Next: Optimize Options, Previous: Static Analyzer Options, Up: Invoking GCC [Contents][Index]

3.10 Options for Debugging Your Program

To tell GCC to emit extra information for use by a debugger, in almost all cases you need only to add `-g` to your other options. Some debug formats can co-exist (like DWARF with CTF) when each of them is enabled explicitly by adding the respective command line option to your other options.

GCC allows you to use `-g` with `-O`. The shortcuts taken by optimized code may occasionally be surprising: 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 are already at hand; some statements may execute in different places because they have been moved out of loops. Nevertheless it is possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

If you are not using some other optimization option, consider using `-Og` (see Optimize Options) with `-g`. With no `-O` option at all, some compiler passes that collect information useful for debugging do not run at all, so that `-Og` may result in a better debugging experience.

`-g`

> Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.
>
> On most systems that use stabs format, `-g` enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but probably makes other debuggers crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use `-gstabs+`, `-gstabs`, `-gxcoff+`, `-gxcoff`, or `-gvms` (see below).

`-ggdb`

> Produce debugging information for use by GDB. This means to use the most expressive format available (DWARF, stabs, or the native format if neither of those are supported), including GDB extensions if at all possible.

`-gdwarf`
`-gdwarf-`*`version`*

> Produce debugging information in DWARF format (if that is supported). The value of *version* may be either 2, 3, 4 or 5; the default version for most targets is 5 (with the exception of VxWorks, TPF and Darwin/Mac OS X, which default to version 2, and AIX, which defaults to version 4).
>
> Note that with DWARF Version 2, some ports require and always use some non-conflicting DWARF 3 extensions in the unwind tables.
>
> Version 4 may require GDB 7.0 and `-fvar-tracking-assignments` for maximum benefit. Version 5 requires GDB 8.0 or higher.

GCC no longer supports DWARF Version 1, which is substantially different than Version 2 and later. For historical reasons, some other DWARF-related options such as `-fno-dwarf2-cfi-asm`) retain a reference to DWARF Version 2 in their names, but apply to all currently-supported versions of DWARF.

`-gbtf`

Request BTF debug information. BTF is the default debugging format for the eBPF target. On other targets, like x86, BTF debug information can be generated along with DWARF debug information when both of the debug formats are enabled explicitly via their respective command line options.

`-gctf`
`-gctf`*level*

Request CTF debug information and use level to specify how much CTF debug information should be produced. If `-gctf` is specified without a value for level, the default level of CTF debug information is 2.

CTF debug information can be generated along with DWARF debug information when both of the debug formats are enabled explicitly via their respective command line options.

Level 0 produces no CTF debug information at all. Thus, `-gctf0` negates `-gctf`.

Level 1 produces CTF information for tracebacks only. This includes callsite information, but does not include type information.

Level 2 produces type information for entities (functions, data objects etc.) at file-scope or global-scope only.

`-gstabs`

Produce debugging information in stabs format (if that is supported), without GDB extensions. This is the format used by DBX on most BSD systems. On MIPS, Alpha and System V Release 4 systems this option produces stabs debugging output that is not understood by DBX. On System V Release 4 systems this option requires the GNU assembler.

`-gstabs+`

Produce debugging information in stabs format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program.

`-gxcoff`

Produce debugging information in XCOFF format (if that is supported). This is the format used by the DBX debugger on IBM RS/6000 systems.

`-gxcoff+`

Produce debugging information in XCOFF format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program, and may cause assemblers other than the GNU assembler (GAS) to fail with an error.

`-gvms`

Produce debugging information in Alpha/VMS debug format (if that is supported). This is the format used by DEBUG on Alpha/VMS systems.

`-g`*level*
`-ggdb`*level*
`-gstabs`*level*
`-gxcoff`*level*
`-gvms`*level*

Request debugging information and also use *level* to specify how much information. The default level is 2.

Level 0 produces no debug information at all. Thus, `-g0` negates `-g`.

Level 1 produces minimal information, enough for making backtraces in parts of the program that you don't plan to debug. This includes descriptions of functions and external variables, and line number tables, but no information about local variables.

Level 3 includes extra information, such as all the macro definitions present in the program. Some debuggers support macro expansion when you use `-g3`.

If you use multiple `-g` options, with or without level numbers, the last such option is the one that is effective.

`-gdwarf` does not accept a concatenated debug level, to avoid confusion with `-gdwarf-`*level*. Instead use an additional `-g`*level* option to change the debug level for DWARF.

`-fno-eliminate-unused-debug-symbols`

By default, no debug information is produced for symbols that are not actually used. Use this option if you want debug information for all symbols.

`-femit-class-debug-always`

Instead of emitting debugging information for a C++ class in only one object file, emit it in all object files using the class. This option should be used only with debuggers that are unable to handle the way GCC normally emits debugging information for classes because using this option increases the size of debugging information by as much as a factor of two.

`-fno-merge-debug-strings`

Direct the linker to not merge together strings in the debugging information that are identical in different object files. Merging is not supported by all assemblers or linkers. Merging decreases the size of the debug information in the output file at the cost of increasing link processing time. Merging is enabled by default.

