NAME

gct, gct.exec – Generic Coverage Tool

SYNOPSIS

gct [**–test-control** *control_file_name*] [**–test-map** *map_file_name*] [**–test-dir** *master_directory*] [**–test-nolink**] [**–test-traditional**] [**–test-cc** *alternate_compiler*] *filename* ...

DESCRIPTION

gct is a generic coverage tool for C programs. It implements branch coverage, multi-condition coverage, loop coverage, weak mutation coverage, and more. This description is a brief summary; see the User's Manual and *A Tutorial Introduction to GCT* for details.

The tool works as a preprocessor for C source files. It parses the source, adds code to test whether the program's execution satisfies certain *coverage conditions*, then passes the instrumented program to the C compiler. GCT also writes a *map file* which describes the meaning and location of each of the coverage conditions. The executable, when run, writes a *log file*. **greport**(1) is then used to see which coverage conditions have yet to be satisfied. More tests are then written in the hopes of satisfying the coverage conditions.

The *control file* is used to control GCT. Control is very flexible and extensive; see the user documentation.

gct normally uses **cc** to compile the instrumented code. GCT's installer may have also created shell scripts that cause GCT to use different C compilers. Such scripts are normally named **gct-***compilername*. Using the control file, you may also tell GCT not to invoke any compiler; this is useful when the instrumented code is to be compiled on a different machine.

gct is actually a shell script that invokes the true executable, **gct.exec,** with whatever options are needed for the system's C compiler.

OPTIONS

These options, when passed to **gct,** will be passed along to **gct.exec.**

–test-dir *master_directory*
: Specifies the *master directory* where the map file will be stored and where **gct** expects to find the control file. The default is the current directory. If you work with hierarchical makefiles that invoke other makefiles in other directories, this option must be used.

–test-control *control_file_name*
: This option changes the default name for the control file from "gct-ctrl" to some other name. Unless the argument is an absolute pathname, it is taken to be relative to the master directory.

–test-map *map_file_name*
: This option changes the default name for the map file from "gct-map" to some other name. Unless the argument is an absolute pathname, it is taken to be relative to the master directory.

–test-cc *alternate_compiler*
: Tell GCT to use *alternate_compiler*. The default is **cc.**

–test-nolink
: This tells GCT not to link executables when it normally would. **gct-remap** is not executed.

–test-traditional
: Tell **gct.exec** to treat the program as traditional C, not ANSI C. The **gcc** script will normally set this if it's needed. The meaning of **-test-traditional** is identical to that of gcc's **-traditional** option. See the gcc(1) manpage (delivered with GCT). The **-traditional** option may also be used, but it will be passed to any C compiler GCT invokes, which may not be appropriate.

In addition to these options, **gct** accepts all the options that **gcc** version 1.37 does - that is, a superset of the options commonly accepted by C compilers. See gcc(1). In addition to obeying the options itself - if relevant - GCT passes options not beginning with "-test" to any compiler it invokes.

CONTROL FILE EXAMPLES

These are intended to be a quick reminder of the syntax. See the User's Guide for complete details.

```
(options instrument     # Instrument all files by default
    -link)              # Don't produce an executable

# Turn on standard subsystem coverage measures
(coverage branch loop multi relational)

test.c          # Instrument test.c
-test2.c        # Don't instrument test2.c

(logfile /usr/tmp/MYLOG)    # Use this file to store the log.

# Instrument c-parse.tab.c.  Also instrument all files
# included in it (via C's #include directive).
# Turn off branch coverage, except for routine yyerror.
# Don't instrument functions hash and is_reserved_word.
(c-parse.tab.c
        (options instrument-included-files)
        (coverage -branch)          # Turn off for all.
   (yyerror (coverage branch))    # Turn back on for yyerror.
   -hash
   -is_reserved_word)
```

DIAGNOSTICS

gct instruments the files and returns exit code 0. On any error, the input files remain untouched and a non-zero exit code is returned.

SEE ALSO

gcc (1), gclean (1), gedit (1), gfilter (1), gmerge (1), gnewer (1), greport (1), grestore (1), gsummary (1), gct-remap (1), gct-newlog (1), gct-init (1), *Generic Coverage Tool (GCT) User's Guide*, *A Tutorial Introduction to GCT*

NAME

greport – report on execution of a program instrumented by gct.

SYNOPSIS

greport [**–l**] [**–n**] [**–test-dir** *master_directory*] [**–test-map** *map_file_name*] [**–edit**]
[**–all**] [**–show-ignored**] [[**–visible-file | –vf**] *source_file_regex*]
[[**–visible-routine | –vr**] *source_routine_regex*] [*log_file_name*]

DESCRIPTION

greport uses the map file and the log file to produce a report on the test execution that resulted in the log file. The report contains information about conditions that have not yet been fully tested. The map file is created during the instrumentation of the source code by **gct**. The log file is usually written by gct_writelog() during a test run. The map file is named *gct-map* by default. The log file is given as the single argument. If no log file is given, **greport** reads from standard input.

The resulting report looks like the output of a C compiler and can be used with **error(1)** or the GNU Emacs next-error function. The complete output format is described in *Generic Coverage Tool (GCT) Documentation*.

Example:

```
host-1% greport test3_log
"lc.c", line 247: if was taken TRUE 0, FALSE 3 times.
"lc.c", line 268: operand i might be constant.
"lc.c", line 568: loop zero times: 0S, one time: 0, many times: 1551.
```

This report shows that a branching statement was never taken true, a particular reference to the variable I always yielded the same value, and a loop was never executed exactly one or exactly zero times. The "0S" in the last case indicates that the user has suppressed this condition with **gedit(1)** -- the loop can never execute zero times.

If another test is run that causes the if to be taken in the TRUE direction, a later report will not display the first line. If that test also causes the loop to execute one time, the last line will not be shown, even though the loop never executed zero times.

OPTIONS

–test-dir *master_directory*
The *master directory* is the directory where all of **gct**'s data files are kept. By default, it is the current directory. This option changes the default.

–test-map *map_file_name*
By default, information about coverage conditions is found in the file "gct-map" in the master directory. This option changes the default. Unless the argument is an absolute pathname, it is taken to be relative to the master directory.

–l Normally, **greport** prints file names relative to the master directory. Given this option, it prints them as absolute pathnames.

–all **greport** will display the values for all coverage conditions, satisfied or no. Such a display might look like:

```
"test.c", line 3: operator < might be <=. (L==R) [0]
"test.c", line 4: while was taken TRUE 1, FALSE 1 times.
"test.c", line 4: loop zero times: 0S, one time: 1, many times: 0.
"test.c", line 4: operand argc might be constant. [1]
```

For output lines that don't normally include counts, the number of times a condition has been satisfied is given in trailing brackets.

