3.8 Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions that are not inherently erroneous but that are risky or suggest there may have been an error.

The following language-independent options do not enable specific warnings but control the kinds of diagnostics produced by GCC.

`-fsyntax-only`

Check the code for syntax errors, but don't do anything beyond that.

`-fmax-errors=`*n*

Limits the maximum number of error messages to *n*, at which point GCC bails out rather than attempting to continue processing the source code. If *n* is 0 (the default), there is no limit on the number of error messages produced. If `-Wfatal-errors` is also specified, then `-Wfatal-errors` takes precedence over this option.

`-w`

Inhibit all warning messages.

`-Werror`

Make all warnings into errors.

`-Werror=`

Make the specified warning into an error. The specifier for a warning is appended; for example `-Werror=switch` turns the warnings controlled by `-Wswitch` into errors. This switch takes a negative form, to be used to negate `-Werror` for specific warnings; for example `-Wno-error=switch` makes `-Wswitch` warnings not be errors, even when `-Werror` is in effect.

The warning message for each controllable warning includes the option that controls the warning. That option can then be used with `-Werror=` and `-Wno-error=` as described above. (Printing of the option in the warning message can be disabled using the `-fno-diagnostics-show-option`

flag.)

Note that specifying `-Werror=`*foo* automatically implies `-W`*foo*. However, `-Wno-error=`*foo* does not imply anything.

`-Wfatal-errors`

This option causes the compiler to abort compilation on the first error occurred rather than trying to keep going and printing further error messages.

You can request many specific warnings with options beginning with '`-W`', for example `-Wimplicit` to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning '`-Wno-`' to turn off warnings; for example, `-Wno-implicit`. This manual lists only one of the two forms, whichever is not the default. For further language-specific options also refer to Options Controlling C++ Dialect and Options Controlling Objective-C and Objective-C++ Dialects. Additional warnings can be produced by enabling the static analyzer; See Options That Control Static Analysis.

Some options, such as `-Wall` and `-Wextra`, turn on other options, such as `-Wunused`, which may turn on further options, such as `-Wunused-value`. The combined effect of positive and negative forms is that more specific options have priority over less specific ones, independently of their position in the command-line. For options of the same specificity, the last one takes effect. Options enabled or disabled via pragmas (see Diagnostic Pragmas) take effect as if they appeared at the end of the command-line.

When an unrecognized warning option is requested (e.g., `-Wunknown-warning`), GCC emits a diagnostic stating that the option is not recognized. However, if the `-Wno-` form is used, the behavior is slightly different: no diagnostic is produced for `-Wno-unknown-warning` unless other diagnostics are being produced. This allows the use of new `-Wno-` options with old compilers, but if something goes wrong, the compiler warns that an unrecognized option is present.

The effectiveness of some warnings depends on optimizations also being enabled. For example `-Wsuggest-final-types` is more effective with link-time optimization and some instances of other warnings may not be issued at all unless optimization is enabled. While optimization in general improves the efficacy of control and data flow sensitive warnings, in some cases it may also cause false positives.

`-Wpedantic`
`-pedantic`

Issue all the warnings demanded by strict ISO C and ISO C++; diagnose all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. This follows the version of the ISO C or C++ standard specified by any `-std` option used.

Valid ISO C and ISO C++ programs should compile properly with or without this option (though a rare few require `-ansi` or a `-std` option specifying the version of the standard). However, without this option, certain GNU extensions and traditional C and C++ features are supported as well. With this option, they are diagnosed (or rejected with `-pedantic-errors`).

`-Wpedantic` does not cause warning messages for use of the alternate keywords whose names begin and end with ‘__’. This alternate format can also be used to disable warnings for non-ISO ‘__intN’ types, i.e. ‘__intN__’. Pedantic warnings are also disabled in the expression that follows __extension__. However, only system header files should use these escape routes; application programs should avoid them. See Alternate Keywords.

Some warnings about non-conforming programs are controlled by options other than `-Wpedantic`; in many cases they are implied by `-Wpedantic` but can be disabled separately by their specific option, e.g. `-Wpedantic -Wno-pointer-sign`.

Where the standard specified with `-std` represents a GNU extended dialect of C, such as ‘gnu90’ or ‘gnu99’, there is a corresponding *base standard*, the version of ISO C on which the GNU extended dialect is based. Warnings from `-Wpedantic` are given where they are required by the base standard. (It does not make sense for such warnings to be given only for features not in the specified GNU C dialect, since by definition the GNU dialects of C include all features the compiler supports with the given option, and there would be nothing to warn about.)

`-pedantic-errors`

Give an error whenever the *base standard* (see `-Wpedantic`) requires a diagnostic, in some cases where there is undefined behavior at compile-time and in some other cases that do not prevent compilation of programs that are valid according to the standard. This is not equivalent to `-Werror=pedantic`: the latter option is unlikely to be useful, as it only makes errors of the diagnostics that are controlled by `-Wpedantic`, whereas this option also affects required diagnostics that are always enabled or controlled by options other than `-Wpedantic`.

If you want the required diagnostics that are warnings by default to be errors instead, but don’t also want to enable the `-Wpedantic` diagnostics, you can specify `-pedantic-errors -Wno-pedantic` (or `-pedantic-errors -Wno-error=pedantic` to enable them but only as warnings).

Some required diagnostics are errors by default, but can be reduced to warnings using `-fpermissive` or their specific warning option, e.g. `-Wno-error=narrowing`.

Some diagnostics for non-ISO practices are controlled by specific warning options other than `-Wpedantic`, but are also made errors by `-pedantic-errors`. For instance:

```
-Wattributes (for standard attributes)
-Wchanges-meaning (C++)
-Wcomma-subscript (C++23 or later)
-Wdeclaration-after-statement (C90 or earlier)
-Welaborated-enum-base (C++11 or later)
-Wimplicit-int (C99 or later)
-Wimplicit-function-declaration (C99 or later)
-Wincompatible-pointer-types
```

`-Wint-conversion`
`-Wlong-long` (C90 or earlier)
`-Wmain`
`-Wnarrowing` (C++11 or later)
`-Wpointer-arith`
`-Wpointer-sign`
`-Wincompatible-pointer-types`
`-Wregister` (C++17 or later)
`-Wvla` (C90 or earlier)
`-Wwrite-strings` (C++11 or later)

`-fpermissive`

Downgrade some required diagnostics about nonconformant code from errors to warnings. Thus, using `-fpermissive` allows some nonconforming code to compile. Some C++ diagnostics are controlled only by this flag, but it also downgrades some C and C++ diagnostics that have their own flag:

`-Wdeclaration-missing-parameter-type` (C and Objective-C only)
`-Wimplicit-function-declaration` (C and Objective-C only)
`-Wimplicit-int` (C and Objective-C only)
`-Wincompatible-pointer-types` (C and Objective-C only)
`-Wint-conversion` (C and Objective-C only)
`-Wnarrowing` (C++ and Objective-C++ only)
`-Wreturn-mismatch` (C and Objective-C only)

The `-fpermissive` option is the default for historic C language modes (`-std=c89`, `-std=gnu89`, `-std=c90`, `-std=gnu90`).

`-Wall`

This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros. This also enables some language-specific warnings described in Options Controlling C++ Dialect and Options Controlling Objective-C and Objective-C++ Dialects.

`-Wall` turns on the following warning flags:

```
-Waddress
-Waligned-new (C++ and Objective-C++ only)
-Warray-bounds=1 (only with -O2)
-Warray-compare
-Warray-parameter=2
-Wbool-compare
-Wbool-operation
-Wc++11-compat  -Wc++14-compat  -Wc++17compat  -Wc++20compat
-Wcatch-value (C++ and Objective-C++ only)
-Wchar-subscripts
-Wclass-memaccess (C++ and Objective-C++ only)
-Wcomment
-Wdangling-else
-Wdangling-pointer=2
-Wdelete-non-virtual-dtor (C++ and Objective-C++ only)
-Wduplicate-decl-specifier (C and Objective-C only)
-Wenum-compare (in C/ObjC; this is on by default in C++)
-Wenum-int-mismatch (C and Objective-C only)
-Wformat=1
-Wformat-contains-nul
-Wformat-diag
-Wformat-extra-args
-Wformat-overflow=1
-Wformat-truncation=1
-Wformat-zero-length
-Wframe-address
-Wimplicit (C and Objective-C only)
-Wimplicit-function-declaration (C and Objective-C only)
-Wimplicit-int (C and Objective-C only)
-Winfinite-recursion
-Winit-self (C++ and Objective-C++ only)
-Wint-in-bool-context
-Wlogical-not-parentheses
-Wmain (only for C/ObjC and unless -ffreestanding)
-Wmaybe-uninitialized
-Wmemset-elt-size
-Wmemset-transposed-args
-Wmisleading-indentation (only for C/C++)
-Wmismatched-dealloc
```

```
-Wmismatched-new-delete (C++ and Objective-C++ only)
-Wmissing-attributes
-Wmissing-braces (only for C/ObjC)
-Wmultistatement-macros
-Wnarrowing (C++ and Objective-C++ only)
-Wnonnull
-Wnonnull-compare
-Wopenmp-simd (C and C++ only)
-Woverloaded-virtual=1 (C++ and Objective-C++ only)
-Wpacked-not-aligned
-Wparentheses
-Wpessimizing-move (C++ and Objective-C++ only)
-Wpointer-sign (only for C/ObjC)
-Wrange-loop-construct (C++ and Objective-C++ only)
-Wreorder (C++ and Objective-C++ only)
-Wrestrict
-Wreturn-type
-Wself-move (C++ and Objective-C++ only)
-Wsequence-point
-Wsign-compare (C++ and Objective-C++ only)
-Wsizeof-array-div
-Wsizeof-pointer-div
-Wsizeof-pointer-memaccess
-Wstrict-aliasing
-Wstrict-overflow=1
-Wswitch
-Wtautological-compare
-Wtrigraphs
-Wuninitialized
-Wunknown-pragmas
-Wunused
-Wunused-but-set-variable
-Wunused-const-variable=1 (only for C/ObjC)
-Wunused-function
-Wunused-label
-Wunused-local-typedefs
-Wunused-value
-Wunused-variable
-Wuse-after-free=2
-Wvla-parameter
```

```
-Wvolatile-register-var
-Wzero-length-bounds
```

Note that some warning flags are not implied by `-Wall`. Some of them warn about constructions that users generally do not consider questionable, but which occasionally you might wish to check for; others warn about constructions that are necessary or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning. Some of them are enabled by `-Wextra` but many of them must be enabled individually.

`-Wextra`

This enables some extra warning flags that are not enabled by `-Wall`. (This option used to be called `-W`. The older name is still supported, but the newer name is more descriptive.)

```
-Wabsolute-value (only for C/ObjC)
-Walloc-size
-Wcalloc-transposed-args
-Wcast-function-type
-Wclobbered
-Wdeprecated-copy (C++ and Objective-C++ only)
-Wempty-body
-Wenum-conversion (only for C/ObjC)
-Wexpansion-to-defined
-Wignored-qualifiers (only for C/C++)
-Wimplicit-fallthrough=3
-Wmaybe-uninitialized
-Wmissing-field-initializers
-Wmissing-parameter-type (C/ObjC only)
-Wold-style-declaration (C/ObjC only)
-Woverride-init (C/ObjC only)
-Wredundant-move (C++ and Objective-C++ only)
-Wshift-negative-value (in C++11 to C++17 and in C99 and newer)
-Wsign-compare (C++ and Objective-C++ only)
-Wsized-deallocation (C++ and Objective-C++ only)
-Wstring-compare
-Wtype-limits
-Wuninitialized
```

`-Wunused-parameter` (only with `-Wunused` or `-Wall`)
`-Wunused-but-set-parameter` (only with `-Wunused` or `-Wall`)

The option `-Wextra` also prints warning messages for the following cases:

- A pointer is compared against integer zero with `<`, `<=`, `>`, or `>=`.
- (C++ only) An enumerator and a non-enumerator both appear in a conditional expression.
- (C++ only) Ambiguous virtual bases.
- (C++ only) Subscripting an array that has been declared `register`.
- (C++ only) Taking the address of a variable that has been declared `register`.
- (C++ only) A base class is not initialized in the copy constructor of a derived class.

`-Wabi` (C, Objective-C, C++ and Objective-C++ only)

Warn about code affected by ABI changes. This includes code that may not be compatible with the vendor-neutral C++ ABI as well as the psABI for the particular target.

Since G++ now defaults to updating the ABI with each major release, normally `-Wabi` warns only about C++ ABI compatibility problems if there is a check added later in a release series for an ABI issue discovered since the initial release. `-Wabi` warns about more things if an older ABI version is selected (with `-fabi-version=`*n*).

`-Wabi` can also be used with an explicit version number to warn about C++ ABI compatibility with a particular `-fabi-version` level, e.g. `-Wabi=2` to warn about changes relative to `-fabi-version=2`.

If an explicit version number is provided and `-fabi-compat-version` is not specified, the version number from this option is used for compatibility aliases. If no explicit version number is provided with this option, but `-fabi-compat-version` is specified, that version number is used for C++ ABI warnings.

Although an effort has been made to warn about all such cases, there are probably some cases that are not warned about, even though G++ is generating incompatible code. There may also be cases where warnings are emitted even though the code that is generated is compatible.

You should rewrite your code to avoid these warnings if you are concerned about the fact that code generated by G++ may not be binary compatible with code generated by other compilers.

Known incompatibilities in `-fabi-version=2` (which was the default from GCC 3.4 to 4.9) include:

- A template with a non-type template parameter of reference type was mangled incorrectly:

```
extern int N;
template <int &> struct S {};
void n (S<N>) {2}
```

This was fixed in `-fabi-version=3`.

- SIMD vector types declared using `__attribute ((vector_size))` were mangled in a non-standard way that does not allow for overloading of functions taking vectors of different sizes.

  The mangling was changed in `-fabi-version=4`.

- `__attribute ((const))` and `noreturn` were mangled as type qualifiers, and `decltype` of a plain declaration was folded away.

  These mangling issues were fixed in `-fabi-version=5`.

- Scoped enumerators passed as arguments to a variadic function are promoted like unscoped enumerators, causing `va_arg` to complain. On most targets this does not actually affect the parameter passing ABI, as there is no way to pass an argument smaller than `int`.

  Also, the ABI changed the mangling of template argument packs, `const_cast`, `static_cast`, prefix increment/decrement, and a class scope function used as a template argument.

  These issues were corrected in `-fabi-version=6`.

- Lambdas in default argument scope were mangled incorrectly, and the ABI changed the mangling of `nullptr_t`.