`-fdebug-prefix-map=`*old*`=`*new*

When compiling files residing in directory *old*, record debugging information describing them as if the files resided in directory *new* instead. This can be used to replace a build-time path with an install-time path in the debug info. It can also be used to change an absolute path to a relative path by using . for *new*.

This can give more reproducible builds, which are location independent, but may require an extra command to tell GDB where to find the source files. See also `-ffile-prefix-map`.

`-fvar-tracking`

Run variable tracking pass. It computes where variables are stored at each position in code. Better debugging information is then generated (if the debugging information format supports this information).

It is enabled by default when compiling with optimization (`-Os`, `-O`, `-O2`, …), debugging information (`-g`) and the debug info format supports it.

`-fvar-tracking-assignments`

Annotate assignments to user variables early in the compilation and attempt to carry the annotations over throughout the compilation all the way to the end, in an attempt to improve debug information while optimizing. Use of `-gdwarf-4` is recommended along with it.

It can be enabled even if var-tracking is disabled, in which case annotations are created and maintained, but discarded at the end. By default, this flag is enabled together with `-fvar-tracking`, except when selective scheduling is enabled.

`-gsplit-dwarf`

If DWARF debugging information is enabled, separate as much debugging information as possible into a separate output file with the extension `.dwo`. This option allows the build system to avoid linking files with debug information. To be useful, this option requires a debugger capable of reading `.dwo` files.

`-gdwarf32`
`-gdwarf64`

If DWARF debugging information is enabled, the `-gdwarf32` selects the 32-bit DWARF format and the `-gdwarf64` selects the 64-bit DWARF format. The default is target specific, on most targets it is `-gdwarf32` though. The 32-bit DWARF format is smaller, but can't support more than 2GiB of debug information in any of the DWARF debug information sections. The 64-bit DWARF format allows larger debug information and might not be well supported by all consumers yet.

`-gdescribe-dies`

Add description attributes to some DWARF DIEs that have no name attribute, such as artificial variables, external references and call site parameter DIEs.

`-gpubnames`

Generate DWARF `.debug_pubnames` and `.debug_pubtypes` sections.

`-ggnu-pubnames`

Generate `.debug_pubnames` and `.debug_pubtypes` sections in a format suitable for conversion into a GDB index. This option is only useful with a linker that can produce GDB index version 7.

`-fdebug-types-section`

When using DWARF Version 4 or higher, type DIEs can be put into their own `.debug_types` section instead of making them part of the `.debug_info` section. It is more efficient to put them in a separate comdat section since the linker can then remove duplicates. But not all DWARF consumers support `.debug_types` sections yet and on some objects `.debug_types` produces larger instead of smaller debugging information.

`-grecord-gcc-switches`
`-gno-record-gcc-switches`

This switch causes the command-line options used to invoke the compiler that may affect code generation to be appended to the DW_AT_producer attribute in DWARF debugging information. The options are concatenated with spaces separating them from each other and from the compiler version. It is enabled by default. See also `-frecord-gcc-switches` for another way of storing compiler options into the object file.

`-gstrict-dwarf`

Disallow using extensions of later DWARF standard version than selected with `-gdwarf-`*`version`*. On most targets using non-conflicting DWARF extensions from later standard versions is allowed.

`-gno-strict-dwarf`

Allow using extensions of later DWARF standard version than selected with `-gdwarf-`*`version`*.

`-gas-loc-support`

Inform the compiler that the assembler supports `.loc` directives. It may then use them for the assembler to generate DWARF2+ line number tables.

This is generally desirable, because assembler-generated line-number tables are a lot more compact than those the compiler can generate itself.

This option will be enabled by default if, at GCC configure time, the assembler was found to support such directives.

`-gno-as-loc-support`

Force GCC to generate DWARF2+ line number tables internally, if DWARF2+ line number tables are to be generated.

`-gas-locview-support`

Inform the compiler that the assembler supports `view` assignment and reset assertion checking in `.loc` directives.

This option will be enabled by default if, at GCC configure time, the assembler was found to support them.

`-gno-as-locview-support`

Force GCC to assign view numbers internally, if `-gvariable-location-views` are explicitly requested.

`-gcolumn-info`
`-gno-column-info`

Emit location column information into DWARF debugging information, rather than just file and line. This option is enabled by default.

`-gstatement-frontiers`
`-gno-statement-frontiers`

This option causes GCC to create markers in the internal representation at the beginning of statements, and to keep them roughly in place throughout compilation, using them to guide the output of `is_stmt` markers in the line number table. This is enabled by default when compiling with optimization (`-Os`, `-O1`, `-O2`, …), and outputting DWARF 2 debug information at the normal level.

`-gvariable-location-views`
`-gvariable-location-views=incompat5`
`-gno-variable-location-views`

Augment variable location lists with progressive view numbers implied from the line number table. This enables debug information consumers to inspect state at certain points of the program, even if no instructions associated with the corresponding source locations are present at that point. If the assembler lacks support for view numbers in line number tables, this will cause the compiler to emit the line number table, which generally makes them somewhat less compact. The augmented line number tables and location lists are fully backward-compatible, so they can be consumed by debug information consumers that are not aware of these augmentations, but they won't derive any benefit from them either.