–show-ignored

Normally, **greport** does not display lines you've ignored (with either **gedit(1)** or by manually editing the mapfile). If this option is given, ignored lines are displayed; this is mainly useful for reviewing earlier decisions to ignore when nearing the end of testing. In the output, ignored counts are suffixed with "I". Note that this option does not imply **−all.** To distinguish between ignoring done with **gedit** and ignoring done by manually editing the mapfile, use the **−edit** option. The editing portion of the output describes only the edits to that line.

−n This option causes **greport** to show only those lines with counts greater than zero. This is most useful in a pipeline like this:

% gnewer -logical log later-log | greport -n

which gives a detailed report on what new coverage was added between *log* and *later-log*. The output format is the same as **−all.** Suppression is ignored.

−edit If this option is given, **greport**'s output is in a format that can be edited and used as **gedit**'s input. See **gedit(1)** for details.

−visible-file

−vf These two options are synonyms. They may be repeated to name several files. If given, **greport** reports only on coverage conditions within the named files. This option overrides any file editing inside the mapfile. Editing still applies to routines contained within the file, as well as to included files. If, for example, a routine is ignored, it remains ignored.

The argument to the option is actually an ed(1)-style regular expression, which is often useful for selecting subsets of a mapfile. For example,

% greport -vf "dir1/.*"

Produces a report on the contents of directory *dir1*. A simple shell script can place per-directory (or per-file) reports thoughout a directory hierarchy.

−visible-routine

−vr **greport** reports only on coverage conditions within the named routines. These options ignore any routine or file editing inside the mapfile. Editing still applies to lines within the routine. Edits to included files also still apply; this is inconsistent, but often useful.

Like **−vf,** the argument can be a regular expression. These options may also be repeated, and they can be combined with **−vf.** In that case, a line in either a visible file or a visible routine is shown.

EXAMPLES

See *A Tutorial Introduction to GCT*.

BUGS

The **visible-routine/vr** option applies to all routines with the same name; use grep to filter out files that don't interest you.

Because **−vf** and **−vr** work by ignoring all non-matching files or routines, the **−show-ignored** option negates their effects.

SEE ALSO

gsummary(1), **gedit**(1), **gct**(1).

NAME

gsummary − summary report on execution of a program instrumented by gct.

SYNOPSIS

gsummary [**−test-dir** *master_directory*] [**−test-map** *map_file_name*]
[[**−visible-file | −vf**] *source_file_regex*] [[**−visible-routine | −vr**] *source_routine_regex*]
[**−files | −f**] [**−routines | −r**] [*log_file_name*]

DESCRIPTION

gsummary uses the map file and log file to produce a summary report. The map file is created during the instrumentation of the source code by **gct**. The log file is usually written by gct_writelog() during a test run. The map file's default name is *gct-map*. The log file is given as the single argument. If no log file is given, **gsummary** reads from standard input.

The resulting report looks like this:

```
ernie-5% gsummary LOG

BINARY BRANCH INSTRUMENTATION (76 conditions total)
3 ( 3.95%) not satisfied.
73 (96.05%) fully satisfied.

SWITCH INSTRUMENTATION (14 conditions total)
0 ( 0.00%) not satisfied.
14 (100.00%) fully satisfied.

LOOP INSTRUMENTATION (24 conditions total)
4 (16.67%) not satisfied.
20 (83.33%) fully satisfied.

MULTIPLE CONDITION INSTRUMENTATION (38 conditions total)
2 ( 5.26%) not satisfied.
36 (94.74%) fully satisfied.

OPERATOR INSTRUMENTATION (27 conditions total)
1 ( 3.70%) not satisfied.
26 (96.30%) fully satisfied.

SUMMARY OF ALL CONDITION TYPES (179 total)
10 ( 5.59%) not satisfied.
169 (94.41%) fully satisfied.
```

This report shows that three branch directions were never taken, all cases (including default) were taken, four loop execution conditions were not satisfied, and so on.

In cases where the user suppressed some conditions using **gedit(1)**, the contribution of the suppressed conditions will be given in brackets, like this:

```
BINARY BRANCH INSTRUMENTATION (76 conditions total)
0 ( 0.00%) not satisfied.
76 (100.00%) fully satisfied. [2 ( 2.63%) suppressed]
```

In this case, two branch directions were manually suppressed (presumably because they are impossible to exercise).

OPTIONS

–visible-file

–vf These two options are synonyms. They may be repeated to name several files. If given, **gsummary** reports only on coverage conditions within the named files. These options ignore any file editing inside the mapfile. Editing still applies to routines contained within the file, as well as to included files. If, for example, a routine is ignored, it remains ignored

The argument to the option is actually an ed(1)-style regular expression, which is often useful for selecting subsets of a mapfile. For example,

```
% gsummary -files -vf "dir1/.*"
```

Produces a per-file summary for the directory *dir1*. A simple shell script can be used to place summaries throughout a directory hierarchy.

–visible-routine

–vr **gsummary** reports only on coverage conditions within the named routines. This option overrides any routine or file editing inside the mapfile. Editing still applies to lines within the routine. Edits to included files also still apply; this is inconsistent, but often useful.

Like **–vf,** the argument can be a regular expression. These options may also be repeated, and they can be combined with **–vf.** In that case, a line in either a visible file or a visible routine is used when calculating the summary.

–files

–f Terse per-file summaries are printed instead of a grand summary. Each line summarizes a single file. The line format is designed so that gsummary's output can be used by tools like **awk(1)** and **sort(1)**. A line of output might look like this:

```
gcc.c      77=ALL  78=BR  50=SW  67=LP  76=ML  81=<  260#
```

The first numeric entry is the total percentage of coverage conditions satisfied for the file. The next five numeric entries give the percent satisfied for branch, switch, loop, multicondition, and relational operator coverage. The final entry is the number of coverage conditions. The tags are terse to try to fit all the information in 80 columns.

Routine instrumentation uses the tag "ROUT", call uses "CALL", race uses "RACE", operand uses "x", and operator uses "<". (The last is not a typo - recall that relational coverage is really a kind of operator coverage.)

–routines

–r Terse per-routine summaries are printed. The format is the same as for the **-files** option.

–test-dir *master_directory*

The *master directory* is the directory where all of **gct**'s data files are kept. By default, it is the current directory. This option changes the default.

–test-map *map_file_name*

By default, information about coverage conditions is found in the file "gct-map" in the master directory. This option changes the default. Unless the argument is an absolute pathname, it is taken to be relative to the master directory.

EXAMPLES

See *A Tutorial Introduction to GCT* for simple examples.

