as much information as you need to look for `hot spots' where the code is using a great deal of computer time. Likewise, because gcov accumulates statistics by line (at the lowest resolution), it works best with a programming style that places only one statement on each

line. If you use complicated macros that expand to loops or to other control structures, the statistics are less helpful---they only report on the line where the macro call appears. If your complex macros behave like functions, you can replace them with inline functions to solve this problem.

gcov creates a logfile called sourcefile.gcov which indicates how many times each line of a source file sourcefile.c has executed. You can use these logfiles along with gprof to aid in fine-tuning the performance of your programs. gprof gives timing information you can use along with the information you get from gcov.

gcov works only on code compiled with GCC. It is not compatible with any other profiling or test coverage mechanism.

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.

-e

--prime-paths

Write path coverage to the output file, and write path summary info to the standard output. This option allows you to see how many prime paths were taken at least once. A path is a sequence of basic blocks. A path is simple if it has no repeated blocks (no loops) except maybe the first and last block, and prime if it is a simple path of maximal length. For the regular output this option only includes the number of paths covered. For more fine grained information on paths you can use --prime-paths-lines or --prime-paths-source. With --json-format all path details are included in the output. This requires you to compile the source with -fpath-coverage.

--prime-paths-lines [=type]

Write path coverage to the output file, and write path summary info to the standard output. This option allows you to see how many prime paths were taken at least once, and dense report on the covered or uncovered paths and how to cover them. This mode is useful for automated reporting and progress tracking. type may be omitted, or one of:

* uncovered - Include the uncovered (not taken) paths. This is the default.
* covered - Include the covered (taken) paths.
* both - Include all paths. This is equivalent to using both covered and uncovered.

This is an example of --prime-paths-lines output:

```
paths covered 12 of 15
path  2 not covered: lines 8 8(false) 11(true) 11 13(true) 13(true) 14 17
path  3 not covered: lines 8 8(false) 11(true) 11 13(true) 13(false) 16 17
path  4 not covered: lines 8 8(false) 11(true) 11 13(false) 16 17
```

This means to cover path 2 you must run lines 8, 11, 13, 14, and 17, evaluting the decision at 8 false and the decisions at 11 and 13 to "false".

--prime-paths-source [=type]

Write path coverage to the output file, and write path summary info to the standard output. This option allows you to see how many prime paths were taken at least once, and detailed report on the uncovered paths an how to cover them. This mode is useful for understanding paths and interactions between sections of your program. type may be omitted, or one of:

* uncovered - Include the uncovered (not taken) paths. This is the default.
* covered - Include the covered (taken) paths.
* both - Include all paths. This is equivalent to using both covered and uncovered.

This is an example of --prime-paths-source output:

```
path 10 not covered:
BB  3:          8:  for (i = 0; i < 10; i++)
BB  3:          9:    total += i;
BB  4: (false)   8:  for (i = 0; i < 10; i++)
BB  5: (true)   11:  int v = total > 100 ? 1 : 2;
BB  6:         11:  int v = total > 100 ? 1 : 2;
BB  8: (false)  13:  if (total != 45 && v == 1)
BB 11:          16:    printf ("Success\n");
BB 12:          17:  return 0;
```

The first (BB) column is the sequence of basic blocks (see -w). The middle column (true/false) is the decision for that line. The third column is the line number. The fourth column is the line itself. These lines must be run in this order to cover path 10.

-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.

--include regex

Include functions matching regex. This option makes gcov only report on functions that match the extended regular expression regex. This flag can be combined with --exclude. If a function matches both includes and excludes, the last include/exclude applies. By default gcov reports on all functions, but if a --include is used then only functions matching the include will be reported.

--exclude regex

Exclude functions matching regex. This option makes gcov not report on functions that match the extended regular expression regex. This flag can be combined with --include. If a function matches both includes and excludes, the last include/exclude applies. By default gcov reports on all functions, and if --exclude is used then functions matching it will be omitted.

-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.

-M

--filter-on-demangled

Make --include and --exclude match demangled names. This does only affects the matching and does not imply --demangled-names, but it can safely be combined with it.