  These issues were corrected in `-fabi-version=7`.

- When mangling a function type with function-cv-qualifiers, the un-qualified function type was incorrectly treated as a substitution candidate.

  This was fixed in `-fabi-version=8`, the default for GCC 5.1.

- `decltype(nullptr)` incorrectly had an alignment of 1, leading to unaligned accesses. Note that this did not affect the ABI of a function with a `nullptr_t` parameter, as parameters have a minimum alignment.

  This was fixed in `-fabi-version=9`, the default for GCC 5.2.

- Target-specific attributes that affect the identity of a type, such as ia32 calling conventions on a function type (stdcall, regparm, etc.), did not affect the mangled name, leading to name collisions when function pointers were used as template arguments.

This was fixed in `-fabi-version=10`, the default for GCC 6.1.

This option also enables warnings about psABI-related changes. The known psABI changes at this point include:

- For SysV/x86-64, unions with `long double` members are passed in memory as specified in psABI. Prior to GCC 4.4, this was not the case. For example:

```
union U {
  long double ld;
  int i;
};
```

`union U` is now always passed in memory.

`-Wno-changes-meaning` (C++ and Objective-C++ only)

C++ requires that unqualified uses of a name within a class have the same meaning in the complete scope of the class, so declaring the name after using it is ill-formed:

```
struct A;
struct B1 { A a; typedef A A; }; // warning, 'A' changes meaning
struct B2 { A a; struct A { }; }; // error, 'A' changes meaning
```

By default, the B1 case is only a warning because the two declarations have the same type, while the B2 case is an error. Both diagnostics can be disabled with `-Wno-changes-meaning`. Alternately, the error case can be reduced to a warning with `-Wno-error=changes-meaning` or `-fpermissive`.

Both diagnostics are also suppressed by `-fms-extensions`.

`-Wchar-subscripts`

Warn if an array subscript has type `char`. This is a common cause of error, as programmers often forget that this type is signed on some machines. This warning is enabled by `-Wall`.

`-Wno-coverage-mismatch`

Warn if feedback profiles do not match when using the `-fprofile-use` option. If a source file is changed between compiling with `-fprofile-generate` and with `-fprofile-use`, the files with the profile feedback can fail to match the source file and GCC cannot use the profile feedback

information. By default, this warning is enabled and is treated as an error. `-Wno-coverage-mismatch` can be used to disable the warning or `-Wno-error=coverage-mismatch` can be used to disable the error. Disabling the error for this warning can result in poorly optimized code and is useful only in the case of very minor changes such as bug fixes to an existing code-base. Completely disabling the warning is not recommended.

`-Wno-coverage-too-many-conditions`

Warn if `-fcondition-coverage` is used and an expression have too many terms and GCC gives up coverage. Coverage is given up when there are more terms in the conditional than there are bits in a `gcov_type_unsigned`. This warning is enabled by default.

`-Wno-coverage-invalid-line-number`

Warn in case a function ends earlier than it begins due to an invalid linenum macros. The warning is emitted only with `--coverage` enabled.

By default, this warning is enabled and is treated as an error. `-Wno-coverage-invalid-line-number` can be used to disable the warning or `-Wno-error=coverage-invalid-line-number` can be used to disable the error.

`-Wno-cpp` (C, Objective-C, C++, Objective-C++ and Fortran only)

Suppress warning messages emitted by `#warning` directives.

`-Wdouble-promotion` (C, C++, Objective-C and Objective-C++ only)

Give a warning when a value of type `float` is implicitly promoted to `double`. CPUs with a 32-bit "single-precision" floating-point unit implement `float` in hardware, but emulate `double` in software. On such a machine, doing computations using `double` values is much more expensive because of the overhead required for software emulation.

It is easy to accidentally do computations with `double` because floating-point literals are implicitly of type `double`. For example, in:

```
float area(float radius)
{
   return 3.14159 * radius * radius;
}
```

the compiler performs the entire computation with `double` because the floating-point literal is a `double`.

`-Wduplicate-decl-specifier` (C and Objective-C only)

Warn if a declaration has duplicate `const`, `volatile`, `restrict` or `_Atomic` specifier. This warning is enabled by `-Wall`.

`-Wformat`
`-Wformat=n`

Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified, and that the conversions specified in the format string make sense. This includes standard functions, and others specified by format attributes (see Declaring Attributes of Functions), in the `printf`, `scanf`, `strftime` and `strfmon` (an X/Open extension, not in the C standard) families (or other target-specific families). Which functions are checked without format attributes having been specified depends on the standard version selected, and such checks of functions without the attribute specified are disabled by `-ffreestanding` or `-fno-builtin`.

The formats are checked against the format features supported by GNU libc version 2.2. These include all ISO C90 and C99 features, as well as features from the Single Unix Specification and some BSD and GNU extensions. Other library implementations may not support all these features; GCC does not support warning about features that go beyond a particular library's limitations. However, if `-Wpedantic` is used with `-Wformat`, warnings are given about format features not in the selected standard version (but not for `strfmon` formats, since those are not in any version of the C standard). See Options Controlling C Dialect.

`-Wformat=1`
`-Wformat`

Option `-Wformat` is equivalent to `-Wformat=1`, and `-Wno-format` is equivalent to `-Wformat=0`. Since `-Wformat` also checks for null format arguments for several functions, `-Wformat` also implies `-Wnonnull`. Some aspects of this level of format checking can be disabled by the options: `-Wno-format-contains-nul`, `-Wno-format-extra-args`, and `-Wno-format-zero-length`. `-Wformat` is enabled by `-Wall`.

`-Wformat=2`

Enable `-Wformat` plus additional format checks. Currently equivalent to `-Wformat -Wformat-nonliteral -Wformat-security -Wformat-y2k`.

`-Wno-format-contains-nul`

If `-Wformat` is specified, do not warn about format strings that contain NUL bytes.

`-Wno-format-extra-args`

If `-Wformat` is specified, do not warn about excess arguments to a `printf` or `scanf` format function. The C standard specifies that such arguments are ignored.

Where the unused arguments lie between used arguments that are specified with '`$`' operand number specifications, normally warnings are still given, since the implementation could not know what type to pass to `va_arg` to skip the unused arguments. However, in the case of `scanf`

formats, this option suppresses the warning if the unused arguments are all pointers, since the Single Unix Specification says that such unused arguments are allowed.

`-Wformat-overflow`
`-Wformat-overflow=`*`level`*

Warn about calls to formatted input/output functions such as `sprintf` and `vsprintf` that might overflow the destination buffer. When the exact number of bytes written by a format directive cannot be determined at compile-time it is estimated based on heuristics that depend on the *level* argument and on optimization. While enabling optimization will in most cases improve the accuracy of the warning, it may also result in false positives.

`-Wformat-overflow`
`-Wformat-overflow=1`

Level *1* of `-Wformat-overflow` enabled by `-Wformat` employs a conservative approach that warns only about calls that most likely overflow the buffer. At this level, numeric arguments to format directives with unknown values are assumed to have the value of one, and strings of unknown length to be empty. Numeric arguments that are known to be bounded to a subrange of their type, or string arguments whose output is bounded either by their directive's precision or by a finite set of string literals, are assumed to take on the value within the range that results in the most bytes on output. For example, the call to `sprintf` below is diagnosed because even with both *a* and *b* equal to zero, the terminating NUL character (`'\0'`) appended by the function to the destination buffer will be written past its end. Increasing the size of the buffer by a single byte is sufficient to avoid the warning, though it may not be sufficient to avoid the overflow.

```
void f (int a, int b)
{
  char buf [13];
  sprintf (buf, "a = %i, b = %i\n", a, b);
}
```

`-Wformat-overflow=2`

Level *2* warns also about calls that might overflow the destination buffer given an argument of sufficient length or magnitude. At level *2*, unknown numeric arguments are assumed to have the minimum representable value for signed types with a precision greater than 1, and the maximum representable value otherwise. Unknown string arguments whose length cannot be assumed to be bounded either by the directive's precision, or by a finite set of string literals they may evaluate to, or the character array they may point to, are assumed to be 1 character long.

At level *2*, the call in the example above is again diagnosed, but this time because with *a* equal to a 32-bit `INT_MIN` the first `%i` directive will write some of its digits beyond the end of the destination buffer. To make the call safe regardless of the values of the two variables, the size of the destination buffer must be increased to at least 34 bytes. GCC includes the minimum size of the buffer in an informational note following the warning.

An alternative to increasing the size of the destination buffer is to constrain the range of formatted values. The maximum length of string arguments can be bounded by specifying the precision in the format directive. When numeric arguments of format directives can be assumed to be bounded by less than the precision of their type, choosing an appropriate length modifier to the format specifier will reduce the required buffer size. For example, if *a* and *b* in the example above can be assumed to be within the precision of the `short int` type then using either the `%hi` format directive or casting the argument to `short` reduces the maximum required size of the buffer to 24 bytes.

```
void f (int a, int b)
{
  char buf [23];
  sprintf (buf, "a = %hi, b = %i\n", a, (short)b);
}
```

`-Wno-format-zero-length`

If `-Wformat` is specified, do not warn about zero-length formats. The C standard specifies that zero-length formats are allowed.

`-Wformat-nonliteral`

If `-Wformat` is specified, also warn if the format string is not a string literal and so cannot be checked, unless the format function takes its format arguments as a `va_list`.

`-Wformat-security`

If `-Wformat` is specified, also warn about uses of format functions that represent possible security problems. At present, this warns about calls to `printf` and `scanf` functions where the format string is not a string literal and there are no format arguments, as in `printf (foo);`. This may be a security hole if the format string came from untrusted input and contains ‘`%n`’. (This is currently a subset of what `-Wformat-nonliteral` warns about, but in future warnings may be added to `-Wformat-security` that are not included in `-Wformat-nonliteral`.)

`-Wformat-signedness`

If `-Wformat` is specified, also warn if the format string requires an unsigned argument and the argument is signed and vice versa.

`-Wformat-truncation`
`-Wformat-truncation=`*`level`*

Warn about calls to formatted input/output functions such as `snprintf` and `vsnprintf` that might result in output truncation. When the exact number of bytes written by a format directive cannot be determined at compile-time it is estimated based on heuristics that depend on the *level* argument and on optimization. While enabling optimization will in most cases improve the accuracy of the warning, it may also result in false positives. Except as noted otherwise, the option uses the same logic `-Wformat-overflow`.

`-Wformat-truncation`
`-Wformat-truncation=1`

Level *1* of `-Wformat-truncation` enabled by `-Wformat` employs a conservative approach that warns only about calls to bounded functions whose return value is unused and that will most likely result in output truncation.

`-Wformat-truncation=2`

Level *2* warns also about calls to bounded functions whose return value is used and that might result in truncation given an argument of sufficient length or magnitude.

`-Wformat-y2k`

If `-Wformat` is specified, also warn about `strftime` formats that may yield only a two-digit year.

`-Wnonnull`

Warn about passing a null pointer for arguments marked as requiring a non-null value by the `nonnull` function attribute.

`-Wnonnull` is included in `-Wall` and `-Wformat`. It can be disabled with the `-Wno-nonnull` option.

`-Wnonnull-compare`

Warn when comparing an argument marked with the `nonnull` function attribute against null inside the function.

`-Wnonnull-compare` is included in `-Wall`. It can be disabled with the `-Wno-nonnull-compare` option.

`-Wnull-dereference`

Warn if the compiler detects paths that trigger erroneous or undefined behavior due to dereferencing a null pointer. This option is only active when `-fdelete-null-pointer-checks` is active, which is enabled by optimizations in most targets. The precision of the warnings depends on the

optimization options used.

`-Wnrvo` (C++ and Objective-C++ only)

Warn if the compiler does not elide the copy from a local variable to the return value of a function in a context where it is allowed by [class.copy.elision]. This elision is commonly known as the Named Return Value Optimization. For instance, in the example below the compiler cannot elide copies from both v1 and v2, so it elides neither.

```
std::vector<int> f()
{
  std::vector<int> v1, v2;
  // ...
  if (cond) return v1;
  else return v2; // warning: not eliding copy
}
```

`-Winfinite-recursion`

Warn about infinitely recursive calls. The warning is effective at all optimization levels but requires optimization in order to detect infinite recursion in calls between two or more functions. `-Winfinite-recursion` is included in `-Wall`.

Compare with `-Wanalyzer-infinite-recursion` which provides a similar diagnostic, but is implemented in a different way (as part of `-fanalyzer`).

`-Winit-self` (C, C++, Objective-C and Objective-C++ only)

Warn about uninitialized variables that are initialized with themselves. Note this option can only be used with the `-Wuninitialized` option.

For example, GCC warns about `i` being uninitialized in the following snippet only when `-Winit-self` has been specified:

```
int f()
{
  int i = i;
  return i;
}
```

This warning is enabled by `-Wall` in C++.

`-Wno-implicit-int` (C and Objective-C only)

This option controls warnings when a declaration does not specify a type. This warning is enabled by default, as an error, in C99 and later dialects of C, and also by `-Wall`. The error can be downgraded to a warning using `-fpermissive` (along with certain other errors), or for this error alone, with `-Wno-error=implicit-int`.

This warning is upgraded to an error by `-pedantic-errors`.

`-Wno-implicit-function-declaration` (C and Objective-C only)

This option controls warnings when a function is used before being declared. This warning is enabled by default, as an error, in C99 and later dialects of C, and also by `-Wall`. The error can be downgraded to a warning using `-fpermissive` (along with certain other errors), or for this error alone, with `-Wno-error=implicit-function-declaration`.

This warning is upgraded to an error by `-pedantic-errors`.

`-Wimplicit` (C and Objective-C only)

Same as `-Wimplicit-int` and `-Wimplicit-function-declaration`. This warning is enabled by `-Wall`.

`-Whardened`

Warn when `-fhardened` did not enable an option from its set (for which see `-fhardened`). For instance, using `-fhardened` and `-fstack-protector` at the same time on the command line causes `-Whardened` to warn because `-fstack-protector-strong` is not enabled by `-fhardened`.

This warning is enabled by default and has effect only when `-fhardened` is enabled.

`-Wimplicit-fallthrough`

`-Wimplicit-fallthrough` is the same as `-Wimplicit-fallthrough=3` and `-Wno-implicit-fallthrough` is the same as `-Wimplicit-fallthrough=0`.