This command prints the summaries for the five source files with the lowest coverage:

```
% gsummary -files /usr/tmp/GCTLOG | sort -n +1 | head -5
```

This command prints the same summaries, but this time for the least-exercised routines in the file *gct-build.c*:

```
% gsummary -r -vf gct-build.c /usr/tmp/GCTLOG | sort -n +1 | head -6
```

INCLUDED FILES

If you use the instrument-included-files option in the control file, you need to know how **gsummary** handles included files.

The argument to **-visible-file** is always a main filename (such as appears in the control file), never an included file. There is no way to get a per-include-file summary.

The **-files** option summarizes an entire file, including instrumented routines from included files.

If a routine comes from an included file, the **-routines** option will note both the main filename and the included filename.

SEE ALSO

gct(1), **greport**(1).

NAME

gedit − manually suppress gct coverage conditions

SYNOPSIS

gedit [**−test-dir** *master_directory*] [**−test-map** *map_file_name*] [**−v**] [*report_file_name*]

DESCRIPTION

gedit modifies the map file to control what **greport(1)** and **gsummary(1)** display. To use **gedit**, you must first create a report file using **greport -edit**:

% greport -edit GCTLOG > REPORT

This report looks like:

"test.c", line 3: [0: 1 0] if was taken TRUE 1, FALSE 0 times.
"test.c", line 3: [1: 0] operator < might be <=.
"test.c", line 3: [2: 1 0] condition 1 (argc, 1) was taken TRUE 1, FALSE 0 times.
"test.c", line 3: [3: 0 0] condition 2 (argc, 1) was taken TRUE 0, FALSE 0 times.

Each line represents one or more coverage conditions. (For example, a branch represents two: one that requires the branch to be taken TRUE, and one that requires the branch to be taken FALSE.) The text of interest is in brackets. The number before the colon is irrelevant to the user; it is the index for the first of that line's coverage conditions. The numbers after the colon show how often the respective conditions were satisfied. For example, in the case of the if statement, the TRUE condition was satisfied once and the FALSE condition never. The operator condition was satisfied zero times.

If a condition is impossible to satisfy, you may want to tell **greport** not to display information about it, even if no test case has ruled it out. The first step in doing this is to suppress the condition by replacing its number by an "S" or "s". You can also add the "S" or "s" to the number. Here is an example:

"test.c", line 3: [0: 1 0S] if was taken TRUE 1, FALSE 0 times.
"test.c", line 3: [1: 0s] operator < might be <=.
"test.c", line 3: [2: 1 s] condition 1 (argc, 1) was taken TRUE 1, FALSE 0 times.
"test.c", line 3: [3: 0 S] condition 2 (argc, 1) was taken TRUE 0, FALSE 0 times.

By doing this, you have declared that you do not want to see the first three lines again -- all the conditions have either been satisfied by a test or suppressed by you. You want the fourth line to be shown until the condition evaluates TRUE.

To cause your changes to take effect, run **gedit**. It will modify the mapfile.

% gedit REPORT

or

% gedit < REPORT

From this point on, **greport** will treat suppressed conditions as if they'd been satisfied. The output from greport will look like:

"test.c", line 3: condition 2 (argc, 1) was taken TRUE 0, FALSE 0S times.

Notice that the FALSE case has been tagged with an S. **gsummary** will treat suppressed conditions as satisfied, but it will also summarize the number of suppressed conditions.

In addition to suppressing a condition, you may "ignore" it. To do so, you use "I" or "i" just as you used "S" or "s" above:

"test.c", line 3: [0: 1 0I] if was taken TRUE 1, FALSE 0 times.
"test.c", line 3: [1: 0i] operator < might be <=.
"test.c", line 3: [2: 1 i] condition 1 (argc, 1) was taken TRUE 1, FALSE 0 times.
"test.c", line 3: [3: 0 I] condition 2 (argc, 1) was taken TRUE 0, FALSE 0 times.

Unlike a suppressed condition, which is treated as if it were satisfied, an ignored condition effectively does not exist. Ignored conditions are not included in **greport** output (even if the *-all* option is used), and they play no part in **gsummary** totals. Ignoring applies to an entire line: for an if, ignoring either of the true or false conditions also ignores the other.

In GCT 1.4, ignoring plays the role that deleting mapfile entries did in earlier versions of GCT. Never delete mapfile entries in GCT 1.4.

OPTIONS

−test-dir *master_directory*
: The *master directory* is the directory where all of **gct**'s data files are kept. By default, it is the current directory. This option changes the default.

−test-map *map_file_name*
: This option changes the default name for the map file from "gct-map" to some other name. Unless the argument is an absolute pathname, it is taken to be relative to the master directory.

−v
: Verbose output to standard error. Not particularly useful.

ERRORS

If **gedit** discovers an error in the input, it will quit, possibly leaving the mapfile partially edited. **gedit** makes a backup copy and leaves it in a file with the extension *gbk.*

BUGS

There is no way to unsuppress a condition, except by manually editing the mapfile. The *Generic Coverage Tool (GCT) User's Guide* describes the mapfile format.

SEE ALSO

gct(1), **greport**(1).

NAME

gct-remap − Update a GCT mapfile when a file is re-instrumented

SYNOPSIS

gct-remap [**−test-dir** *master_directory*] [**−test-map** *mapfile*] [**−forget**] [**−strict**]
[**−update-copy**] [**−v**]

NOTE

This program is only available with GCT Expansion Kit 1.

DESCRIPTION

A normal instrumentation of a system begins with **gct-init(1)**, continues by using GCT on many C files, then finishes by using GCT again to link all the resulting object files together. When one of the files is changed, it is reinstrumented by calling GCT on it, then using GCT to relink the system.

If GCT is not used to relink the system (if, for example, the -produce-object option is used), a normally hidden step - calling **gct-remap** - must be done manually before the system is linked. If this is not done, the mapfile will not be updated and will not correspond to the executable. Neither **greport** nor **gsummary** will work correctly. (They will fail in an obvious way.)

In addition to incrementally updating the mapfile, **gct_remap** also preserves edits made to the mapfile, both those made with **gedit(1),** called "line edits", and the other kinds described in the *Generic Coverage Tool (GCT) User's Guide*.

If **gct_remap** made any changes, it leaves backup copies of the changed files in *gct-map.gbk*, *gct-ps-defs.c˜* *gct-ps-defs.h˜* (The inconsistent naming is to avoid filename length limits on older versions of UNIX.)

gct_remap takes no arguments, other than options.