-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

-:0:<tag>:<value>

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

<tag> <information>

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:}
```

When you compile with --coverage -fpath-coverage and use the option -e your output looks like this:

```
    $ gcov -t -e tmp
        -:    0:Source:tmp.cpp
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    1:#include <stdio.h>
        -:    2:
    paths covered 4 of 15
        1:    3:int main ()
        -:    4:{
        -:    5:  int i, total;
        1:    6:  total = 0;
```

```
        -:    7:
       11:    8:  for (i = 0; i < 10; i++)
       10:    9:    total += i;
        -:   10:
       1*:   11:  int v = total > 100 ? 1 : 2;
        -:   12:
       1*:   13:  if (total != 45 && v == 1)
    #####:   14:    printf ("Failure\n");
        -:   15:  else
        1:   16:    printf ("Success\n");
        1:   17:  return 0;
        -:   18:}
```

This output is useful to figure out roughly where coverage is missing and testing how different inputs change the coverage. The --prime-paths-source is a useful tool for understanding paths.

```
$ gcov -t --prime-paths-source tmp
        -:    0:Source:tmp.cpp
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    1:#include <stdio.h>
        -:    2:
paths covered 4 of 15
path 1:
BB  2:          3:int main ()
BB  2:          6:  total = 0;
BB  2:          8:  for (i = 0; i < 10; i++)
BB  4: (false)   8:  for (i = 0; i < 10; i++)
BB  5: (true)   11:  int v = total > 100 ? 1 : 2;
BB  6:          11:  int v = total > 100 ? 1 : 2;
BB  8: (true)   13:  if (total != 45 && v == 1)
BB  9: (true)   13:  if (total != 45 && v == 1)
BB 10:          14:    printf ("Failure\n");
```

BB 12: 17: return 0;

In this mode, gcov will print details on the missing paths. The first column lists the sequence of basic blocks (BB). The second column is the decision to take at that line if there is one. The final columns are the line number and the line itself. This is useful for understanding the paths, in particular those that are hard to cover or even unreachable. Lines may be repeated, for example the "for" loop, if the same line is a part of multiple basic blocks. This mode is intended for humans and good at understanding what code is exercised under testing or for given inputs. This output is quite verbose, and for focusing on specific functions it can be combined with the filters --include and --exclude.

A denser output is available with --prime-paths-lines, which looks like this:

```
-:    0:Source:tmp.cpp
-:    0:Graph:tmp.gcno
-:    0:Data:tmp.gcda
-:    0:Runs:1
-:    1:#include <stdio.h>
-:    2:
paths covered 4 of 15
path  1 not covered: lines 8 8(false) 11(true) 11 13(true) 13(true) 14 17
path  2 not covered: lines 8 8(false) 11(true) 11 13(true) 13(false) 16 17
path  3 not covered: lines 8 8(false) 11(true) 11 13(false) 16 17
path  4 not covered: lines 8 8(false) 11(false) 11 13(true) 13(true) 14 17
path  5 not covered: lines 8 8(false) 11(false) 11 13(true) 13(false) 16 17
path  6 not covered: lines 8 8(false) 11(false) 11 13(false) 16 17
path  8 not covered: lines 9 8(false) 11(true) 11 13(true) 13(true) 14 17
path  9 not covered: lines 9 8(false) 11(true) 11 13(true) 13(false) 16 17
path 10 not covered: lines 9 8(false) 11(true) 11 13(false) 16 17
path 11 not covered: lines 9 8(false) 11(false) 11 13(true) 13(true) 14 17
path 12 not covered: lines 9 8(false) 11(false) 11 13(true) 13(false) 16 17
    1:    3:int main ()
    -:    4:{
```

In this mode, every missing path is expanded using the lines and decisions like --prime-paths-source but printed on a single line. This mode provides a good overview over the paths and for tracking how different tests and inputs exercises the code.

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.

You can report on a subset of functions by using --include and --exclude. This is very useful when combined with --stdout trying to understand behavior and coverage for a particular function by running a test, looking at gcov output, testing another input, and running gcov again.