`-Wimplicit-fallthrough=`*n*

Warn when a switch case falls through. For example:

```
switch (cond)
  {
  case 1:
    a = 1;
    break;
  case 2:
```

```
    a = 2;
  case 3:
    a = 3;
    break;
  }
```

This warning does not warn when the last statement of a case cannot fall through, e.g. when there is a return statement or a call to function declared with the noreturn attribute. `-Wimplicit-fallthrough=` also takes into account control flow statements, such as ifs, and only warns when appropriate. E.g.

```
switch (cond)
  {
  case 1:
    if (i > 3) {
      bar (5);
      break;
    } else if (i < 1) {
      bar (0);
    } else
      return;
  default:
    …
  }
```

Since there are occasions where a switch case fall through is desirable, GCC provides an attribute, `__attribute__ ((fallthrough))`, that is to be used along with a null statement to suppress this warning that would normally occur:

```
switch (cond)
  {
  case 1:
    bar (0);
    __attribute__ ((fallthrough));
  default:
    …
  }
```

C++17 provides a standard way to suppress the `-Wimplicit-fallthrough` warning using `[[fallthrough]];` instead of the GNU attribute. In C++11 or C++14 users can use `[[gnu::fallthrough]];`, which is a GNU extension. Instead of these attributes, it is also possible to add a

fallthrough comment to silence the warning. The whole body of the C or C++ style comment should match the given regular expressions listed below. The option argument *n* specifies what kind of comments are accepted:

- `-Wimplicit-fallthrough=0` disables the warning altogether.
- `-Wimplicit-fallthrough=1` matches `.*` regular expression, any comment is used as fallthrough comment.
- `-Wimplicit-fallthrough=2` case insensitively matches `.*falls?[ \t-]*thr(ough|u).*` regular expression.
- `-Wimplicit-fallthrough=3` case sensitively matches one of the following regular expressions:
  - `-fallthrough`
  - `@fallthrough@`
  - `lint -fallthrough[ \t]*`
  - `[ \t.!]*(ELSE,? |INTENTIONAL(LY)? )?`
    `FALL(S | |-)?THR(OUGH|U)[ \t.!]*(-[^\n\r]*)?`
  - `[ \t.!]*(Else,? |Intentional(ly)? )?`
    `Fall((s | |-)[Tt]|t)hr(ough|u)[ \t.!]*(-[^\n\r]*)?`
  - `[ \t.!]*([Ee]lse,? |[Ii]ntentional(ly)? )?`
    `fall(s | |-)?thr(ough|u)[ \t.!]*(-[^\n\r]*)?`
- `-Wimplicit-fallthrough=4` case sensitively matches one of the following regular expressions:
  - `-fallthrough`
  - `@fallthrough@`
  - `lint -fallthrough[ \t]*`
  - `[ \t]*FALLTHR(OUGH|U)[ \t]*`
- `-Wimplicit-fallthrough=5` doesn't recognize any comments as fallthrough comments, only attributes disable the warning.

The comment needs to be followed after optional whitespace and other comments by `case` or `default` keywords or by a user label that precedes some `case` or `default` label.

```
switch (cond)
  {
  case 1:
    bar (0);
    /* FALLTHRU */
  default:
    …
  }
```

The `-Wimplicit-fallthrough=3` warning is enabled by `-Wextra`.

`-Wno-if-not-aligned` (C, C++, Objective-C and Objective-C++ only)

Control if warnings triggered by the `warn_if_not_aligned` attribute should be issued. These warnings are enabled by default.

`-Wignored-qualifiers` (C and C++ only)

Warn if the return type of a function has a type qualifier such as `const`. For ISO C such a type qualifier has no effect, since the value returned by a function is not an lvalue. For C++, the warning is only emitted for scalar types or `void`. ISO C prohibits qualified `void` return types on function definitions, so such return types always receive a warning even without this option.

This warning is also enabled by `-Wextra`.

`-Wno-ignored-attributes` (C and C++ only)

This option controls warnings when an attribute is ignored. This is different from the `-Wattributes` option in that it warns whenever the compiler decides to drop an attribute, not that the attribute is either unknown, used in a wrong place, etc. This warning is enabled by default.

`-Wmain`

Warn if the type of `main` is suspicious. `main` should be a function with external linkage, returning int, taking either zero arguments, two, or three arguments of appropriate types. This warning is enabled by default in C++ and is enabled by either `-Wall` or `-Wpedantic`.

This warning is upgraded to an error by `-pedantic-errors`.

`-Wmisleading-indentation` (C and C++ only)

Warn when the indentation of the code does not reflect the block structure. Specifically, a warning is issued for `if`, `else`, `while`, and `for` clauses with a guarded statement that does not use braces, followed by an unguarded statement with the same indentation.

In the following example, the call to “bar” is misleadingly indented as if it were guarded by the “if” conditional.

```
if (some_condition ())
  foo ();
  bar ();  /* Gotcha: this is not guarded by the "if".  */
```

In the case of mixed tabs and spaces, the warning uses the `-ftabstop=` option to determine if the statements line up (defaulting to 8).

The warning is not issued for code involving multiline preprocessor logic such as the following example.

```
  if (flagA)
    foo (0);
#if SOME_CONDITION_THAT_DOES_NOT_HOLD
  if (flagB)
#endif
    foo (1);
```

The warning is not issued after a `#line` directive, since this typically indicates autogenerated code, and no assumptions can be made about the layout of the file that the directive references.

This warning is enabled by `-Wall` in C and C++.

`-Wmissing-attributes`

Warn when a declaration of a function is missing one or more attributes that a related function is declared with and whose absence may adversely affect the correctness or efficiency of generated code. For example, the warning is issued for declarations of aliases that use attributes to specify less restrictive requirements than those of their targets. This typically represents a potential optimization opportunity. By contrast, the `-Wattribute-alias=2` option controls warnings issued when the alias is more restrictive than the target, which could lead to incorrect code generation. Attributes considered include `alloc_align`, `alloc_size`, `cold`, `const`, `hot`, `leaf`, `malloc`, `nonnull`, `noreturn`, `nothrow`, `pure`, `returns_nonnull`, and `returns_twice`.

In C++, the warning is issued when an explicit specialization of a primary template declared with attribute `alloc_align`, `alloc_size`, `assume_aligned`, `format`, `format_arg`, `malloc`, or `nonnull` is declared without it. Attributes `deprecated`, `error`, and `warning` suppress the warning. (see Declaring Attributes of Functions).

You can use the `copy` attribute to apply the same set of attributes to a declaration as that on another declaration without explicitly enumerating the attributes. This attribute can be applied to declarations of functions (see Common Function Attributes), variables (see Common Variable Attributes), or types (see Common Type Attributes).

`-Wmissing-attributes` is enabled by `-Wall`.

For example, since the declaration of the primary function template below makes use of both attribute `malloc` and `alloc_size` the declaration of the explicit specialization of the template is diagnosed because it is missing one of the attributes.

```
template <class T>
T* __attribute__ ((malloc, alloc_size (1)))
allocate (size_t);
```

```
template <>
void* __attribute__ ((malloc))   // missing alloc_size
allocate<void> (size_t);
```

`-Wmissing-braces`

Warn if an aggregate or union initializer is not fully bracketed. In the following example, the initializer for `a` is not fully bracketed, but that for `b` is fully bracketed.

```
int a[2][2] = { 0, 1, 2, 3 };
int b[2][2] = { { 0, 1 }, { 2, 3 } };
```

This warning is enabled by `-Wall`.

`-Wmissing-include-dirs` (C, C++, Objective-C, Objective-C++ and Fortran only)

Warn if a user-supplied include directory does not exist. This option is disabled by default for C, C++, Objective-C and Objective-C++. For Fortran, it is partially enabled by default by warning for -I and -J, only.

`-Wno-missing-profile`

This option controls warnings if feedback profiles are missing when using the `-fprofile-use` option. This option diagnoses those cases where a new function or a new file is added between compiling with `-fprofile-generate` and with `-fprofile-use`, without regenerating the profiles. In these cases, the profile feedback data files do not contain any profile feedback information for the newly added function or file respectively. Also, in the case when profile count data (.gcda) files are removed, GCC cannot use any profile feedback information. In all these cases, warnings are issued to inform you that a profile generation step is due. Ignoring the warning can result in poorly optimized code. `-Wno-missing-profile` can be used to disable the warning, but this is not recommended and should be done only when non-existent profile data is justified.

`-Wmismatched-dealloc`

Warn for calls to deallocation functions with pointer arguments returned from allocation functions for which the former isn't a suitable deallocator. A pair of functions can be associated as matching allocators and deallocators by use of attribute `malloc`. Unless disabled by the `-fno-builtin` option the standard functions `calloc`, `malloc`, `realloc`, and `free`, as well as the corresponding forms of C++ `operator new` and `operator delete` are implicitly associated as matching allocators and deallocators. In the following example `mydealloc` is the deallocator for pointers returned from `myalloc`.

```
void mydealloc (void*);

__attribute__ ((malloc (mydealloc, 1))) void*
myalloc (size_t);

void f (void)
{
  void *p = myalloc (32);
  // …use p…
  free (p);   // warning: not a matching deallocator for myalloc
  mydealloc (p);   // ok
}
```

In C++, the related option `-Wmismatched-new-delete` diagnoses mismatches involving either `operator new` or `operator delete`.

Option `-Wmismatched-dealloc` is included in `-Wall`.

`-Wmultistatement-macros`

Warn about unsafe multiple statement macros that appear to be guarded by a clause such as `if`, `else`, `for`, `switch`, or `while`, in which only the first statement is actually guarded after the macro is expanded.

For example:

```
#define DOIT x++; y++
if (c)
  DOIT;
```

will increment `y` unconditionally, not just when `c` holds. The can usually be fixed by wrapping the macro in a do-while loop:

```
#define DOIT do { x++; y++; } while (0)
if (c)
  DOIT;
```

This warning is enabled by `-Wall` in C and C++.

`-Wparentheses`

Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often get confused about.

Also warn if a comparison like `x<=y<=z` appears; this is equivalent to `(x<=y ? 1 : 0) <= z`, which is a different interpretation from that of ordinary mathematical notation.

Also warn for dangerous uses of the GNU extension to `?:` with omitted middle operand. When the condition in the `?:` operator is a boolean expression, the omitted value is always 1. Often programmers expect it to be a value computed inside the conditional expression instead.

For C++ this also warns for some cases of unnecessary parentheses in declarations, which can indicate an attempt at a function call instead of a declaration:

```
{
  // Declares a local variable called mymutex.
  std::unique_lock<std::mutex> (mymutex);
  // User meant std::unique_lock<std::mutex> lock (mymutex);
}
```

This warning is enabled by `-Wall`.

`-Wno-self-move` (C++ and Objective-C++ only)

This warning warns when a value is moved to itself with `std::move`. Such a `std::move` typically has no effect.

```
struct T {
…
};
void fn()
{
  T t;
  …
  t = std::move (t);
}
```

This warning is enabled by `-Wall`.

`-Wsequence-point`

Warn about code that may have undefined semantics because of violations of sequence point rules in the C and C++ standards.

The C and C++ standards define the order in which expressions in a C/C++ program are evaluated in terms of *sequence points*, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point, and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a `&&`, `||`, `? :` or `,` (comma) operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places. Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order, since, for example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee have ruled that function calls do not overlap.

It is not specified when between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior; the C and C++ standards specify that “Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.”. If a program breaks these rules, the results on any particular implementation are entirely unpredictable.

Examples of code with undefined behavior are `a = a++;`, `a[n] = b[n++]` and `a[i++] = i;`. Some more complicated cases are not diagnosed by this option, and it may give an occasional false positive result, but in general it has been found fairly effective at detecting this sort of problem in programs.

The C++17 standard will define the order of evaluation of operands in more cases: in particular it requires that the right-hand side of an assignment be evaluated before the left-hand side, so the above examples are no longer undefined. But this option will still warn about them, to help people avoid writing code that is undefined in C and earlier revisions of C++.

The standard is worded confusingly, therefore there is some debate over the precise meaning of the sequence point rules in subtle cases. Links to discussions of the problem, including proposed formal definitions, may be found on the GCC readings page, at https://gcc.gnu.org/readings.html.

This warning is enabled by `-Wall` for C and C++.

`-Wno-return-local-addr`

Do not warn about returning a pointer (or in C++, a reference) to a variable that goes out of scope after the function returns.

`-Wreturn-mismatch`

Warn about return statements without an expressions in functions which do not return `void`. Also warn about a `return` statement with an expression in a function whose return type is `void`, unless the expression type is also `void`. As a GNU extension, the latter case is accepted without a warning unless `-Wpedantic` is used.

Attempting to use the return value of a non-`void` function other than `main` that flows off the end by reaching the closing curly brace that terminates the function is undefined.

This warning is specific to C and enabled by default. In C99 and later language dialects, it is treated as an error. It can be downgraded to a warning using `-fpermissive` (along with other warnings), or for just this warning, with `-Wno-error=return-mismatch`.

`-Wreturn-type`

Warn whenever a function is defined with a return type that defaults to `int` (unless `-Wimplicit-int` is active, which takes precedence). Also warn if execution may reach the end of the function body, or if the function does not contain any return statement at all.

Attempting to use the return value of a non-`void` function other than `main` that flows off the end by reaching the closing curly brace that terminates the function is undefined.

Unlike in C, in C++, flowing off the end of a non-`void` function other than `main` results in undefined behavior even when the value of the function is not used.

This warning is enabled by default in C++ and by `-Wall` otherwise.

`-Wno-shift-count-negative`

Controls warnings if a shift count is negative. This warning is enabled by default.

`-Wno-shift-count-overflow`

Controls warnings if a shift count is greater than or equal to the bit width of the type. This warning is enabled by default.

`-Wshift-negative-value`

Warn if left shifting a negative value. This warning is enabled by `-Wextra` in C99 (and newer) and C++11 to C++17 modes.

`-Wno-shift-overflow`
`-Wshift-overflow=`*n*

These options control warnings about left shift overflows.

`-Wshift-overflow=1`

This is the warning level of `-Wshift-overflow` and is enabled by default in C99 and C++11 modes (and newer). This warning level does not warn about left-shifting 1 into the sign bit. (However, in C, such an overflow is still rejected in contexts where an integer constant expression is required.) No warning is emitted in C++20 mode (and newer), as signed left shifts always wrap.

`-Wshift-overflow=2`

This warning level also warns about left-shifting 1 into the sign bit, unless C++14 mode (or newer) is active.

`-Wswitch`

Warn whenever a `switch` statement has an index of enumerated 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 (even if there is a `default` label). This warning is enabled by `-Wall`.

`-Wswitch-default`

Warn whenever a `switch` statement does not have a `default` case.

`-Wswitch-enum`

Warn whenever a `switch` statement has an index of enumerated type and lacks a `case` for one or more of the named codes of that enumeration. `case` labels outside the enumeration range also provoke warnings when this option is used. The only difference between `-Wswitch` and this option is that this option gives a warning about an omitted enumeration code even if there is a `default` label.