OPTIONS

−test-dir *master_directory*
The *master directory* is the directory where all of **gct**'s data files are kept. By default, it is the current directory. This option changes the default. Because **gct-remap** also updates the files gct-ps-defs.c and gct-ps-defs.h, this option must be given whenever the mapfile is not in the current directory. That is, you must type

% gct-remap -test-dir /tmp -test-map my-mapfile

not

% gct-remap -test-map /tmp/my-mapfile

−test-map *map_file_name*
This option changes the default name for the map file from "gct-map" to some other name. Unless the argument is an absolute pathname, it is taken to be relative to the master directory. Note that **gct-remap** also updates the files gct-ps-defs.c and gct-ps-defs.h. Their names cannot change.

−strict Preserve line edits only on routines whose code is completely unchanged. (Comments and white space can change.) The default is to preserve them on routines with the same instrumentation in the same order as before. Other types of edits are always preserved.

−forget Line edits are erased on all files, whether reinstrumented or not. It is as if **gedit** had never been run. Other types of edits are always preserved.

−update-copy
There may be more than one mapfile, each having different edits. Once **gct-remap** has been run on the first, this option should be used for the others. The updated version is first concatenated to the end of the other copies, to tell **gct-remap** what has changed. Here is an example.

% make CC=gct # Updates gct-map

```
% cat gct-map >> other-mapfile
% gct-remap -test-map other-mapfile -update-copy
```

The already-updated mapfile should contain no edits; if it does, they will be merged into the updated copy.

This option can also be used to update a completely new mapfile with the edits of an older mapfile:

```
% gct-init
% make CC=gct          # Create gct-map from scratch.
% cat gct-map >> older-mapfile
% gct-remap -test-map older-mapfile -update-copy
% mv older-mapfile gct-map
```

In both cases, the newer (current) mapfile should be at the *end* of the concatenated mapfile given to **gct-remap.**

−v **gct-remap** commentary normally tells you only what edits were lost because of changes. If this option is given, it also tells you what edits were preserved.

BUGS AND DEFICIENCIES

gct-remap commentary will describe a routine as being inside a particular file. The routine may actually be inside a file included by that file.

SEE ALSO

gct(1), **gedit**(1), **gct-newlog**(1),

NAME

gct-newlog − Update a GCT logfile to match new mapfile.

SYNOPSIS

gct-newlog [**−test-dir** *master_directory*] [**−test-map** *new-mapfile*] [**−strict**] [**−v**] *old-mapfile old-logfile*

NOTE

This program is only available with GCT Expansion Kit 1.

DESCRIPTION

This program takes an *old-mapfile,* an *old-logfile,* and a *new-mapfile* and produces a new logfile by updating the old one. It is typically used to update a logfile after some files in a program have been changed and reinstrumented.

If a routine in the old mapfile is compatible with the new mapfile, the corresponding counts in the old logfile are carried over to the new logfile. Otherwise, they are set to zero. By default, a routine is compatible if it has the same name, came from the same source file, and had the same instrumentation added in the same order. The new logfile is written to standard output.

The following is a possible use of **gct-newlog:**

```
% gct -o exec test.c test2.c
% gct-newlog gct-map.gbk GCTLOG > GCTLOG.new
```

This takes advantage of the fact that GCT's updating of the mapfile at link time leaves a usable version of the old mapfile in *gct-map.gbk.* It is usually better to explicitly save a copy of the old mapfile before reinstrumenting code.

gct-newlog does not require that one mapfile be a newer version of the other. It can be used to combine mapfiles from quite independent programs. (This is useful only if the two programs share routines.)

OPTIONS

−test-dir *master_directory*
: The *master directory* is the directory where all of **gct**'s data files are kept. By default, it is the current directory. This option changes the default. **gct-remap** also looks for the *old-mapfile* in this directory.

−test-map *map_file_name*
: This option changes the default name for the new map file from "gct-map" to some other name. Unless the argument is an absolute pathname, it is taken to be relative to the master directory.

−strict
: Use a stricter definition of compatibility: two versions of a routine are compatible only if their code is identical. (Comments and white space can change.)

−v
: **gct-newlog** normally does not print messages about which routines are incompatible. This option turns on those messages.

BUGS AND DEFICIENCIES

gct-newlog prints a message when all logfile entries for a routine are set to zero. That message describes the routine as being inside a particular C file. It may actually be inside a file included by that C file.

SEE ALSO

gct(1), **gedit**(1), **gct-remap**(1).

NAME

gnewer – produce difference of two gct log files.

SYNOPSIS

gnewer [-logical] *old-log new-log*

DESCRIPTION

gnewer is used to compare two log files. The *new-log* is a cumulative log, containing all the information of *old-log* plus information from later test runs.

gnewer is most often used when testing programs that run cyclically. For example, suppose you have a particular test for the UNIX kernel, and you want to see which conditions it forces. The process of booting the kernel may satisfy some of those conditions. Your test session would look like this:

1. Boot the kernel and extract a snapshot, *old-log*, of the conditions that were satisfied.
2. Execute your test.
3. Extract another snapshot, *new-log*.
4. Run **gnewer** to see which conditions were satisfied by the test (plus whatever conditions were satisfied during the extraction):

```
% gnewer old-log new-log | greport
% gnewer old-log new-log | gsummary
```

For each coverage condition, the arithmetic difference between the new and old logfiles is printed to standard output. This does not distinguish between two cases:

1) The condition was satisfied 5 times in the old log file and 10 times in the new one.
2) The condition was satisfied 0 times in the old log file and 5 times in the new one.

In both cases, the resulting logfile will have 5 for that coverage condition. To distinguish them, use the -*logical* option. Case 1 will result in a 0 in the output (meaning that either both or neither of the test suites satisfied the coverage condition). Case 2 will result in a 1 (meaning that the new tests satisfy a previously unsatisfied coverage condition).

In summary, **gnewer** piped into **gsummary** will summarize what the new tests do; **gnewer -logical** piped into **gsummary** will summarize what the new tests uniquely do.

It is sometimes useful to pipe the output of **gnewer -logical** to **greport -n**; see the **greport** manpage for an example.

SEE ALSO

greport(1), **gct**(1).

NAME

gfilter − filter GCT greport output.

SYNOPSIS

gfilter [**−branch**] [**−loop**] [**−operator**] [**−operand**] [**−multi**] [**−routine**] [**−call**] [**−race**] [*report_file_name*]

DESCRIPTION

gfilter reads the output from **greport**(1). It passes through only report lines that correspond to flag arguments. The flag arguments have the same names as those used in the **gct**(1) control file.

There is no **-relational** option; the **-operator** option applies to relational as well as all other kinds of operator lines.

If no file name is given, standard input is used.

