NAME
gcov – coverage testing tool

SYNOPSIS
gcov [-v|--version] [-h|--help]
    [-a|--all--blocks]
    [-b|--branch--probabilities]
    [-c|--branch--counts]
    [-n|--no--output]
    [-l|--long--file--names]
    [-p|--preserve--paths]
    [-f|--function--summaries]
    [-o|--object--directory directory|file] sourcefile
    [-u|--unconditional--branches]

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
- - h
    Display help about using gcov (on the standard output), and exit without doing any further processing.
- - v
## --version
Display the `gcov` version number (on the standard output), and exit without doing any further processing.

## --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.

## --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 `--unconditional-branches` option is given.

## --branch-counts
Write branch frequencies as the number of branches taken, rather than the percentage of branches taken.

## --no-output
Do not create the `gcov` output file.

## --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. If you use the `--preserve-paths` option, both the including and included file names will be complete path names.

## --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 .. components renamed to `^`. This is useful if sourcefiles are in several different directories. It also affects the `--long-file-names` option.

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

## --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 source file name, without its extension. If a file is specified here, the data files are named after that file, without its extension. If this option is not supplied, it defaults to the current directory.

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

`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 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.

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 and ###### for lines which were never executed. Some
lines of information at the start have line_number of zero.

The 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 two special GCC options: --profile-arcs
--ftest-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
--profile-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.c, this is what
you see when you use the basic gcov facility:

$ gcc -fprofile-arcs -ftest-coverage tmp.c
$ a.out
$ gcov tmp.c

90.00% of 10 source lines executed in file tmp.c
Creating tmp.c.gcov.

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

  -: 0:Source:tmp.c
  -: 0:Graph:tmp.gcno
  -: 0:Data:tmp.gcda
  -: 0:Runs:1
  -: 0:Programs:1
  -: 1:#include <stdio.h>
  -: 2:
  -: 3:int main (void)
  1: 4:
  1: 5: int i, total;
  -: 6:
  1: 7: total = 0;
  -: 8:
  11: 9: for (i = 0; i < 10; i++)
10:    total += i;
11: 
12: if (total != 45)
###: 13:    printf ("Failure\n");
-: 14: else
  1: 15:    printf ("Success\n");
  1: 16: return 0;
-: 17:}

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

```
-: 0:Source:tmp.c
-: 0:Graph:tmp.gcno
-: 0:Data:tmp.gcdar
-: 0:Runs:1
-: 0:Programs:1
-: 1:#include <stdio.h>
-: 2:
-: 3:int main (void)
  1: 4:{
  1: 4:block 0
  1: 5: int i, total;
  1: 6:
  1: 7: total = 0;
  1: 8:
  11: 9: for (i = 0; i < 10; i++)
  11: 9:block 0
  10: 10: total += i;
  10: 10:block 0
  11: 12: if (total != 45)
  1: 12:block 0
###: 13:    printf ("Failure\n");
$$$$$: 13:block 0
  14: else
  15:    printf ("Success\n");
  15:block 0
  16: return 0;
  16:block 0
-: 17:}
```

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 13 contains a basic block that was not executed.

When you use the -b option, your output looks like this:
$ gcov -b tmp.c
90.00% of 10 source lines executed in file tmp.c
80.00% of 5 branches executed in file tmp.c
80.00% of 5 branches taken at least once in file tmp.c
50.00% of 2 calls executed in file tmp.c
Creating tmp.c.gcov.

Here is a sample of a resulting tmp.c.gcov file:

```
#: 0:Source:tmp.c
#: 0:Graph:tmp.gcno
#: 0:Data:tmp.gcda
#: 0:Runs:1
#: 0:Programs:1
#: 1:#include <stdio.h>
#: 2:
#: 3:int main (void)

function main called 1 returned 1 blocks executed 75%
  1: 4:{
  1: 5: int i, total;
  #: 6:
  1: 7: total = 0;
  #: 8:
  11: 9: for (i = 0; i < 10; i++)
  branch 0 taken 91% (fallthrough)
  branch 1 taken 9%

  10: 10: total += i;
  #: 11:
  1: 12: if (total != 45)
  branch 0 taken 0% (fallthrough)
  branch 1 taken 100%
  #####: 13: printf ("Failure\n");
  call 0 never executed
  #: 14: else
  1: 15: printf ("Success\n");
  call 0 called 1 returned 100%
  1: 16: return 0;
  #: 17:}
```

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.

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 --profile-archs, 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.

Using gcov with GCC Optimization

If you plan to use gcov to help optimize your code, you must first compile your program with two special
GCC options: --profile-archs --ftest-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:

```c
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:

```plaintext
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.

SEE ALSO

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

COPYRIGHT


Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free
Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with
the Invariant Sections being “GNU General Public License” and “Funding Free Software”, the Front-
Cover texts being (a) (see below), and with the Back-Cover Texts being (b) (see below). A copy of the
license is included in the gfdl(7) man page.
(a) The FSF’s Front-Cover Text is:
   A GNU Manual

(b) The FSF’s Back-Cover Text is:
   You have freedom to copy and modify this GNU Manual, like GNU
   software. Copies published by the Free Software Foundation raise
   funds for GNU development.