`-Wno-switch-bool`

Do not warn when a `switch` statement has an index of boolean type and the case values are outside the range of a boolean type. It is possible to suppress this warning by casting the controlling expression to a type other than `bool`. For example:

```
switch ((int) (a == 4))
  {
  …
  }
```

This warning is enabled by default for C and C++ programs.

`-Wno-switch-outside-range`

This option controls warnings when a `switch` case has a value that is outside of its respective type range. This warning is enabled by default for C and C++ programs.

`-Wno-switch-unreachable`

Do not warn when a `switch` statement contains statements between the controlling expression and the first case label, which will never be executed. For example:

```
switch (cond)
  {
   i = 15;
  …
   case 5:
  …
  }
```

`-Wswitch-unreachable` does not warn if the statement between the controlling expression and the first case label is just a declaration:

```
switch (cond)
  {
   int i;
  …
   case 5:
   i = 5;
  …
  }
```

This warning is enabled by default for C and C++ programs.

`-Wsync-nand` (C and C++ only)

Warn when `__sync_fetch_and_nand` and `__sync_nand_and_fetch` built-in functions are used. These functions changed semantics in GCC 4.4.

`-Wtrivial-auto-var-init`

Warn when `-ftrivial-auto-var-init` cannot initialize the automatic variable. A common situation is an automatic variable that is declared between the controlling expression and the first case label of a `switch` statement.

`-Wunused-but-set-parameter`

Warn whenever a function parameter is assigned to, but otherwise unused (aside from its declaration).

To suppress this warning use the `unused` attribute (see Specifying Attributes of Variables).

This warning is also enabled by `-Wunused` together with `-Wextra`.

`-Wunused-but-set-variable`

Warn whenever a local variable is assigned to, but otherwise unused (aside from its declaration). This warning is enabled by `-Wall`.

To suppress this warning use the `unused` attribute (see Specifying Attributes of Variables).

This warning is also enabled by `-Wunused`, which is enabled by `-Wall`.

`-Wunused-function`

Warn whenever a static function is declared but not defined or a non-inline static function is unused. This warning is enabled by `-Wall`.

`-Wunused-label`

Warn whenever a label is declared but not used. This warning is enabled by `-Wall`.

To suppress this warning use the `unused` attribute (see Specifying Attributes of Variables).

`-Wunused-local-typedefs` (C, Objective-C, C++ and Objective-C++ only)

Warn when a typedef locally defined in a function is not used. This warning is enabled by `-Wall`.

`-Wunused-parameter`

Warn whenever a function parameter is unused aside from its declaration. This option is not enabled by `-Wunused` unless `-Wextra` is also specified.

To suppress this warning use the `unused` attribute (see Specifying Attributes of Variables).

`-Wno-unused-result`

Do not warn if a caller of a function marked with attribute `warn_unused_result` (see Declaring Attributes of Functions) does not use its return value. The default is `-Wunused-result`.

`-Wunused-variable`

Warn whenever a local or static variable is unused aside from its declaration. This option implies `-Wunused-const-variable=1` for C, but not for C++. This warning is enabled by `-Wall`.

To suppress this warning use the `unused` attribute (see Specifying Attributes of Variables).

`-Wunused-const-variable`
`-Wunused-const-variable=`*n*

Warn whenever a constant static variable is unused aside from its declaration.

To suppress this warning use the `unused` attribute (see Specifying Attributes of Variables).

`-Wunused-const-variable=1`

Warn about unused static const variables defined in the main compilation unit, but not about static const variables declared in any header included.

`-Wunused-const-variable=1` is enabled by either `-Wunused-variable` or `-Wunused` for C, but not for C++. In C this declares variable storage, but in C++ this is not an error since const variables take the place of `#defines`.

`-Wunused-const-variable=2`

This warning level also warns for unused constant static variables in headers (excluding system headers). It is equivalent to the short form `-Wunused-const-variable`. This level must be explicitly requested in both C and C++ because it might be hard to clean up all headers included.

`-Wunused-value`

Warn whenever a statement computes a result that is explicitly not used. To suppress this warning cast the unused expression to `void`. This includes an expression-statement or the left-hand side of a comma expression that contains no side effects. For example, an expression such as `x[i,j]` causes a warning, while `x[(void)i,j]` does not.

This warning is enabled by `-Wall`.

`-Wunused`

All the above `-Wunused` options combined, except those documented as needing to be specified explicitly.

In order to get a warning about an unused function parameter, you must either specify `-Wextra -Wunused` (note that `-Wall` implies `-Wunused`), or separately specify `-Wunused-parameter` and/or `-Wunused-but-set-parameter`.

`-Wunused` enables only `-Wunused-const-variable=1` rather than `-Wunused-const-variable`, and only for C, not C++.

`-Wuse-after-free` (C, Objective-C, C++ and Objective-C++ only)
`-Wuse-after-free=`*n*

Warn about uses of pointers to dynamically allocated objects that have been rendered indeterminate by a call to a deallocation function. The warning is enabled at all optimization levels but may yield different results with optimization than without.

`-Wuse-after-free=1`

At level 1 the warning attempts to diagnose only unconditional uses of pointers made indeterminate by a deallocation call or a successful call to `realloc`, regardless of whether or not the call resulted in an actual reallocation of memory. This includes double-`free` calls as well as uses in arithmetic and relational expressions. Although undefined, uses of indeterminate pointers in equality (or inequality) expressions are not diagnosed at this level.

`-Wuse-after-free=2`

At level 2, in addition to unconditional uses, the warning also diagnoses conditional uses of pointers made indeterminate by a deallocation call. As at level 2, uses in equality (or inequality) expressions are not diagnosed. For example, the second call to `free` in the following function is diagnosed at this level:

```
struct A { int refcount; void *data; };

void release (struct A *p)
{
  int refcount = --p->refcount;
  free (p);
  if (refcount == 0)
    free (p->data);   // warning: p may be used after free
}
```

`-Wuse-after-free=3`

At level 3, the warning also diagnoses uses of indeterminate pointers in equality expressions. All uses of indeterminate pointers are undefined but equality tests sometimes appear after calls to `realloc` as an attempt to determine whether the call resulted in relocating the object to a different address. They are diagnosed at a separate level to aid gradually transitioning legacy code to safe alternatives. For example, the equality test in the function below is diagnosed at this level:

```
void adjust_pointers (int**, int);

void grow (int **p, int n)
{
  int **q = (int**)realloc (p, n *= 2);
  if (q == p)
    return;
  adjust_pointers ((int**)q, n);
}
```

To avoid the warning at this level, store offsets into allocated memory instead of pointers. This approach obviates needing to adjust the stored pointers after reallocation.

`-Wuse-after-free=2` is included in `-Wall`.

`-Wuseless-cast` (C, Objective-C, C++ and Objective-C++ only)

Warn when an expression is cast to its own type. This warning does not occur when a class object is converted to a non-reference type as that is a way to create a temporary:

```
struct S { };
void g (S&&);
void f (S&& arg)
{
  g (S(arg)); // make arg prvalue so that it can bind to S&&
}
```

`-Wuninitialized`

Warn if an object with automatic or allocated storage duration is used without having been initialized. In C++, also warn if a non-static reference or non-static `const` member appears in a class without constructors.

In addition, passing a pointer (or in C++, a reference) to an uninitialized object to a `const`-qualified argument of a built-in function known to read the object is also diagnosed by this warning. (`-Wmaybe-uninitialized` is issued for ordinary functions.)

If you want to warn about code that uses the uninitialized value of the variable in its own initializer, use the `-Winit-self` option.

These warnings occur for individual uninitialized elements of structure, union or array variables as well as for variables that are uninitialized as a whole. They do not occur for variables or elements declared `volatile`. Because these warnings depend on optimization, the exact variables or elements for which there are warnings depend on the precise optimization options and version of GCC used.

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.

In C++, this warning also warns about using uninitialized objects in member-initializer-lists. For example, GCC warns about `b` being uninitialized in the following snippet:

```
struct A {
  int a;
  int b;
  A() : a(b) { }
};
```

`-Wno-invalid-memory-model`

This option controls warnings for invocations of Built-in Functions for Memory Model Aware Atomic Operations, Legacy `__sync` Built-in Functions for Atomic Memory Access, and the C11 atomic generic functions with a memory consistency argument that is either invalid for the operation or outside the range of values of the `memory_order` enumeration. For example, since the `__atomic_store` and `__atomic_store_n` built-ins are only defined for the relaxed, release, and sequentially consistent memory orders the following code is diagnosed:

```
void store (int *i)
{
  __atomic_store_n (i, 0, memory_order_consume);
}
```

`-Winvalid-memory-model` is enabled by default.

`-Wmaybe-uninitialized`

For an object with automatic or allocated storage duration, if there exists a path from the function entry to a use of the object that is initialized, but there exist some other paths for which the object is not initialized, the compiler emits a warning if it cannot prove the uninitialized paths are not executed at run time.

In addition, passing a pointer (or in C++, a reference) to an uninitialized object to a `const`-qualified function argument is also diagnosed by this warning. (`-Wuninitialized` is issued for built-in functions known to read the object.) Annotating the function with attribute `access (none)` indicates that the argument isn't used to access the object and avoids the warning (see Common Function Attributes).

These warnings are only possible in optimizing compilation, because otherwise GCC does not keep track of the state of variables.

These warnings are made optional because GCC may not be able to determine when the code is correct in spite of 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 GCC doesn't know this. To suppress the warning, you need to provide a default case with assert(0) or similar code.

This option also warns when a non-volatile automatic variable might be changed by a call to `longjmp`. The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` 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` cannot in fact be called at the place that would cause a problem.

Some spurious warnings can be avoided if you declare all the functions you use that never return as `noreturn`. See Declaring Attributes of Functions.

This warning is enabled by `-Wall` or `-Wextra`.

`-Wunknown-pragmas`

Warn when a `#pragma` directive is encountered that is not understood by GCC. If this command-line option is used, warnings are even issued for unknown pragmas in system header files. This is not the case if the warnings are only enabled by the `-Wall` command-line option.

`-Wno-pragmas`

Do not warn about misuses of pragmas, such as incorrect parameters, invalid syntax, or conflicts between pragmas. See also `-Wunknown-pragmas`.

`-Wno-prio-ctor-dtor`

Do not warn if a priority from 0 to 100 is used for constructor or destructor. The use of constructor and destructor attributes allow you to assign a priority to the constructor/destructor to control its order of execution before `main` is called or after it returns. The priority values must be greater than 100 as the compiler reserves priority values between 0–100 for the implementation.

`-Wstrict-aliasing`

This option is only active when `-fstrict-aliasing` is active. It warns about code that might break the strict aliasing rules that the compiler is using for optimization. The warning does not catch all cases, but does attempt to catch the more common pitfalls. It is included in `-Wall`. It is equivalent to `-Wstrict-aliasing=3`

`-Wstrict-aliasing=n`

This option is only active when `-fstrict-aliasing` is active. It warns about code that might break the strict aliasing rules that the compiler is using for optimization. Higher levels correspond to higher accuracy (fewer false positives). Higher levels also correspond to more effort, similar to the way `-O` works. `-Wstrict-aliasing` is equivalent to `-Wstrict-aliasing=3`.

Level 1: Most aggressive, quick, least accurate. Possibly useful when higher levels do not warn but `-fstrict-aliasing` still breaks the code, as it has very few false negatives. However, it has many false positives. Warns for all pointer conversions between possibly incompatible types, even if never dereferenced. Runs in the front end only.

Level 2: Aggressive, quick, not too precise. May still have many false positives (not as many as level 1 though), and few false negatives (but possibly more than level 1). Unlike level 1, it only warns when an address is taken. Warns about incomplete types. Runs in the front end only.

Level 3 (default for `-Wstrict-aliasing`): Should have very few false positives and few false negatives. Slightly slower than levels 1 or 2 when optimization is enabled. Takes care of the common pun+dereference pattern in the front end: `*(int*)&some_float`. If optimization is enabled, it also runs in the back end, where it deals with multiple statement cases using flow-sensitive points-to information. Only warns when the converted pointer is dereferenced. Does not warn about incomplete types.

`-Wstrict-overflow`
`-Wstrict-overflow=`*n*

This option is only active when signed overflow is undefined. It warns about cases where the compiler optimizes based on the assumption that signed overflow does not occur. Note that it does not warn about all cases where the code might overflow: it only warns about cases where the compiler implements some optimization. Thus this warning depends on the optimization level.

An optimization that assumes that signed overflow does not occur is perfectly safe if the values of the variables involved are such that overflow never does, in fact, occur. Therefore this warning can easily give a false positive: a warning about code that is not actually a problem. To help focus on important issues, several warning levels are defined. No warnings are issued for the use of undefined signed overflow when estimating how many iterations a loop requires, in particular when determining whether a loop will be executed at all.

`-Wstrict-overflow=1`

Warn about cases that are both questionable and easy to avoid. For example the compiler simplifies `x + 1 > x` to `1`. This level of `-Wstrict-overflow` is enabled by `-Wall`; higher levels are not, and must be explicitly requested.

`-Wstrict-overflow=2`

Also warn about other cases where a comparison is simplified to a constant. For example: `abs (x) >= 0`. This can only be simplified when signed integer overflow is undefined, because `abs (INT_MIN)` overflows to `INT_MIN`, which is less than zero. `-Wstrict-overflow` (with no level) is the same as `-Wstrict-overflow=2`.

`-Wstrict-overflow=3`

Also warn about other cases where a comparison is simplified. For example: `x + 1 > 1` is simplified to `x > 0`.

`-Wstrict-overflow=4`

Also warn about other simplifications not covered by the above cases. For example: `(x * 10) / 5` is simplified to `x * 2`.

`-Wstrict-overflow=5`

Also warn about cases where the compiler reduces the magnitude of a constant involved in a comparison. For example: `x + 2 > y` is simplified to `x + 1 >= y`. This is reported only at the highest warning level because this simplification applies to many comparisons, so this warning level gives a very large number of false positives.

`-Wstring-compare`

Warn for calls to `strcmp` and `strncmp` whose result is determined to be either zero or non-zero in tests for such equality owing to the length of one argument being greater than the size of the array the other argument is stored in (or the bound in the case of `strncmp`). Such calls could be

mistakes. For example, the call to `strcmp` below is diagnosed because its result is necessarily non-zero irrespective of the contents of the array `a`.

```
extern char a[4];
void f (char *d)
{
  strcpy (d, "string");
  …
  if (0 == strcmp (a, d))   // cannot be true
    puts ("a and d are the same");
}
```

`-Wstring-compare` is enabled by `-Wextra`.