```
$ gcov -m --stdout --include inc tmp
        -:    0:Source:tmp.cpp
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
       2*:    8:  void inc () { b++; }
------------------
Foo<char>::inc():
    #####:    8:  void inc () { b++; }
------------------
Foo<int>::inc():
        2:    8:  void inc () { b++; }
------------------
```

gcov will match on mangled names by default, which you can control with the -M flag. Note that matching and reporting are independent, so you can match on mangled names while printing demangled names, and vice versa. To report on the "int" instantiation of "Foo" matching on mangled and demangled names:

```
$ gcov -t -m -M tmp --include 'Foo<int>'
```

```
        -:    0:Source:tmp.cpp
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        1:    7:  Foo(): b (1000) {}
        2:    8:  void inc () { b++; }
$ gcov -t -m tmp --include 'FooIi'
        -:    0:Source:tmp.cpp
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        1:    7:  Foo(): b (1000) {}
        2:    8:  void inc () { b++; }
```

The arguments to --include and --exclude are extended regular expressions (like grep -E), so the pattern "in.?" matches both "inc" and "main". If used with -M then all "int" instantiations of "Foo" would match too. --include and --exclude can be used multiple times, and if a name matches multiple filters it is the last one to match which takes preference. For example, to match "main" and the "int" instatiation of "inc", while omitting the "Foo" constructor:

```
$ gcov -t -m -M --include in --exclude Foo --include '<int>::inc' tmp
        -:    0:Source:tmp.cpp
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        2:    8:  void inc () { b++; }
        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;
```

Using gcov with GCC Optimization

If you plan to use gcov to help optimize your code, you must first compile your program with a special GCC option --coverage. Aside from that, you can use any other GCC options; but if you want to prove that every single line in your program was executed, you should not compile with optimization at the same time. On some machines the optimizer can eliminate some simple code lines by combining them with other lines. For example, code like this:

```
if (a != b)
  c = 1;
else
  c = 0;
```

can be compiled into one instruction on some machines. In this case, there is no way for gcov to calculate separate execution counts for each line because there isn't separate code for each line. Hence the gcov output looks like this if you compiled the program with optimization:

```
      100:   12:if (a != b)
      100:   13:  c = 1;
      100:   14:else
      100:   15:  c = 0;
```

The output shows that this block of code, combined by optimization, executed 100 times. In one sense this result is correct, because there was only one instruction representing all four of these lines. However, the output does not indicate how many times the result was 0 and how many times the result was 1.

Inlineable functions can create unexpected line counts. Line counts are shown for the source code of the inlineable function, but what is shown depends on where the function is inlined, or if it is not inlined at all.

If the function is not inlined, the compiler must emit an out of line copy of the function, in any object file that needs it. If fileA.o and fileB.o both contain out of line bodies of a particular inlineable function, they will also both contain coverage counts for that function. When fileA.o and fileB.o are linked together, the linker will, on many systems, select one of those out of line bodies for all calls to that function, and remove or ignore the other. Unfortunately, it will not remove the coverage counters for the unused function body. Hence when instrumented, all but one use of that function will show zero counts.

If the function is inlined in several places, the block structure in each location might not be the same. For instance, a condition might now be calculable at compile time in some instances. Because the coverage of all the uses of the inline function will be shown for the same source lines, the line counts themselves might seem inconsistent.

Long-running applications can use the "__gcov_reset" and "__gcov_dump" facilities to restrict profile collection to the program region of interest. Calling "__gcov_reset(void)" will clear all run-time profile counters to zero, and calling "__gcov_dump(void)" will cause the profile information collected at that point to be dumped to .gcda output files.

Instrumented applications use a static destructor with priority 99 to invoke the "__gcov_dump" function. Thus "__gcov_dump" is executed after all user defined static destructors, as well as handlers registered with "atexit".

If an executable loads a dynamic shared object via dlopen functionality, -Wl,--dynamic-list-data is needed to dump all profile data.

Profiling run-time library reports various errors related to profile manipulation and profile saving. Errors are printed into standard error output or GCOV_ERROR_FILE file, if environment variable is used. In order to terminate immediately after an errors occurs set GCOV_EXIT_AT_ERROR environment variable. That can help users to find profile clashing which leads to a misleading profile.

SEE ALSO

gpl(7), gfdl(7), fsf-funding(7), gcc(1) and the Info entry for gcc.

COPYRIGHT