Example:

```
ernie-62% gfilter report.g
ernie-63% gfilter -multi report.g
"test.c", line 3: condition 1 (argc, 1) was taken TRUE 1, FALSE 0 times.
"test.c", line 3: condition 2 (argc, 1) was taken TRUE 0, FALSE 0 times.
ernie-64% gfilter -branch -multi report.g
"test.c", line 3: if was taken TRUE 1, FALSE 0 times.
"test.c", line 3: condition 1 (argc, 1) was taken TRUE 1, FALSE 0 times.
"test.c", line 3: condition 2 (argc, 1) was taken TRUE 0, FALSE 0 times.
"test.c", line 4: while was taken TRUE 1, FALSE 1 times.
```

SEE ALSO

greport(1), **gct**(1).

NAME

gmerge − merge GCT log files

SYNOPSIS

gmerge *log_file_name1* [*log_file_name2* ...]

DESCRIPTION

gmerge takes one or more log files produced by programs instrumented with gct(1). It produces a summary logfile on its standard output. The counts in the summary log file are the sum of the counts in the input log files.

A maximum of 100 logfiles can be merged at a time.

Example of usage:

```
host-1% gmerge test3_log test4_log > all_log
```

SEE ALSO

greport(1), **gct**(1).

NAME

gct-init – initialize GCT files

SYNOPSIS

gct-init [–**test-map** *mapfile-name*]

DESCRIPTION

gct-init copies initial versions of GCT utility files into the current directory. By default, it creates a mapfile named *gct-map*. If the **-test-map** option is given, its argument is used as the name.

SEE ALSO

gct(1).

NAME

gclean − remove files created by GCT.

SYNOPSIS

gclean

DESCRIPTION

gclean removes all the files in the current directory that GCT created. These files are:

gct-ps-defs.h
gct-map
gct-map.gbk
gct-rscript
gct-rscript.bk
gct_backup (and all its contents)
gct-ps-defs.c
gct-ps-defs.c˜
gct-ps-defs.h
gct-ps-defs.h˜
gct-write.c
gct-defs.h

If you edited any of these files (when using race coverage, for example, or by calling **gedit**), you may not want to use **gclean.**

BUGS

gclean doesn't remove all the gct_backup directories that might have been created; only the one in the current directory.

SEE ALSO

gct(1), **gedit**(1).

NAME

grestore − restore the original versions of files instrumented by gct.

SYNOPSIS

grestore [**−test-dir** *master_directory*]

DESCRIPTION

grestore is only relevant if (options -produce-object) was used in the control file. In this case, GCT moves the original source into a subdirectory named *gct_backup*, and instructions for copying it back are placed in a file called **gct-rscript** in the master directory. **grestore** restores the original source.

If the source files are in several directories, there will be the same number of **gct_backup** directories, but only one **gct-rscript**.

Example of use:

    host-1% grestore

You may get a message that **gct-rscript** does not exist when you think it should. The usual cause of this is as follows:

1) You run GCT from within your makefile. GCT fails for some reason, such as a syntax error in a file. The makefile halts because of the error, therefore **grestore** isn't run.

2) You fix the problem and run **make** again. GCT immediately fails because you're trying to instrument already-instrumented files.

3) You run **grestore** by hand. Unfortunately, the first thing the makefile did was run **gct-init**, which removes **gct-rscript**.

Don't panic. The previous version of **gct-rscript** is in **gct-rscript.bk**. All you need to do is type:

```
% mv gct-rscript.bk gct-rscript
% grestore
```

Even if **gct-rscript.bk** is missing for some reason, the original versions of the instrumented files are still in the **gct_backup** directories, and they can be copied to their original locations manually.

To avoid the problem, make sure you run **grestore** manually if the make exits abnormally.

grestore does not remove the copies of the files in the **gct_backup** directory. This wastes space, but can be useful if the original copy is accidentally deleted.

OPTIONS

−test-dir *master_directory*

If you specified a master directory when you used **gct**, **gct-rscript** is in that directory. You must use the **-test-dir** option to give **grestore** the same directory as you gave **gct**.

BUGS

The restored file's attributes should be identical to the original file's, save for an updated modification time (to force replacement of the instrumented object file the next time the program is rebuilt). GCT currently falls short. The read-write-execute bits of the mode will be the same, but other mode bits are lost. (It's rare for C source files to have the other bits set.) The file will be owned by the instrumenter, who may not have been the original owner.

SEE ALSO

gct(1).

NAME

glist − Create a code listing with GCT test coverage annotations.

SYNOPSIS

glist [**-s** *point-size*] [**-l**] [**-t**] [**-r**] *report-file-name...*

DESCRIPTION

glist creates a listing of your source code, annotated with messages indicating where you have not fully tested your code. **glist** uses *report-file-name* or stdin to determine where to place annotations. The output is PostScript(tm) suitable for printing on any PostScript printer or previewer.

glist requires **perl**

OPTIONS

−s *point-size*
Specifies the point size to use printing lines of code. The default point size is 10 points.

−l Specifies to print line numbers in the right hand margin every five lines. By default **glist** only prints line numbers to the right of lines which have missing test coverage.

−t Specifies that **glist** should truncate lines which extend past the right hand margin. By default long lines are wrapped.

−r Specifies that the listing should be rotated (i.e. printed in a landscape orientation instead of portrait orientation).

SEE ALSO

gct(1), **greport**(1), **gfilter**(1).

AUTHOR

glist was writting by Pete Smoot of Hewlett-Packard Company, based on work by Gisle Aas of the Norwegian Computing Center.

NAME

gcc − GNU project C Compiler

SYNOPSIS

gcc [options] files

WARNING

This man page is an extract of the documentation of the *GNU C compiler* and is limited to the meaning of the options. It is updated only occasionally, because the GNU project does not use nroff. For complete, current documentation, refer to the Info file **gcc** or the DVI file **gcc.dvi** which are made from the Texinfo source file **gcc.texinfo**.

DESCRIPTION

The *GNU C compiler* uses a command syntax much like the Unix C compiler. The *gcc* program accepts options and file names as operands. Multiple single-letter options may *not* be grouped: **−dr** is very different from **−d −r**. When you invoke GNU CC, it normally does preprocessing, compilation, assembly and linking. File names which end in **.c** are taken as C source to be preprocessed and compiled; file names ending in **.i** are taken as preprocessor output to be compiled; compiler output files plus any input files with names ending in **.s** are assembled; then the resulting object files, plus any other input files, are linked together to produce an executable. Command options allow you to stop this process at an intermediate stage. For example, the **−c** option says not to run the linker. Then the output consists of object files output by the assembler. Other command options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; these are not documented here, but you rarely need to use any of them.