`-Wno-stringop-overflow`

`-Wstringop-overflow`

`-Wstringop-overflow=`*type*

Warn for calls to string manipulation functions such as `memcpy` and `strcpy` that are determined to overflow the destination buffer. The optional argument is one greater than the type of Object Size Checking to perform to determine the size of the destination. See Object Size Checking. The argument is meaningful only for functions that operate on character arrays but not for raw memory functions like `memcpy` which always make use of Object Size type-0. The option also warns for calls that specify a size in excess of the largest possible object or at most `SIZE_MAX / 2` bytes. The option produces the best results with optimization enabled but can detect a small subset of simple buffer overflows even without optimization in calls to the GCC built-in functions like `__builtin_memcpy` that correspond to the standard functions. In any case, the option warns about just a subset of buffer overflows detected by the corresponding overflow checking built-ins. For example, the option issues a warning for the `strcpy` call below because it copies at least 5 characters (the string `"blue"` including the terminating NUL) into the buffer of size 4.

```
enum Color { blue, purple, yellow };
const char* f (enum Color clr)
{
  static char buf [4];
  const char *str;
  switch (clr)
    {
      case blue: str = "blue"; break;
      case purple: str = "purple"; break;
      case yellow: str = "yellow"; break;
    }
```

```
  return strcpy (buf, str);   // warning here
}
```

Option `-Wstringop-overflow=2` is enabled by default.

`-Wstringop-overflow`
`-Wstringop-overflow=1`

The `-Wstringop-overflow=1` option uses type-zero Object Size Checking to determine the sizes of destination objects. At this setting the option does not warn for writes past the end of subobjects of larger objects accessed by pointers unless the size of the largest surrounding object is known. When the destination may be one of several objects it is assumed to be the largest one of them. On Linux systems, when optimization is enabled at this setting the option warns for the same code as when the `_FORTIFY_SOURCE` macro is defined to a non-zero value.

`-Wstringop-overflow=2`

The `-Wstringop-overflow=2` option uses type-one Object Size Checking to determine the sizes of destination objects. At this setting the option warns about overflows when writing to members of the largest complete objects whose exact size is known. However, it does not warn for excessive writes to the same members of unknown objects referenced by pointers since they may point to arrays containing unknown numbers of elements. This is the default setting of the option.

`-Wstringop-overflow=3`

The `-Wstringop-overflow=3` option uses type-two Object Size Checking to determine the sizes of destination objects. At this setting the option warns about overflowing the smallest object or data member. This is the most restrictive setting of the option that may result in warnings for safe code.

`-Wstringop-overflow=4`

The `-Wstringop-overflow=4` option uses type-three Object Size Checking to determine the sizes of destination objects. At this setting the option warns about overflowing any data members, and when the destination is one of several objects it uses the size of the largest of them to decide whether to issue a warning. Similarly to `-Wstringop-overflow=3` this setting of the option may result in warnings for benign code.

`-Wno-stringop-overread`

Warn for calls to string manipulation functions such as `memchr`, or `strcpy` that are determined to read past the end of the source sequence.

Option `-Wstringop-overread` is enabled by default.

`-Wno-stringop-truncation`

Do not warn for calls to bounded string manipulation functions such as `strncat`, `strncpy`, and `stpncpy` that may either truncate the copied string or leave the destination unchanged.

In the following example, the call to `strncat` specifies a bound that is less than the length of the source string. As a result, the copy of the source will be truncated and so the call is diagnosed. To avoid the warning use `bufsize - strlen (buf) - 1)` as the bound.

```
void append (char *buf, size_t bufsize)
{
  strncat (buf, ".txt", 3);
}
```

As another example, the following call to `strncpy` results in copying to `d` just the characters preceding the terminating NUL, without appending the NUL to the end. Assuming the result of `strncpy` is necessarily a NUL-terminated string is a common mistake, and so the call is diagnosed. To avoid the warning when the result is not expected to be NUL-terminated, call `memcpy` instead.

```
void copy (char *d, const char *s)
{
  strncpy (d, s, strlen (s));
}
```

In the following example, the call to `strncpy` specifies the size of the destination buffer as the bound. If the length of the source string is equal to or greater than this size the result of the copy will not be NUL-terminated. Therefore, the call is also diagnosed. To avoid the warning, specify `sizeof buf - 1` as the bound and set the last element of the buffer to `NUL`.

```
void copy (const char *s)
{
  char buf[80];
  strncpy (buf, s, sizeof buf);
  …
}
```

In situations where a character array is intended to store a sequence of bytes with no terminating `NUL` such an array may be annotated with attribute `nonstring` to avoid this warning. Such arrays, however, are not suitable arguments to functions that expect `NUL`-terminated strings. To

help detect accidental misuses of such arrays GCC issues warnings unless it can prove that the use is safe. See Common Variable Attributes.

`-Wstrict-flex-arrays` (C and C++ only)

Warn about improper usages of flexible array members according to the *level* of the `strict_flex_array (level)` attribute attached to the trailing array field of a structure if it's available, otherwise according to the *level* of the option `-fstrict-flex-arrays=level`. See Common Variable Attributes, for more information about the attribute, and Options Controlling C Dialect for more information about the option. `-Wstrict-flex-arrays` is effective only when *level* is greater than 0.

When *level*=1, warnings are issued for a trailing array reference of a structure that have 2 or more elements if the trailing array is referenced as a flexible array member.

When *level*=2, in addition to *level*=1, additional warnings are issued for a trailing one-element array reference of a structure if the array is referenced as a flexible array member.

When *level*=3, in addition to *level*=2, additional warnings are issued for a trailing zero-length array reference of a structure if the array is referenced as a flexible array member.

This option is more effective when `-ftree-vrp` is active (the default for `-O2` and above) but some warnings may be diagnosed even without optimization.

`-Wsuggest-attribute=[pure|const|noreturn|format|cold|malloc]returns_nonnull|`

Warn for cases where adding an attribute may be beneficial. The attributes currently supported are listed below.

```
-Wsuggest-attribute=pure
-Wsuggest-attribute=const
-Wsuggest-attribute=noreturn
-Wmissing-noreturn
-Wsuggest-attribute=malloc
-Wsuggest-attribute=returns_nonnull
-Wno-suggest-attribute=returns_nonnull
```

Warn about functions that might be candidates for attributes `pure`, `const`, `noreturn`, `malloc` or `returns_nonnull`. The compiler only warns for functions visible in other compilation units or (in the case of `pure` and `const`) if it cannot prove that the function returns normally. A function returns normally if it doesn't contain an infinite loop or return abnormally by throwing, calling `abort` or trapping. This analysis requires option `-fipa-pure-const`, which is enabled by default at `-O` and higher. Higher optimization levels improve the accuracy of the analysis.

`-Wsuggest-attribute=format`
`-Wmissing-format-attribute`

Warn about function pointers that might be candidates for `format` attributes. Note these are only possible candidates, not absolute ones. GCC guesses that function pointers with `format` attributes that are used in assignment, initialization, parameter passing or return statements should have a corresponding `format` attribute in the resulting type. I.e. the left-hand side of the assignment or initialization, the type of the parameter variable, or the return type of the containing function respectively should also have a `format` attribute to avoid the warning.

GCC also warns about function definitions that might be candidates for `format` attributes. Again, these are only possible candidates. GCC guesses that `format` attributes might be appropriate for any function that calls a function like `vprintf` or `vscanf`, but this might not always be the case, and some functions for which `format` attributes are appropriate may not be detected.

`-Wsuggest-attribute=cold`

Warn about functions that might be candidates for `cold` attribute. This is based on static detection and generally only warns about functions which always leads to a call to another `cold` function such as wrappers of C++ `throw` or fatal error reporting functions leading to `abort`.

`-Walloc-size`

Warn about calls to allocation functions decorated with attribute `alloc_size` that specify insufficient size for the target type of the pointer the result is assigned to, including those to the built-in forms of the functions `aligned_alloc`, `alloca`, `calloc`, `malloc`, and `realloc`.

`-Walloc-zero`

Warn about calls to allocation functions decorated with attribute `alloc_size` that specify zero bytes, including those to the built-in forms of the functions `aligned_alloc`, `alloca`, `calloc`, `malloc`, and `realloc`. Because the behavior of these functions when called with a zero size differs among implementations (and in the case of `realloc` has been deprecated) relying on it may result in subtle portability bugs and should be avoided.

`-Wcalloc-transposed-args`

Warn about calls to allocation functions decorated with attribute `alloc_size` with two arguments, which use `sizeof` operator as the earlier size argument and don't use it as the later size argument. This is a coding style warning. The first argument to `calloc` is documented to be number of elements in array, while the second argument is size of each element, so `calloc (n, sizeof (int))` is preferred over `calloc (sizeof (int), n)`. If `sizeof` in the earlier argument and not the latter is intentional, the warning can be suppressed by using `calloc (sizeof (struct S) + 0, n)` or `calloc (1 * sizeof (struct S), 4)` or using `sizeof` in the later argument as well.

`-Walloc-size-larger-than=`*byte-size*

Warn about calls to functions decorated with attribute `alloc_size` that attempt to allocate objects larger than the specified number of bytes, or where the result of the size computation in an integer type with infinite precision would exceed the value of '`PTRDIFF_MAX`' on the target. `-Walloc-size-larger-than='PTRDIFF_MAX'` is enabled by default. Warnings controlled by the option can be disabled either by specifying *byte-size* of '`SIZE_MAX`' or more or by `-Wno-alloc-size-larger-than`. See Declaring Attributes of Functions.

`-Wno-alloc-size-larger-than`

Disable `-Walloc-size-larger-than=` warnings. The option is equivalent to `-Walloc-size-larger-than='SIZE_MAX'` or larger.

`-Walloca`

This option warns on all uses of `alloca` in the source.

`-Walloca-larger-than=`*byte-size*

This option warns on calls to `alloca` with an integer argument whose value is either zero, or that is not bounded by a controlling predicate that limits its value to at most *byte-size*. It also warns for calls to `alloca` where the bound value is unknown. Arguments of non-integer types are considered unbounded even if they appear to be constrained to the expected range.

For example, a bounded case of `alloca` could be:

```
void func (size_t n)
{
  void *p;
  if (n <= 1000)
    p = alloca (n);
  else
    p = malloc (n);
  f (p);
}
```

In the above example, passing `-Walloca-larger-than=1000` would not issue a warning because the call to `alloca` is known to be at most 1000 bytes. However, if `-Walloca-larger-than=500` were passed, the compiler would emit a warning.

Unbounded uses, on the other hand, are uses of `alloca` with no controlling predicate constraining its integer argument. For example:

```
void func ()
{
  void *p = alloca (n);
  f (p);
}
```

If `-Walloca-larger-than=500` were passed, the above would trigger a warning, but this time because of the lack of bounds checking.

Note, that even seemingly correct code involving signed integers could cause a warning:

```
void func (signed int n)
{
  if (n < 500)
    {
      p = alloca (n);
      f (p);
    }
}
```

In the above example, *n* could be negative, causing a larger than expected argument to be implicitly cast into the `alloca` call.

This option also warns when `alloca` is used in a loop.

`-Walloca-larger-than=‘PTRDIFF_MAX’` is enabled by default but is usually only effective when `-ftree-vrp` is active (default for `-O2` and above).

See also `-Wvla-larger-than=‘byte-size’`.

`-Wno-alloca-larger-than`

Disable `-Walloca-larger-than=` warnings. The option is equivalent to `-Walloca-larger-than=‘SIZE_MAX’` or larger.

`-Warith-conversion`

Do warn about implicit conversions from arithmetic operations even when conversion of the operands to the same type cannot change their values. This affects warnings from `-Wconversion`, `-Wfloat-conversion`, and `-Wsign-conversion`.

```
void f (char c, int i)
{
```

```
  c = c + i; // warns with -Wconversion
  c = c + 1; // only warns with -Warith-conversion
}
```

`-Warray-bounds`

`-Warray-bounds=`*n*

Warn about out of bounds subscripts or offsets into arrays. This warning is enabled by `-Wall`. It is more effective when `-ftree-vrp` is active (the default for `-O2` and above) but a subset of instances are issued even without optimization.

By default, the trailing array of a structure will be treated as a flexible array member by `-Warray-bounds` or `-Warray-bounds=`*n* if it is declared as either a flexible array member per C99 standard onwards (‘`[]`’), a GCC zero-length array extension (‘`[0]`’), or an one-element array (‘`[1]`’). As a result, out of bounds subscripts or offsets into zero-length arrays or one-element arrays are not warned by default.

You can add the option `-fstrict-flex-arrays` or `-fstrict-flex-arrays=`*level* to control how this option treat trailing array of a structure as a flexible array member:

when *level*<=1, no change to the default behavior.

when *level*=2, additional warnings will be issued for out of bounds subscripts or offsets into one-element arrays;

when *level*=3, in addition to *level*=2, additional warnings will be issued for out of bounds subscripts or offsets into zero-length arrays.

`-Warray-bounds=1`

This is the default warning level of `-Warray-bounds` and is enabled by `-Wall`; higher levels are not, and must be explicitly requested.

`-Warray-bounds=2`

This warning level also warns about the intermediate results of pointer arithmetic that may yield out of bounds values. This warning level may give a larger number of false positives and is deactivated by default.

`-Warray-compare`

Warn about equality and relational comparisons between two operands of array type. This comparison was deprecated in C++20. For example:

```
int arr1[5];
int arr2[5];
bool same = arr1 == arr2;
```

`-Warray-compare` is enabled by `-Wall`.

`-Warray-parameter`

`-Warray-parameter=`*n*

Warn about redeclarations of functions involving parameters of array or pointer types of inconsistent kinds or forms, and enable the detection of out-of-bounds accesses to such parameters by warnings such as `-Warray-bounds`.

If the first function declaration uses the array form for a parameter declaration, the bound specified in the array is assumed to be the minimum number of elements expected to be provided in calls to the function and the maximum number of elements accessed by it. Failing to provide arguments of sufficient size or accessing more than the maximum number of elements may be diagnosed by warnings such as `-Warray-bounds` or `-Wstringop-overflow`. At level 1, the warning diagnoses inconsistencies involving array parameters declared using the `T[static N]` form.