This is enabled by default when outputting DWARF 2 debug information at the normal level, as long as there is assembler support, `-fvar-tracking-assignments` is enabled and `-gstrict-dwarf` is not. When assembler support is not available, this may still be enabled, but it will force GCC to output internal line number tables, and if `-ginternal-reset-location-views` is not enabled, that will most certainly lead to silently mismatching location views.

There is a proposed representation for view numbers that is not backward compatible with the location list format introduced in DWARF 5, that can be enabled with `-gvariable-location-views=incompat5`. This option may be removed in the future, is only provided as a reference implementation of the proposed representation. Debug information consumers are not expected to support this extended format, and they would be rendered unable to decode location lists using it.

`-ginternal-reset-location-views`
`-gno-internal-reset-location-views`

Attempt to determine location views that can be omitted from location view lists. This requires the compiler to have very accurate insn length estimates, which isn't always the case, and it may cause incorrect view lists to be generated silently when using an assembler that does not support location view lists. The GNU assembler will flag any such error as a `view number mismatch`. This is only enabled on ports that define a reliable estimation function.

`-ginline-points`
`-gno-inline-points`

Generate extended debug information for inlined functions. Location view tracking markers are inserted at inlined entry points, so that address and view numbers can be computed and output in debug information. This can be enabled independently of location views, in which case the view numbers won't be output, but it can only be enabled along with statement frontiers, and it is only enabled by default if location views are enabled.

`-gz`[`=`*type*]

Produce compressed debug sections in DWARF format, if that is supported. If *type* is not given, the default type depends on the capabilities of the assembler and linker used. *type* may be one of '`none`' (don't compress debug sections), '`zlib`' (use zlib compression in ELF gABI format), or '`zlib-gnu`' (use zlib compression in traditional GNU format). If the linker doesn't support writing compressed debug sections, the option is rejected. Otherwise, if the assembler does not support them, `-gz` is silently ignored when producing object files.

`-femit-struct-debug-baseonly`

Emit debug information for struct-like types only when the base name of the compilation source file matches the base name of file in which the struct is defined.

This option substantially reduces the size of debugging information, but at significant potential loss in type information to the debugger. See `-femit-struct-debug-reduced` for a less aggressive option. See `-femit-struct-debug-detailed` for more detailed control.

This option works only with DWARF debug output.

`-femit-struct-debug-reduced`

Emit debug information for struct-like types only when the base name of the compilation source file matches the base name of file in which the type is defined, unless the struct is a template or defined in a system header.

This option significantly reduces the size of debugging information, with some potential loss in type information to the debugger. See `-femit-struct-debug-baseonly` for a more aggressive option. See `-femit-struct-debug-detailed` for more detailed control.

This option works only with DWARF debug output.

`-femit-struct-debug-detailed`[`=`*spec-list*]

Specify the struct-like types for which the compiler generates debug information. The intent is to reduce duplicate struct debug information between different object files within the same program.

This option is a detailed version of `-femit-struct-debug-reduced` and `-femit-struct-debug-baseonly`, which serves for most needs.

A specification has the syntax
[`'dir:'`|`'ind:'`][`'ord:'`|`'gen:'`](`'any'`|`'sys'`|`'base'`|`'none'`)

The optional first word limits the specification to structs that are used directly (‘dir:’) or used indirectly (‘ind:’). A struct type is used directly when it is the type of a variable, member. Indirect uses arise through pointers to structs. That is, when use of an incomplete struct is valid, the use is indirect. An example is ‘struct one direct; struct two * indirect;’.

The optional second word limits the specification to ordinary structs (‘ord:’) or generic structs (‘gen:’). Generic structs are a bit complicated to explain. For C++, these are non-explicit specializations of template classes, or non-template classes within the above. Other programming languages have generics, but -femit-struct-debug-detailed does not yet implement them.

The third word specifies the source files for those structs for which the compiler should emit debug information. The values ‘none’ and ‘any’ have the normal meaning. The value ‘base’ means that the base of name of the file in which the type declaration appears must match the base of the name of the main compilation file. In practice, this means that when compiling foo.c, debug information is generated for types declared in that file and foo.h, but not other header files. The value ‘sys’ means those types satisfying ‘base’ or declared in system or compiler headers.

You may need to experiment to determine the best settings for your application.

The default is -femit-struct-debug-detailed=all.

This option works only with DWARF debug output.

-fno-dwarf2-cfi-asm

Emit DWARF unwind info as compiler generated .eh_frame section instead of using GAS .cfi_* directives.

-fno-eliminate-unused-debug-types

Normally, when producing DWARF output, GCC avoids producing debug symbol output for types that are nowhere used in the source file being compiled. Sometimes it is useful to have GCC emit debugging information for all types declared in a compilation unit, regardless of whether or not they are actually used in that compilation unit, for example if, in the debugger, you want to cast a value to a type that is not actually used in your program (but is declared). More often, however, this results in a significant amount of wasted space.

Next: Optimize Options, Previous: Static Analyzer Options, Up: Invoking GCC
[Contents][Index]