OPTIONS

Here are the options to control the overall compilation process, including those that say whether to link, whether to assemble, and so on.

−o *file* Place output in file *file*. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.

If **−o** is not specified, the default is to put an executable file in **a.out**, the object file *source***.c** in *source***.o**, an assembler file in *source***.s**, and preprocessed C on standard output.

−c Compile or assemble the source files, but do not link. Produce object files with names made by replacing **.c** or **.s** with **.o** at the end of the input file names. Do nothing at all for object files specified as input.

−S Compile into assembler code but do not assemble. The assembler output file name is made by replacing **.c** with **.s** at the end of the input file name. Do nothing at all for assembler source files or object files specified as input.

−E Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output.

−v Compiler driver program prints the commands it executes as it runs the preprocessor, compiler proper, assembler and linker. Some of these are directed to print their own version numbers.

−pipe Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

−B*prefix*

Compiler driver program tries *prefix* as a prefix for each program it tries to run. These programs are *cpp*, *cc1*, *as* and *ld*.

For each subprogram to be run, the compiler driver first tries the **−B** prefix, if any. If that name is not found, or if **−B** was not specified, the driver tries two standard prefixes, which are **/usr/lib/gcc-** and **/usr/local/lib/gcc-**. If neither of those results in a file name that is found, the unmodified program name is searched for using the directories specified in your **PATH** environment variable.

The run-time support file **gnulib** is also searched for using the −**B** prefix, if needed. If it is not found there, the two standard prefixes above are tried, and that is all. The file is left out of the link if it is not found by those means. Most of the time, on most machines, you can do without it.

You can get a similar result from the environment variable **GCC_EXEC_PREFIX**; if it is defined, its value is used as a prefix in the same way. If both the −**B** option and the **GCC_EXEC_PREFIX** variable are present, the −**B** option is used first and the environment variable value second.

-b*prefix* The argument *prefix* is used as a second prefix for the compiler executables and libraries. This prefix is optional: the compiler tries each file first with it, then without it. This prefix follows the prefix specified with −**B** or the default prefixes.

Thus, −**bvax-** −**Bcc/** in the presence of environment variable **GCC_EXEC_PREFIX** with definition **/u/foo/** causes GNU CC to try the following file names for the preprocessor executable:

cc/vax-cpp
cc/cpp
/u/foo/vax-cpp
/u/foo/cpp
/usr/local/lib/gcc-vax-cpp
/usr/local/lib/gcc-cpp
/usr/lib/gcc-vax-cpp
/usr/lib/gcc-cpp These options control the details of C compilation itself.

−**ansi** Support all ANSI standard C programs.

This turns off certain features of GNU C that are incompatible with ANSI C, such as the **asm**, **inline** and **typeof** keywords, and predefined macros such as **unix** and **vax** that identify the type of system you are using. It also enables the undesirable and rarely used ANSI trigraph feature.

The alternate keywords **__asm__**, **__inline__** and **__typeof__** continue to work despite −**ansi**. You would not want to use them in an ANSI C program, of course, but it useful to put them in header files that might be included in compilations done with −**ansi**. Alternate predefined macros such as **__unix__** and **__vax__** are also available, with or without −**ansi**.

The −**ansi** option does not cause non-ANSI programs to be rejected gratuitously. For that, −**pedantic** is required in addition to −**ansi**.

The macro **__STRICT_ANSI__** is predefined when the −**ansi** option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ANSI standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

−**traditional**
Attempt to support some aspects of traditional C compilers. Specifically:

∗ All **extern** declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.

∗ The keywords **typeof**, **inline**, **signed**, **const** and **volatile** are not recognized.

∗ Comparisons between pointers and integers are always allowed.

∗ Integer types **unsigned short** and **unsigned char** promote to **unsigned int**.

∗ Out-of-range floating point literals are not an error.

∗ All automatic variables not declared **register** are preserved by *longjmp*(3C). Ordinarily, GNU C follows ANSI C: automatic variables not declared **volatile** may be clobbered.

∗ In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.

∗ In the preprocessor, macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.

∗ The predefined macro **__STDC__** is not defined when you use **−traditional**, but **__GNUC__** is (since the GNU extensions which **__GNUC__** indicates are not affected by **−traditional**). If you need to write header files that work differently depending on whether **−traditional** is in use, by testing both of these predefined macros you can distinguish four situations: GNU C, traditional GNU C, other ANSI C compilers, and other old C compilers.

−O Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

Without **−O**, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Without **−O**, only variables declared **register** are allocated in registers. The resulting compiled code is a little worse than produced by PCC without **−O**.

With **−O**, the compiler tries to reduce code size and execution time.

Some of the **−f** options described below turn specific kinds of optimization on or off.

−g Produce debugging information in the operating system's native format (for DBX or SDB). GDB also can work with this debugging information.

Unlike most other C compilers, GNU CC allows you to use **−g** with **−O**. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops. Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

−gg Produce debugging information in GDB's own format. This requires the GNU assembler and linker in order to work.

This feature will probably be eliminated. It was intended to enable GDB to read the symbol table faster, but it doesn't result in enough of a speedup to be worth the larger object files and executables. We are working on other ways of making GDB start even faster, which work with DBX format debugging information and could be made to work with SDB format.

−w Inhibit all warning messages.

−W Print extra warning messages for these events:

∗ An automatic variable is used without first being initialized.

These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. If you don't specify **−O**, you simply won't get these warnings.

These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared **volatile**, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.

Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

These warnings are made optional because GNU CC is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{
  int x;
  switch (y)
    {
    case 1: x = 1;
      break;
    case 2: x = 4;
      break;
    case 3: x = 5;
    }
  foo (x);
}
```

If the value of *y* is always 1, 2 or 3, then *x* is always initialized, but GNU CC doesn't know this. Here is another common case:

```
{
  int save_y;
  if (change_y) save_y = y, y = new_y;
  ...
  if (change_y) y = save_y;
}
```

This has no bug because *save_y* is used only if it is set.

Some spurious warnings can be avoided if you declare as **volatile** all the functions you use that never return.

∗ A nonvolatile automatic variable might be changed by a call to *longjmp*(3C). These warnings as well are possible only in optimizing compilation.

The compiler sees only the calls to *setjmp*(3C). It cannot know where *longjmp*(3C) will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because *longjmp*(3C) cannot in fact be called at the place which would cause a problem.

∗ A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would evoke such a warning:

```
foo (a)
{
  if (a > 0)
    return a;
}
```

Spurious warnings can occur because GNU CC does not realize that certain functions (including *abort*(3C) and *longjmp*(3C)) will never return.

∗ An expression-statement contains no side effects.

In the future, other useful warnings may also be enabled by this option.

−Wimplicit
Warn whenever a function is implicitly declared.

−Wreturn-type
Warn whenever a function is defined with a return-type that defaults to **int**. Also warn about any **return** statement with no return-value in a function whose return-type is not **void**.

−Wunused
Warn whenever a local variable is unused aside from its declaration, and whenever a function is declared static but never defined.

−Wswitch
Warn whenever a **switch** statement has an index of enumeral type and lacks a **case** for one or more of the named codes of that enumeration. (The presence of a **default** label prevents this warning.) **case** labels outside the enumeration range also provoke warnings when this option is used.

−Wcomment
Warn whenever a comment-start sequence **/∗** appears in a comment.

−Wtrigraphs
Warn if any trigraphs are encountered (assuming they are enabled).

−Wall All of the above **−W** options combined. These are all the options which pertain to usage that we do not recommend and that we believe is always easy to avoid, even in conjunction with macros.

The other **−W**... options below are not implied by **−Wall** because certain kinds of useful macros are almost impossible to write without causing those warnings.

−Wshadow
Warn whenever a local variable shadows another local variable.

−Wid-clash-*len*
Warn whenever two distinct identifiers match in the first *len* characters. This may help you prepare a program that will compile with certain obsolete, brain-damaged compilers.

−Wpointer-arith
Warn about anything that depends on the size of a function type or of **void**. GNU C assigns these types a size of 1, for convenience in calculations with **void ∗** pointers and pointers to functions.

−Wcast-qual
Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a **const char ∗** is cast to an ordinary **char ∗**.

−Wwrite-strings
Give string constants the type **const char[***length***]** so that copying the address of one into a non-**const char ∗** pointer will get a warning. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using **const** in declarations and prototypes. Otherwise, it will just be a nuisance; this is why we did not make **−Wall** request these warnings.

−p Generate extra code to write profile information suitable for the analysis program *prof*(1).

−pg Generate extra code to write profile information suitable for the analysis program *gprof*(1).

−a Generate extra code to write profile information for basic blocks, suitable for the analysis program *tcov*(1). Eventually GNU *gprof*(1) should be extended to process this data.

−l*library*
Search a standard list of directories for a library named *library*, which is actually a file named **lib***library***.a**. The linker uses this file as if it had been specified precisely by name.

The directories searched include several standard system directories plus any that you specify with **−L**.

Normally the files found this way are library files--archive files whose members are object files. The linker handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an **−l** option and specifying a file name is that **−l** searches several directories.

−L*dir* Add directory *dir* to the list of directories to be searched for **−l**.

−nostdlib
Don’t use the standard system libraries and startup files when linking. Only the files you specify (plus **gnulib**) will be passed to the linker.

−m*machinespec*
Machine-dependent option specifying something about the type of target machine. These options are defined by the macro **TARGET_SWITCHES** in the machine description. The default for the options is also defined by that macro, which enables you to change the defaults.

These are the **−m** options defined in the 68000 machine description:

−m68020
−mc68020
Generate output for a 68020 (rather than a 68000). This is the default if you use the unmodified sources.

−m68000
−mc68000
Generate output for a 68000 (rather than a 68020).

−m68881
Generate output containing 68881 instructions for floating point. This is the default if you use the unmodified sources.

−mfpa

Generate output containing Sun FPA instructions for floating point.

−msoft-float
Generate output containing library calls for floating point.

−mshort
Consider type **int** to be 16 bits wide, like **short int**.

−mnobitfield
Do not use the bit-field instructions. **−m68000** implies **−mnobitfield**.

−mbitfield
Do use the bit-field instructions. **−m68020** implies **−mbitfield**. This is the default if you use the unmodified sources.

−mrtd
Use a different function-calling convention, in which functions that take a fixed number of arguments return with the **rtd** instruction, which pops their arguments while returning. This saves one instruction in the caller since there is no need to pop the arguments there.

This calling convention is incompatible with the one normally used on Unix, so you cannot use it if you need to call libraries compiled with the Unix compiler.

Also, you must provide function prototypes for all functions that take variable numbers of arguments (including **printf**(3S)); otherwise incorrect code will be generated for calls to those functions.

In addition, seriously incorrect code will result if you call a function with too many arguments. (Normally, extra arguments are harmlessly ignored.)

The **rtd** instruction is supported by the 68010 and 68020 processors, but not by the 68000.

These **−m** options are defined in the Vax machine description:

−munix
Do not output certain jump instructions (**aobleq** and so on) that the Unix assembler for the Vax cannot handle across long ranges.

−mgnu
Do output those jump instructions, on the assumption that you will assemble with the GNU assembler.

−mg
Output code for g-format floating point numbers instead of d-format.

These **−m** switches are supported on the Sparc:

−mfpu
Generate output containing floating point instructions. This is the default if you use the unmodified sources.

−msoft-float

Generate output containing library calls for floating point.

–mno-epilogue

Generate separate return instructions for **return** statements. This has both advantages and disadvantages; I don't recall what they are.

These **–m** options are defined in the Convex machine description:

–mc1

Generate output for a C1. This is the default when the compiler is configured for a C1.

–mc2

Generate output for a C2. This is the default when the compiler is configured for a C2.

–margcount

Generate code which puts an argument count in the word preceding each argument list. Some nonportable Convex and Vax programs need this word. (Debuggers don't; this info is in the symbol table.)

–mnoargcount

Omit the argument count word. This is the default if you use the unmodified sources.

–f*flag* Specify machine-independent flags. Most flags have both positive and negative forms; the negative form of **–ffoo** would be **–fno-foo**. In the table below, only one of the forms is listed--the one which is not the default. You can figure out the other form by either removing **no-** or adding it.

–fpcc-struct-return

Use the same convention for returning **struct** and **union** values that is used by the usual C compiler on your system. This convention is less efficient for small structures, and on many machines it fails to be reentrant; but it has the advantage of allowing intercallability between GCC-compiled code and PCC-compiled code.

–ffloat-store

Do not store floating-point variables in registers. This prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a **double** is supposed to have.

For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use **–ffloat-store** for such programs.

–fno-asm

Do not recognize **asm**, **inline** or **typeof** as a keyword. These words may then be used as identifiers. You can use **__asm__**, **__inline__** and **__typeof__** instead.

–fno-defer-pop

Always pop the arguments to each function call as soon as that function returns. Normally the compiler (when optimizing) lets arguments accumulate on the stack for several function calls and pops them all at once.

–fstrength-reduce

Perform the optimizations of loop strength reduction and elimination of iteration variables.

–fcombine-regs