For example, the warning triggers for the second declaration of `f` because the first one with the keyword `static` specifies that the array argument must have at least four elements, while the second allows an array of any size to be passed to `f`.

```
void f (int[static 4]);
void f (int[]);           // warning (inconsistent array form)

void g (void)
{
  int *p = (int *)malloc (1 * sizeof (int));
  f (p);                  // warning (array too small)
  …
}
```

At level 2 the warning also triggers for redeclarations involving any other inconsistency in array or pointer argument forms denoting array sizes. Pointers and arrays of unspecified bound are considered equivalent and do not trigger a warning.

```
void g (int*);
void g (int[]);     // no warning
void g (int[8]);    // warning (inconsistent array bound)
```

`-Warray-parameter=2` is included in `-Wall`. The `-Wvla-parameter` option triggers warnings for similar inconsistencies involving Variable Length Array arguments.

The short form of the option `-Warray-parameter` is equivalent to `-Warray-parameter=2`. The negative form `-Wno-array-parameter` is equivalent to `-Warray-parameter=0`.

`-Wattribute-alias=n`
`-Wno-attribute-alias`

Warn about declarations using the `alias` and similar attributes whose target is incompatible with the type of the alias. See Declaring Attributes of Functions.

`-Wattribute-alias=1`

The default warning level of the `-Wattribute-alias` option diagnoses incompatibilities between the type of the alias declaration and that of its target. Such incompatibilities are typically indicative of bugs.

`-Wattribute-alias=2`

At this level `-Wattribute-alias` also diagnoses cases where the attributes of the alias declaration are more restrictive than the attributes applied to its target. These mismatches can potentially result in incorrect code generation. In other cases they may be benign and could be resolved simply by adding the missing attribute to the target. For comparison, see the `-Wmissing-attributes` option, which controls diagnostics when the alias declaration is less restrictive than the target, rather than more restrictive.

Attributes considered include `alloc_align`, `alloc_size`, `cold`, `const`, `hot`, `leaf`, `malloc`, `nonnull`, `noreturn`, `nothrow`, `pure`, `returns_nonnull`, and `returns_twice`.

`-Wattribute-alias` is equivalent to `-Wattribute-alias=1`. This is the default. You can disable these warnings with either `-Wno-attribute-alias` or `-Wattribute-alias=0`.

`-Wbidi-chars=[none|unpaired|any|ucn]`

Warn about possibly misleading UTF-8 bidirectional control characters in comments, string literals, character constants, and identifiers. Such characters can change left-to-right writing direction into right-to-left (and vice versa), which can cause confusion between the logical order and visual order. This may be dangerous; for instance, it may seem that a piece of code is not commented out, whereas it in fact is.

There are three levels of warning supported by GCC. The default is `-Wbidi-chars=unpaired`, which warns about improperly terminated bidi contexts. `-Wbidi-chars=none` turns the warning off. `-Wbidi-chars=any` warns about any use of bidirectional control characters.

By default, this warning does not warn about UCNs. It is, however, possible to turn on such checking by using `-Wbidi-chars=unpaired,ucn` or `-Wbidi-chars=any,ucn`. Using `-Wbidi-chars=ucn` is valid, and is equivalent to `-Wbidi-chars=unpaired,ucn`, if no previous `-Wbidi-chars=any` was specified.

`-Wbool-compare`

Warn about boolean expression compared with an integer value different from `true`/`false`. For instance, the following comparison is always false:

```
int n = 5;
…
if ((n > 1) == 2) { … }
```

This warning is enabled by `-Wall`.

`-Wbool-operation`

Warn about suspicious operations on expressions of a boolean type. For instance, bitwise negation of a boolean is very likely a bug in the program. For C, this warning also warns about incrementing or decrementing a boolean, which rarely makes sense. (In C++, decrementing a boolean is always invalid. Incrementing a boolean is invalid in C++17, and deprecated otherwise.)

This warning is enabled by `-Wall`.

`-Wduplicated-branches`

Warn when an if-else has identical branches. This warning detects cases like

```
if (p != NULL)
  return 0;
else
  return 0;
```

It doesn't warn when both branches contain just a null statement. This warning also warn for conditional operators:

```
  int i = x ? *p : *p;
```

`-Wduplicated-cond`

Warn about duplicated conditions in an if-else-if chain. For instance, warn for the following code:

```
if (p->q != NULL) { … }
```

```
else if (p->q != NULL) { … }
```

`-Wframe-address`

Warn when the '`__builtin_frame_address`' or '`__builtin_return_address`' is called with an argument greater than 0. Such calls may return indeterminate values or crash the program. The warning is included in `-Wall`.

`-Wno-discarded-qualifiers` (C and Objective-C only)

Do not warn if type qualifiers on pointers are being discarded. Typically, the compiler warns if a `const char *` variable is passed to a function that takes a `char *` parameter. This option can be used to suppress such a warning.

`-Wno-discarded-array-qualifiers` (C and Objective-C only)

Do not warn if type qualifiers on arrays which are pointer targets are being discarded. Typically, the compiler warns if a `const int (*)[]` variable is passed to a function that takes a `int (*)[]` parameter. This option can be used to suppress such a warning.

`-Wno-incompatible-pointer-types` (C and Objective-C only)

Do not warn when there is a conversion between pointers that have incompatible types. This warning is for cases not covered by `-Wno-pointer-sign`, which warns for pointer argument passing or assignment with different signedness.

By default, in C99 and later dialects of C, GCC treats this issue as an error. The error can be downgraded to a warning using `-fpermissive` (along with certain other errors), or for this error alone, with `-Wno-error=incompatible-pointer-types`.

This warning is upgraded to an error by `-pedantic-errors`.

`-Wno-int-conversion` (C and Objective-C only)

Do not warn about incompatible integer to pointer and pointer to integer conversions. This warning is about implicit conversions; for explicit conversions the warnings `-Wno-int-to-pointer-cast` and `-Wno-pointer-to-int-cast` may be used.

By default, in C99 and later dialects of C, GCC treats this issue as an error. The error can be downgraded to a warning using `-fpermissive` (along with certain other errors), or for this error alone, with `-Wno-error=int-conversion`.

This warning is upgraded to an error by `-pedantic-errors`.

`-Wzero-length-bounds`

Warn about accesses to elements of zero-length array members that might overlap other members of the same object. Declaring interior zero-length arrays is discouraged because accesses to them are undefined. See Arrays of Length Zero.

For example, the first two stores in function `bad` are diagnosed because the array elements overlap the subsequent members `b` and `c`. The third store is diagnosed by `-Warray-bounds` because it is beyond the bounds of the enclosing object.

```
struct X { int a[0]; int b, c; };
struct X x;

void bad (void)
{
  x.a[0] = 0;   // -Wzero-length-bounds
  x.a[1] = 1;   // -Wzero-length-bounds
  x.a[2] = 2;   // -Warray-bounds
}
```

Option `-Wzero-length-bounds` is enabled by `-Warray-bounds`.

`-Wno-div-by-zero`

Do not warn about compile-time integer division by zero. Floating-point division by zero is not warned about, as it can be a legitimate way of obtaining infinities and NaNs.

`-Wsystem-headers`

Print warning messages for constructs found in system header files. Warnings from system headers are normally suppressed, on the assumption that they usually do not indicate real problems and would only make the compiler output harder to read. Using this command-line option tells GCC to emit warnings from system headers as if they occurred in user code. However, note that using `-Wall` in conjunction with this option does *not* warn about unknown pragmas in system headers—for that, `-Wunknown-pragmas` must also be used.

`-Wtautological-compare`

Warn if a self-comparison always evaluates to true or false. This warning detects various mistakes such as:

```
int i = 1;
…
if (i > i) { … }
```

This warning also warns about bitwise comparisons that always evaluate to true or false, for instance:

```
if ((a & 16) == 10) { … }
```

will always be false.

This warning is enabled by `-Wall`.

`-Wtrampolines`

Warn about trampolines generated for pointers to nested functions. A trampoline is a small piece of data or code that is created at run time on the stack when the address of a nested function is taken, and is used to call the nested function indirectly. For some targets, it is made up of data only and thus requires no special treatment. But, for most targets, it is made up of code and thus requires the stack to be made executable in order for the program to work properly.

`-Wfloat-equal`

Warn if floating-point values are used in equality comparisons.

The idea behind this is that sometimes it is convenient (for the programmer) to consider floating-point values as approximations to infinitely precise real numbers. If you are doing this, then you need to compute (by analyzing the code, or in some other way) the maximum or likely maximum error that the computation introduces, and allow for it when performing comparisons (and when producing output, but that's a different problem). In particular, instead of testing for equality, you should check to see whether the two values have ranges that overlap; and this is done with the relational operators, so equality comparisons are probably mistaken.

`-Wtraditional` (C and Objective-C only)

Warn about certain constructs that behave differently in traditional and ISO C. Also warn about ISO C constructs that have no traditional C equivalent, and/or problematic constructs that should be avoided.

- Macro parameters that appear within string literals in the macro body. In traditional C macro replacement takes place within string literals, but in ISO C it does not.
- In traditional C, some preprocessor directives did not exist. Traditional preprocessors only considered a line to be a directive if the '`#`' appeared in column 1 on the line. Therefore `-Wtraditional` warns about directives that traditional C understands but ignores because the '`#`' does not appear as the first character on the line. It also suggests you hide directives like `#pragma` not understood by traditional C by indenting them. Some traditional implementations do not recognize `#elif`, so this option suggests avoiding it altogether.
- A function-like macro that appears without arguments.

- The unary plus operator.
- The ‘U’ integer constant suffix, or the ‘F’ or ‘L’ floating-point constant suffixes. (Traditional C does support the ‘L’ suffix on integer constants.) Note, these suffixes appear in macros defined in the system headers of most modern systems, e.g. the ‘_MIN’/‘_MAX’ macros in <limits.h>. Use of these macros in user code might normally lead to spurious warnings, however GCC’s integrated preprocessor has enough context to avoid warning in these cases.
- A function declared external in one block and then used after the end of the block.
- A switch statement has an operand of type long.
- A non-static function declaration follows a static one. This construct is not accepted by some traditional C compilers.
- The ISO type of an integer constant has a different width or signedness from its traditional type. This warning is only issued if the base of the constant is ten. I.e. hexadecimal or octal values, which typically represent bit patterns, are not warned about.
- Usage of ISO string concatenation is detected.
- Initialization of automatic aggregates.
- Identifier conflicts with labels. Traditional C lacks a separate namespace for labels.
- Initialization of unions. If the initializer is zero, the warning is omitted. This is done under the assumption that the zero initializer in user code appears conditioned on e.g. __STDC__ to avoid missing initializer warnings and relies on default initialization to zero in the traditional C case.
- Conversions by prototypes between fixed/floating-point values and vice versa. The absence of these prototypes when compiling with traditional C causes serious problems. This is a subset of the possible conversion warnings; for the full set use -Wtraditional-conversion.
- Use of ISO C style function definitions. This warning intentionally is *not* issued for prototype declarations or variadic functions because these ISO C features appear in your code when using libiberty’s traditional C compatibility macros, PARAMS and VPARAMS. This warning is also bypassed for nested functions because that feature is already a GCC extension and thus not relevant to traditional C compatibility.

`-Wtraditional-conversion` (C and Objective-C only)

Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed-point argument except when the same as the default promotion.

`-Wdeclaration-after-statement` (C and Objective-C only)

Warn when a declaration is found after a statement in a block. This construct, known from C++, was introduced with ISO C99 and is by default allowed in GCC. It is not supported by ISO C90. See Mixed Declarations, Labels and Code.

This warning is upgraded to an error by `-pedantic-errors`.

`-Wshadow`

Warn whenever a local variable or type declaration shadows another variable, parameter, type, class member (in C++), or instance variable (in Objective-C) or whenever a built-in function is shadowed. Note that in C++, the compiler warns if a local variable shadows an explicit typedef, but not if it shadows a struct/class/enum. If this warning is enabled, it includes also all instances of local shadowing. This means that `-Wno-shadow=local` and `-Wno-shadow=compatible-local` are ignored when `-Wshadow` is used. Same as `-Wshadow=global`.

`-Wno-shadow-ivar` (Objective-C only)

Do not warn whenever a local variable shadows an instance variable in an Objective-C method.

`-Wshadow=global`

Warn for any shadowing. Same as `-Wshadow`.

`-Wshadow=local`

Warn when a local variable shadows another local variable or parameter.

`-Wshadow=compatible-local`

Warn when a local variable shadows another local variable or parameter whose type is compatible with that of the shadowing variable. In C++, type compatibility here means the type of the shadowing variable can be converted to that of the shadowed variable. The creation of this flag (in addition to `-Wshadow=local`) is based on the idea that when a local variable shadows another one of incompatible type, it is most likely intentional, not a bug or typo, as shown in the following example:

```
for (SomeIterator i = SomeObj.begin(); i != SomeObj.end(); ++i)
{
  for (int i = 0; i < N; ++i)
  {
    ...
  }
  ...
}
```

Since the two variable `i` in the example above have incompatible types, enabling only `-Wshadow=compatible-local` does not emit a warning. Because their types are incompatible, if a programmer accidentally uses one in place of the other, type checking is expected to catch that and emit an error or warning. Use of this flag instead of `-Wshadow=local` can possibly reduce the number of warnings triggered by intentional shadowing. Note that this also means that shadowing `const char *i` by `char *i` does not emit a warning.

This warning is also enabled by `-Wshadow=local`.

`-Wlarger-than=`*byte-size*

Warn whenever an object is defined whose size exceeds *byte-size*. `-Wlarger-than=‘PTRDIFF_MAX’` is enabled by default. Warnings controlled by the option can be disabled either by specifying *byte-size* of ‘`SIZE_MAX`’ or more or by `-Wno-larger-than`.

Also warn for calls to bounded functions such as `memchr` or `strnlen` that specify a bound greater than the largest possible object, which is ‘`PTRDIFF_MAX`’ bytes by default. These warnings can only be disabled by `-Wno-larger-than`.

`-Wno-larger-than`

Disable `-Wlarger-than=` warnings. The option is equivalent to `-Wlarger-than=‘SIZE_MAX’` or larger.

`-Wframe-larger-than=`*byte-size*

Warn if the size of a function frame exceeds *byte-size*. The computation done to determine the stack frame size is approximate and not conservative. The actual requirements may be somewhat greater than *byte-size* even if you do not get a warning. In addition, any space allocated via `alloca`, variable-length arrays, or related constructs is not included by the compiler when determining whether or not to issue a warning. `-Wframe-larger-than=‘PTRDIFF_MAX’` is enabled by default. Warnings controlled by the option can be disabled either by specifying *byte-size* of ‘`SIZE_MAX`’ or more or by `-Wno-frame-larger-than`.

`-Wno-frame-larger-than`

Disable `-Wframe-larger-than=` warnings. The option is equivalent to `-Wframe-larger-than=‘SIZE_MAX’` or larger.

`-Wfree-nonheap-object`

Warn when attempting to deallocate an object that was either not allocated on the heap, or by using a pointer that was not returned from a prior call to the corresponding allocation function. For example, because the call to `stpcpy` returns a pointer to the terminating nul character and not to the beginning of the object, the call to `free` below is diagnosed.

```
void f (char *p)
{
  p = stpcpy (p, "abc");
  // ...
  free (p);   // warning
}
```

`-Wfree-nonheap-object` is included in `-Wall`.

`-Wstack-usage=`*`byte-size`*

Warn if the stack usage of a function might exceed *byte-size*. The computation done to determine the stack usage is conservative. Any space allocated via `alloca`, variable-length arrays, or related constructs is included by the compiler when determining whether or not to issue a warning.

The message is in keeping with the output of `-fstack-usage`.

- If the stack usage is fully static but exceeds the specified amount, it's:

```
warning: stack usage is 1120 bytes
```

- If the stack usage is (partly) dynamic but bounded, it's:

```
warning: stack usage might be 1648 bytes
```

- If the stack usage is (partly) dynamic and not bounded, it's:

```
warning: stack usage might be unbounded
```

`-Wstack-usage=`'`PTRDIFF_MAX`' is enabled by default. Warnings controlled by the option can be disabled either by specifying *byte-size* of '`SIZE_MAX`' or more or by `-Wno-stack-usage`.

`-Wno-stack-usage`

Disable `-Wstack-usage=` warnings. The option is equivalent to `-Wstack-usage=`'`SIZE_MAX`' or larger.

`-Wunsafe-loop-optimizations`

Warn if the loop cannot be optimized because the compiler cannot assume anything on the bounds of the loop indices. With `-funsafe-loop-optimizations` warn if the compiler makes such assumptions.

`-Wno-pedantic-ms-format` (MinGW targets only)

When used in combination with `-Wformat` and `-pedantic` without GNU extensions, this option disables the warnings about non-ISO `printf` / `scanf` format width specifiers `I32`, `I64`, and `I` used on Windows targets, which depend on the MS runtime.

`-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. In C++, warn also when an arithmetic operation involves `NULL`. This warning is also enabled by `-Wpedantic`.

This warning is upgraded to an error by `-pedantic-errors`.

`-Wno-pointer-compare`

Do not warn if a pointer is compared with a zero character constant. This usually means that the pointer was meant to be dereferenced. For example:

```
const char *p = foo ();
if (p == '\0')
  return 42;
```

Note that the code above is invalid in C++11.

This warning is enabled by default.

`-Wno-tsan`

Disable warnings about unsupported features in ThreadSanitizer.

ThreadSanitizer does not support `std::atomic_thread_fence` and can report false positives.

`-Wtype-limits`

Warn if a comparison is always true or always false due to the limited range of the data type, but do not warn for constant expressions. For example, warn if an unsigned variable is compared against zero with `<` or `>=`. This warning is also enabled by `-Wextra`.

`-Wabsolute-value` (C and Objective-C only)

Warn for calls to standard functions that compute the absolute value of an argument when a more appropriate standard function is available. For example, calling `abs(3.14)` triggers the warning because the appropriate function to call to compute the absolute value of a double argument is `fabs`. The option also triggers warnings when the argument in a call to such a function has an unsigned type. This warning can be suppressed with an explicit type cast and it is also enabled by `-Wextra`.

`-Wcomment`

`-Wcomments`

Warn whenever a comment-start sequence ‘`/*`’ appears in a ‘`/*`’ comment, or whenever a backslash-newline appears in a ‘`//`’ comment. This warning is enabled by `-Wall`.

`-Wtrigraphs`

Warn if any trigraphs are encountered that might change the meaning of the program. Trigraphs within comments are not warned about, except those that would form escaped newlines.

This option is implied by `-Wall`. If `-Wall` is not given, this option is still enabled unless trigraphs are enabled. To get trigraph conversion without warnings, but get the other `-Wall` warnings, use ‘`-trigraphs -Wall -Wno-trigraphs`’.

`-Wundef`

Warn if an undefined identifier is evaluated in an `#if` directive. Such identifiers are replaced with zero.

`-Wexpansion-to-defined`

Warn whenever ‘`defined`’ is encountered in the expansion of a macro (including the case where the macro is expanded by an ‘`#if`’ directive). Such usage is not portable. This warning is also enabled by `-Wpedantic` and `-Wextra`.

`-Wunused-macros`

Warn about macros defined in the main file that are unused. A macro is *used* if it is expanded or tested for existence at least once. The preprocessor also warns if the macro has not been used at the time it is redefined or undefined.

Built-in macros, macros defined on the command line, and macros defined in include files are not warned about.

Note: If a macro is actually used, but only used in skipped conditional blocks, then the preprocessor reports it as unused. To avoid the warning in such a case, you might improve the scope of the macro’s definition by, for example, moving it into the first skipped block. Alternatively, you could provide a dummy use with something like:

```
#if defined the_macro_causing_the_warning
#endif
```

`-Wno-endif-labels`

Do not warn whenever an `#else` or an `#endif` are followed by text. This sometimes happens in older programs with code of the form

```
#if FOO
…
#else FOO
…
#endif FOO
```

The second and third `FOO` should be in comments. This warning is on by default.

`-Wbad-function-cast` (C and Objective-C only)

Warn when a function call is cast to a non-matching type. For example, warn if a call to a function returning an integer type is cast to a pointer type.

`-Wc90-c99-compat` (C and Objective-C only)

Warn about features not present in ISO C90, but present in ISO C99. For instance, warn about use of variable length arrays, `long long` type, `bool` type, compound literals, designated initializers, and so on. This option is independent of the standards mode. Warnings are disabled in the expression that follows `__extension__`.

`-Wc99-c11-compat` (C and Objective-C only)

Warn about features not present in ISO C99, but present in ISO C11. For instance, warn about use of anonymous structures and unions, `_Atomic` type qualifier, `_Thread_local` storage-class specifier, `_Alignas` specifier, `Alignof` operator, `_Generic` keyword, and so on. This option is independent of the standards mode. Warnings are disabled in the expression that follows `__extension__`.

`-Wc11-c23-compat` (C and Objective-C only)
`-Wc11-c2x-compat` (C and Objective-C only)

Warn about features not present in ISO C11, but present in ISO C23. For instance, warn about omitting the string in `_Static_assert`, use of ‘`[[]]`’ syntax for attributes, use of decimal floating-point types, and so on. This option is independent of the standards mode. Warnings are disabled in the expression that follows `__extension__`. The name `-Wc11-c2x-compat` is deprecated.

When not compiling in C23 mode, these warnings are upgraded to errors by `-pedantic-errors`.

`-Wc++-compat` (C and Objective-C only)

Warn about ISO C constructs that are outside of the common subset of ISO C and ISO C++, e.g. request for implicit conversion from `void *` to a pointer to non-`void` type.

`-Wc++11-compat` (C++ and Objective-C++ only)

Warn about C++ constructs whose meaning differs between ISO C++ 1998 and ISO C++ 2011, e.g., identifiers in ISO C++ 1998 that are keywords in ISO C++ 2011. This warning turns on `-Wnarrowing` and is enabled by `-Wall`.

`-Wc++14-compat` (C++ and Objective-C++ only)

Warn about C++ constructs whose meaning differs between ISO C++ 2011 and ISO C++ 2014. This warning is enabled by `-Wall`.

`-Wc++17-compat` (C++ and Objective-C++ only)

Warn about C++ constructs whose meaning differs between ISO C++ 2014 and ISO C++ 2017. This warning is enabled by `-Wall`.

`-Wc++20-compat` (C++ and Objective-C++ only)

Warn about C++ constructs whose meaning differs between ISO C++ 2017 and ISO C++ 2020. This warning is enabled by `-Wall`.

`-Wno-c++11-extensions` (C++ and Objective-C++ only)

Do not warn about C++11 constructs in code being compiled using an older C++ standard. Even without this option, some C++11 constructs will only be diagnosed if `-Wpedantic` is used.

`-Wno-c++14-extensions` (C++ and Objective-C++ only)

Do not warn about C++14 constructs in code being compiled using an older C++ standard. Even without this option, some C++14 constructs will only be diagnosed if `-Wpedantic` is used.

`-Wno-c++17-extensions` (C++ and Objective-C++ only)

Do not warn about C++17 constructs in code being compiled using an older C++ standard. Even without this option, some C++17 constructs will only be diagnosed if `-Wpedantic` is used.

`-Wno-c++20-extensions` (C++ and Objective-C++ only)

Do not warn about C++20 constructs in code being compiled using an older C++ standard. Even without this option, some C++20 constructs will only be diagnosed if `-Wpedantic` is used.

`-Wno-c++23-extensions` (C++ and Objective-C++ only)

Do not warn about C++23 constructs in code being compiled using an older C++ standard. Even without this option, some C++23 constructs will only be diagnosed if `-Wpedantic` is used.

`-Wno-c++26-extensions` (C++ and Objective-C++ only)

Do not warn about C++26 constructs in code being compiled using an older C++ standard. Even without this option, some C++26 constructs will only be diagnosed if `-Wpedantic` is used.

`-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 *`.

Also warn when making a cast that introduces a type qualifier in an unsafe way. For example, casting `char **` to `const char **` is unsafe, as in this example:

```
/* p is char ** value.  */
const char **q = (const char **) p;
/* Assignment of readonly string to const char * is OK.  */
*q = "string";
/* Now char** pointer points to read-only memory.  */
**p = 'b';
```

`-Wcast-align`

Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries.

`-Wcast-align=strict`

Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` regardless of the target machine.

`-Wcast-function-type`

Warn when a function pointer is cast to an incompatible function pointer. In a cast involving function types with a variable argument list only the types of initial arguments that are provided are considered. Any parameter of pointer-type matches any other pointer-type. Any benign

differences in integral types are ignored, like `int` vs. `long` on ILP32 targets. Likewise type qualifiers are ignored. The function type `void (*) (void)` is special and matches everything, which can be used to suppress this warning. In a cast involving pointer to member types this warning warns whenever the type cast is changing the pointer to member type. This warning is enabled by `-Wextra`.

`-Wcast-user-defined`

Warn when a cast to reference type does not involve a user-defined conversion that the programmer might expect to be called.

```
struct A { operator const int&(); } a;
auto r = (int&)a; // warning
```

This warning is enabled by default.

`-Wwrite-strings`

When compiling C, give string constants the type `const char[`*`length`*`]` so that copying the address of one into a non-`const char *` pointer produces a warning. These warnings 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 is just a nuisance. This is why we did not make `-Wall` request these warnings.

When compiling C++, warn about the deprecated conversion from string literals to `char *`. This warning is enabled by default for C++ programs.

This warning is upgraded to an error by `-pedantic-errors` in C++11 mode or later.

`-Wclobbered`

Warn for variables that might be changed by `longjmp` or `vfork`. This warning is also enabled by `-Wextra`.

`-Wno-complain-wrong-lang`

By default, language front ends complain when a command-line option is valid, but not applicable to that front end. This may be disabled with `-Wno-complain-wrong-lang`, which is mostly useful when invoking a single compiler driver for multiple source files written in different languages, for example:

```
$ g++ -fno-rtti a.cc b.f90
```

The driver `g++` invokes the C++ front end to compile `a.cc` and the Fortran front end to compile `b.f90`. The latter front end diagnoses '`f951: Warning: command-line option '-fno-rtti' is valid for C++/D/ObjC++ but not for Fortran`', which may be disabled with `-Wno-complain-wrong-lang`.

`-Wcompare-distinct-pointer-types` (C and Objective-C only)

Warn if pointers of distinct types are compared without a cast. This warning is enabled by default.

`-Wconversion`

Warn for implicit conversions that may alter a value. This includes conversions between real and integer, like `abs (x)` when `x` is `double`; conversions between signed and unsigned, like `unsigned ui = -1`; and conversions to smaller types, like `sqrtf (M_PI)`. Do not warn for explicit casts like `abs ((int) x)` and `ui = (unsigned) -1`, or if the value is not changed by the conversion like in `abs (2.0)`. Warnings about conversions between signed and unsigned integers can be disabled by using `-Wno-sign-conversion`.

For C++, also warn for confusing overload resolution for user-defined conversions; and conversions that never use a type conversion operator: conversions to `void`, the same type, a base class or a reference to them. Warnings about conversions between signed and unsigned integers are disabled by default in C++ unless `-Wsign-conversion` is explicitly enabled.

Warnings about conversion from arithmetic on a small type back to that type are only given with `-Warith-conversion`.

`-Wdangling-else`

Warn about constructions where there may be confusion to which `if` statement an `else` branch belongs. Here is an example of such a case:

```
{
  if (a)
    if (b)
      foo ();
  else
    bar ();
}
```

In C/C++, every `else` branch belongs to the innermost possible `if` statement, which in this example is `if (b)`. This is often not what the programmer expected, as illustrated in the above example by indentation the programmer chose. When there is the potential for this confusion, GCC issues a warning when this flag is specified. To eliminate the warning, add explicit braces around the innermost `if` statement so there is no way the `else` can belong to the enclosing `if`. The resulting code looks like this:

```
{
  if (a)
    {
      if (b)
        foo ();
      else
        bar ();
    }
}
```

This warning is enabled by `-Wparentheses`.

`-Wdangling-pointer`

`-Wdangling-pointer=`*n*

Warn about uses of pointers (or C++ references) to objects with automatic storage duration after their lifetime has ended. This includes local variables declared in nested blocks, compound literals and other unnamed temporary objects. In addition, warn about storing the address of such objects in escaped pointers. The warning is enabled at all optimization levels but may yield different results with optimization than without.

`-Wdangling-pointer=1`

At level 1, the warning diagnoses only unconditional uses of dangling pointers.

`-Wdangling-pointer=2`

At level 2, in addition to unconditional uses the warning also diagnoses conditional uses of dangling pointers.

The short form `-Wdangling-pointer` is equivalent to `-Wdangling-pointer=2`, while `-Wno-dangling-pointer` and `-Wdangling-pointer=0` have the same effect of disabling the warnings. `-Wdangling-pointer=2` is included in `-Wall`.

This example triggers the warning at level 1; the address of the unnamed temporary is unconditionally referenced outside of its scope.

```
char f (char c1, char c2, char c3)
{
  char *p;
  {
    p = (char[]) { c1, c2, c3 };
  }
```

```
  // warning: using dangling pointer 'p' to an unnamed temporary
  return *p;
}
```

In the following function the store of the address of the local variable `x` in the escaped pointer `*p` triggers the warning at level 1.

```
void g (int **p)
{
  int x = 7;
  // warning: storing the address of local variable 'x' in '*p'
  *p = &x;
}
```

In this example, the array *a* is out of scope when the pointer *s* is used. Since the code that sets `s` is conditional, the warning triggers at level 2.

```
extern void frob (const char *);
void h (char *s)
{
  if (!s)
    {
      char a[12] = "tmpname";
      s = a;
    }
  // warning: dangling pointer 's' to 'a' may be used
  frob (s);
}
```

`-Wdate-time`

Warn when macros `__TIME__`, `__DATE__` or `__TIMESTAMP__` are encountered as they might prevent bit-wise-identical reproducible compilations.