Allow the combine pass to combine an instruction that copies one register into another. This might or might not produce better code when used in addition to **–O**. I am interested in hearing about the difference this makes.

–fforce-mem

Force memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load. I am interested in hearing about the difference this makes.

–fforce-addr
Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as **–fforce-mem** may. I am interested in hearing about the difference this makes.

–fomit-frame-pointer
Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. **It also makes debugging impossible.**

On some machines, such as the Vax, this flag has no effect, because the standard calling sequence automatically handles the frame pointer and nothing is saved by pretending it doesn't exist. The machine-description macro **FRAME_POINTER_REQUIRED** controls whether a target machine supports this flag.

–finline-functions
Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

If all calls to a given function are integrated, and the function is declared **static**, then the function is normally not output as assembler code in its own right.

–fcaller-saves
Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.

This option is enabled by default on certain machines, usually those which have no call-preserved registers to use instead.

–fkeep-inline-functions
Even if all calls to a given function are integrated, and the function is declared **static**, nevertheless output a separate run-time callable version of the function.

–fwritable-strings
Store string constants in the writable data segment and don't uniquize them. This is for compatibility with old programs which assume they can write into string constants. Writing into string constants is a very bad idea; constants should be constant.

–fcond-mismatch
Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void.

–fno-function-cse
Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

–fvolatile
Consider all memory references through pointers to be volatile.

–fshared-data
Requests that the data and non-**const** variables of this compilation be shared data rather than

private data. The distinction makes sense only on certain operating systems, where shared data is shared between processes running the same program, while private data exists in one copy per process.

–funsigned-char

Let the type **char** be the unsigned, like **unsigned char**.

Each kind of machine has a default for what **char** should be. It is either like **unsigned char** by default or like **signed char** by default. (Actually, at present, the default is always signed.)

The type **char** is always a distinct type from either **signed char** or **unsigned char**, even though its behavior is always just like one of those two.

Note that this is equivalent to **–fno-signed-char**, which is the negative form of **–fsigned-char**.

–fsigned-char

Let the type **char** be signed, like **signed char**.

Note that this is equivalent to **–fno-unsigned-char**, which is the negative form of **–funsigned-char**.

–fdelayed-branch

If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

–ffixed-*reg*

Treat the register named *reg* as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role).

reg must be the name of a register. The register names accepted are machine-specific and are defined in the **REGISTER_NAMES** macro in the machine description macro file.

This flag does not have a negative form, because it specifies a three-way choice.

–fcall-used-*reg*

Treat the register named *reg* as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register REG.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

This flag does not have a negative form, because it specifies a three-way choice.

–fcall-saved-*reg*

Treat the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register *reg* if they use it.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

A different sort of disaster will result from the use of this flag for a register in which function values may be returned.

This flag does not have a negative form, because it specifies a three-way choice.

–d*letters*

Says to make debugging dumps at times specified by *letters*. Here are the possible letters:

r
: Dump after RTL generation.

j
: Dump after first jump optimization.

J
: Dump after last jump optimization.

s
: Dump after CSE (including the jump optimization that sometimes follows CSE).

L
: Dump after loop optimization.

f
: Dump after flow analysis.

c
: Dump after instruction combination.

l
: Dump after local register allocation.

g
: Dump after global register allocation.

d
: Dump after delayed branch scheduling.

m
: Print statistics on memory usage, at the end of the run.

−pedantic
Issue all the warnings demanded by strict ANSI standard C; reject all programs that use forbidden extensions.

Valid ANSI standard C programs should compile properly with or without this option (though a rare few will require **−ansi**). However, without this option, certain GNU extensions and traditional C features are supported as well. With this option, they are rejected. There is no reason to use this option; it exists only to satisfy pedants.

−pedantic does not cause warning messages for use of the alternate keywords whose names begin and end with **__**.

−static On Suns running version 4, this prevents linking with the shared libraries. (**−g** has the same effect.) These options control the C preprocessor, which is run on each C source file before actual compilation. If you use the '-E' option, nothing is done except C preprocessing. Some of these options make sense only together with '-E' because they request preprocessor output that is not suitable for actual compilation.

−C Tell the preprocessor not to discard comments. Used with the **−E** option.

−I*dir* Search directory *dir* for include files.

−I- Any directories specified with **−I** options before the **−I-** option are searched only for the case of **#include "***file***"**; they are not searched for **#include <***file***>**.

If additional directories are specified with **−I** options after the **−I-**, these directories are searched for all **#include** directives. (Ordinarily *all* **−I** directories are used this way.)

In addition, the **−I-** option inhibits the use of the current directory as the first search directory for **#include "***file***"**. Therefore, the current directory is searched only if it is requested explicitly with **−I.**. Specifying both **−I-** and **-I.** allows you to control precisely which directories are searched before the current one and which are searched after.

–nostdinc
Do not search the standard system directories for header files. Only the directories you have specified with **–I** options (and the current directory, if appropriate) are searched.

Between **–nostdinc** and **–I-**, you can eliminate all directories from the search path except those you specify.

–M Tell the preprocessor to output a rule suitable for **make***(1)* describing the dependencies of each source file. For each source file, the preprocessor outputs one **make**-rule whose target is the object file name for that source file and whose dependencies are all the files **#include**d in it. This rule may be a single line or may be continued with **\-newline** if it is long.

–**M** implies –**E**.

–MM Like **–M** but the output mentions only the user-header files included with **#include "***file***"**. System header files included with **#include <***file***>** are omitted.

–**MM** implies **–E**.

–D*macro*
Define macro *macro* with the empty string as its definition.

–D*macro=defn*
Define macro *macro* as *defn*.

–U*macro*
Undefine macro *macro*.

–trigraphs
Support ANSI C trigraphs. You don't want to know about this brain-damage. The **–ansi** option also has this effect.

FILES

file.c	C source file
file.s	assembly language file
file.o	object file
a.out	link edited output
/tmp/cc∗	temporary files
LIBDIR/gcc-cpp	preprocessor
LIBDIR/gcc-cc1	compiler
LIBDIR/gcc-gnulib	library needed by GCC on some machines
/lib/crt[01n].o	start-up routine
/lib/libc.a	standard C library, see *intro*(3)
/usr/include	standard directory for **#include** files
LIBDIR/gcc-include	standard gcc directory for **#include** files

LIBDIR is usually **/usr/local/lib**.

SEE ALSO

as(1), ld(1), adb(1), dbx(1), sdb(1).

BUGS

Bugs should be reported to **bug-gcc@prep.ai.mit.edu**. Bugs tend actually to be fixed if they can be isolated, so it is in your interest to report them in such a way that they can be easily reproduced.

COPYING

AUTHORS

See the GNU CC Manual for the contributors to GNU CC.