`-Wempty-body`

Warn if an empty body occurs in an `if`, `else` or `do while` statement. This warning is also enabled by `-Wextra`.

`-Wno-endif-labels`

Do not warn about stray tokens after `#else` and `#endif`.

`-Wenum-compare`

Warn about a comparison between values of different enumerated types. In C++ enumerated type mismatches in conditional expressions are also diagnosed and the warning is enabled by default. In C this warning is enabled by `-Wall`.

`-Wenum-conversion`

Warn when a value of enumerated type is implicitly converted to a different enumerated type. This warning is enabled by `-Wextra` in C.

`-Wenum-int-mismatch` (C and Objective-C only)

Warn about mismatches between an enumerated type and an integer type in declarations. For example:

```
enum E { l = -1, z = 0, g = 1 };
int foo(void);
enum E foo(void);
```

In C, an enumerated type is compatible with `char`, a signed integer type, or an unsigned integer type. However, since the choice of the underlying type of an enumerated type is implementation-defined, such mismatches may cause portability issues. In C++, such mismatches are an error. In C, this warning is enabled by `-Wall` and `-Wc++-compat`.

`-Wjump-misses-init` (C, Objective-C only)

Warn if a `goto` statement or a `switch` statement jumps forward across the initialization of a variable, or jumps backward to a label after the variable has been initialized. This only warns about variables that are initialized when they are declared. This warning is only supported for C and Objective-C; in C++ this sort of branch is an error in any case.

`-Wjump-misses-init` is included in `-Wc++-compat`. It can be disabled with the `-Wno-jump-misses-init` option.

`-Wsign-compare`

Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. In C++, this warning is also enabled by `-Wall`. In C, it is also enabled by `-Wextra`.

`-Wsign-conversion`

Warn for implicit conversions that may change the sign of an integer value, like assigning a signed integer expression to an unsigned integer variable. An explicit cast silences the warning. In C, this option is enabled also by `-Wconversion`.

`-Wflex-array-member-not-at-end` (C and C++ only)

Warn when a structure containing a C99 flexible array member as the last field is not at the end of another structure. This warning warns e.g. about

```
struct flex  { int length; char data[]; };
struct mid_flex { int m; struct flex flex_data; int n; };
```

`-Wfloat-conversion`

Warn for implicit conversions that reduce the precision of a real value. This includes conversions from real to integer, and from higher precision real to lower precision real values. This option is also enabled by `-Wconversion`.

`-Wno-scalar-storage-order`

Do not warn on suspicious constructs involving reverse scalar storage order.

`-Wsizeof-array-div`

Warn about divisions of two sizeof operators when the first one is applied to an array and the divisor does not equal the size of the array element. In such a case, the computation will not yield the number of elements in the array, which is likely what the user intended. This warning warns e.g. about

```
int fn ()
{
  int arr[10];
  return sizeof (arr) / sizeof (short);
}
```

This warning is enabled by `-Wall`.

`-Wsizeof-pointer-div`

Warn for suspicious divisions of two sizeof expressions that divide the pointer size by the element size, which is the usual way to compute the array size but won't work out correctly with pointers. This warning warns e.g. about `sizeof (ptr) / sizeof (ptr[0])` if `ptr` is not an array, but a pointer. This warning is enabled by `-Wall`.

`-Wsizeof-pointer-memaccess`

Warn for suspicious length parameters to certain string and memory built-in functions if the argument uses `sizeof`. This warning triggers for example for `memset (ptr, 0, sizeof (ptr));` if `ptr` is not an array, but a pointer, and suggests a possible fix, or about `memcpy (&foo, ptr, sizeof (&foo));`. `-Wsizeof-pointer-memaccess` also warns about calls to bounded string copy functions like `strncat` or `strncpy` that specify as the bound a `sizeof` expression of the source array. For example, in the following function the call to `strncat` specifies the size of the source string as the bound. That is almost certainly a mistake and so the call is diagnosed.

```
void make_file (const char *name)
{
  char path[PATH_MAX];
  strncpy (path, name, sizeof path - 1);
  strncat (path, ".text", sizeof ".text");
  …
}
```

The `-Wsizeof-pointer-memaccess` option is enabled by `-Wall`.

`-Wno-sizeof-array-argument`

Do not warn when the `sizeof` operator is applied to a parameter that is declared as an array in a function definition. This warning is enabled by default for C and C++ programs.

`-Wmemset-elt-size`

Warn for suspicious calls to the `memset` built-in function, if the first argument references an array, and the third argument is a number equal to the number of elements, but not equal to the size of the array in memory. This indicates that the user has omitted a multiplication by the element size. This warning is enabled by `-Wall`.

`-Wmemset-transposed-args`

Warn for suspicious calls to the `memset` built-in function where the second argument is not zero and the third argument is zero. For example, the call `memset (buf, sizeof buf, 0)` is diagnosed because `memset (buf, 0, sizeof buf)` was meant instead. The diagnostic is only emitted if the third argument is a literal zero. Otherwise, if it is an expression that is folded to zero, or a cast of zero to some type, it is far less likely that the arguments have been mistakenly transposed and no warning is emitted. This warning is enabled by `-Wall`.

`-Waddress`

Warn about suspicious uses of address expressions. These include comparing the address of a function or a declared object to the null pointer constant such as in

```
void f (void);
void g (void)
{
  if (!f)   // warning: expression evaluates to false
    abort ();
}
```

comparisons of a pointer to a string literal, such as in

```
void f (const char *x)
{
  if (x == "abc")   // warning: expression evaluates to false
    puts ("equal");
}
```

and tests of the results of pointer addition or subtraction for equality to null, such as in

```
void f (const int *p, int i)
{
  return p + i == NULL;
}
```

Such uses typically indicate a programmer error: the address of most functions and objects necessarily evaluates to true (the exception are weak symbols), so their use in a conditional might indicate missing parentheses in a function call or a missing dereference in an array expression. The subset of the warning for object pointers can be suppressed by casting the pointer operand to an integer type such as `intptr_t` or `uintptr_t`. Comparisons against string literals result in unspecified behavior and are not portable, and suggest the intent was to call `strcmp`. The warning is suppressed if the suspicious expression is the result of macro expansion. `-Waddress` warning is enabled by `-Wall`.

`-Wno-address-of-packed-member`

Do not warn when the address of packed member of struct or union is taken, which usually results in an unaligned pointer value. This is enabled by default.

`-Wlogical-op`

Warn about suspicious uses of logical operators in expressions. This includes using logical operators in contexts where a bit-wise operator is likely to be expected. Also warns when the operands of a logical operator are the same:

```
extern int a;
if (a < 0 && a < 0) { … }
```

`-Wlogical-not-parentheses`

Warn about logical not used on the left hand side operand of a comparison. This option does not warn if the right operand is considered to be a boolean expression. Its purpose is to detect suspicious code like the following:

```
int a;
…
if (!a > 1) { … }
```

It is possible to suppress the warning by wrapping the LHS into parentheses:

```
if ((!a) > 1) { … }
```

This warning is enabled by `-Wall`.

`-Waggregate-return`

Warn if any functions that return structures or unions are defined or called. (In languages where you can return an array, this also elicits a warning.)

`-Wno-aggressive-loop-optimizations`

Warn if in a loop with constant number of iterations the compiler detects undefined behavior in some statement during one or more of the iterations.

`-Wno-attributes`

Do not warn if an unexpected `__attribute__` is used, such as unrecognized attributes, function attributes applied to variables, etc. This does not stop errors for incorrect use of supported attributes.

Warnings about ill-formed uses of standard attributes are upgraded to errors by `-pedantic-errors`.

Additionally, using `-Wno-attributes=`, it is possible to suppress warnings about unknown scoped attributes (in C++11 and C23). For example, `-Wno-attributes=vendor::attr` disables warning about the following declaration:

```
[[vendor::attr]] void f();
```

It is also possible to disable warning about all attributes in a namespace using `-Wno-attributes=vendor::` which prevents warning about both of these declarations:

```
[[vendor::safe]] void f();
[[vendor::unsafe]] void f2();
```

Note that `-Wno-attributes=` does not imply `-Wno-attributes`.

`-Wno-builtin-declaration-mismatch`

Warn if a built-in function is declared with an incompatible signature or as a non-function, or when a built-in function declared with a type that does not include a prototype is called with arguments whose promoted types do not match those expected by the function. When `-Wextra` is specified, also warn when a built-in function that takes arguments is declared without a prototype. The `-Wbuiltin-declaration-mismatch` warning is enabled by default. To avoid the warning include the appropriate header to bring the prototypes of built-in functions into scope.

For example, the call to `memset` below is diagnosed by the warning because the function expects a value of type `size_t` as its argument but the type of `32` is `int`. With `-Wextra`, the declaration of the function is diagnosed as well.

```
extern void* memset ();
void f (void *d)
{
  memset (d, '\0', 32);
}
```

`-Wno-builtin-macro-redefined`

Do not warn if certain built-in macros are redefined. This suppresses warnings for redefinition of `__TIMESTAMP__`, `__TIME__`, `__DATE__`, `__FILE__`, and `__BASE_FILE__`.

`-Wstrict-prototypes` (C and Objective-C only)

Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration that specifies the argument types.)

`-Wold-style-declaration` (C and Objective-C only)

Warn for obsolescent usages, according to the C Standard, in a declaration. For example, warn if storage-class specifiers like `static` are not the first things in a declaration. This warning is also enabled by `-Wextra`.

`-Wold-style-definition` (C and Objective-C only)

Warn if an old-style function definition is used. A warning is given even if there is a previous prototype. A definition using '`()`' is not considered an old-style definition in C23 mode, because it is equivalent to '`(void)`' in that case, but is considered an old-style definition for older standards.

`-Wmissing-parameter-type` (C and Objective-C only)

A function parameter is declared without a type specifier in K&R-style functions:

```
void foo(bar) { }
```

This warning is also enabled by `-Wextra`.

`-Wno-declaration-missing-parameter-type` (C and Objective-C only)

Do not warn if a function declaration contains a parameter name without a type. Such function declarations do not provide a function prototype and prevent most type checking in function calls.

This warning is enabled by default. In C99 and later dialects of C, it is treated as an error. The error can be downgraded to a warning using `-fpermissive` (along with certain other errors), or for this error alone, with `-Wno-error=declaration-missing-parameter-type`.

This warning is upgraded to an error by `-pedantic-errors`.

`-Wmissing-prototypes` (C and Objective-C only)

Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. Use this option to detect global functions that do not have a matching prototype declaration in a header file. This option is not valid for C++ because all function declarations provide prototypes and a non-matching declaration declares an overload rather than conflict with an earlier declaration. Use `-Wmissing-declarations` to detect missing declarations in C++.

`-Wmissing-variable-declarations` (C and Objective-C only)

Warn if a global variable is defined without a previous declaration. Use this option to detect global variables that do not have a matching extern declaration in a header file.

`-Wmissing-declarations`

Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype. Use this option to detect global functions that are not declared in header files. In C, no warnings are issued for functions with previous non-prototype declarations; use `-Wmissing-prototypes` to detect missing prototypes. In C++, no warnings are issued for function templates, or for inline functions, or for functions in anonymous namespaces.

`-Wmissing-field-initializers`

Warn if a structure's initializer has some fields missing. For example, the following code causes such a warning, because `x.h` is implicitly zero:

```
struct s { int f, g, h; };
struct s x = { 3, 4 };
```

In C this option does not warn about designated initializers, so the following modification does not trigger a warning:

```
struct s { int f, g, h; };
struct s x = { .f = 3, .g = 4 };
```

In C this option does not warn about the universal zero initializer '`{ 0 }`':

```
struct s { int f, g, h; };
struct s x = { 0 };
```

Likewise, in C++ this option does not warn about the empty { } initializer, for example:

```
struct s { int f, g, h; };
s x = { };
```

This warning is included in `-Wextra`. To get other `-Wextra` warnings without this one, use `-Wextra -Wno-missing-field-initializers`.

`-Wno-missing-requires`

By default, the compiler warns about a concept-id appearing as a C++20 simple-requirement:

```
bool satisfied = requires { C<T> };
```

Here '`satisfied`' will be true if '`C<T>`' is a valid expression, which it is for all T. Presumably the user meant to write

```
bool satisfied = requires { requires C<T> };
```

so '`satisfied`' is only true if concept '`C`' is satisfied for type '`T`'.

This warning can be disabled with `-Wno-missing-requires`.

`-Wno-missing-template-keyword`

The member access tokens ., -> and :: must be followed by the `template` keyword if the parent object is dependent and the member being named is a template.

```
template <class X>
void DoStuff (X x)
{
  x.template DoSomeOtherStuff<X>(); // Good.
  x.DoMoreStuff<X>(); // Warning, x is dependent.
}
```

In rare cases it is possible to get false positives. To silence this, wrap the expression in parentheses. For example, the following is treated as a template, even where m and N are integers:

```
void NotATemplate (my_class t)
{
  int N = 5;

  bool test = t.m < N > (0); // Treated as a template.
  test = (t.m < N) > (0); // Same meaning, but not treated as a template.
}
```

This warning can be disabled with `-Wno-missing-template-keyword`.

`-Wno-multichar`

Do not warn if a multicharacter constant (‘`''FOOF''`’) is used. Usually they indicate a typo in the user’s code, as they have implementation-defined values, and should not be used in portable code.

`-Wnormalized=[none|id|nfc|nfkc]`

In ISO C and ISO C++, two identifiers are different if they are different sequences of characters. However, sometimes when characters outside the basic ASCII character set are used, you can have two different character sequences that look the same. To avoid confusion, the ISO 10646 standard sets out some *normalization rules* which when applied ensure that two sequences that look the same are turned into the same sequence. GCC can warn you if you are using identifiers that have not been normalized; this option controls that warning.

There are four levels of warning supported by GCC. The default is `-Wnormalized=nfc`, which warns about any identifier that is not in the ISO 10646 “C” normalized form, *NFC*. NFC is the recommended form for most uses. It is equivalent to `-Wnormalized`.

Unfortunately, there are some characters allowed in identifiers by ISO C and ISO C++ that, when turned into NFC, are not allowed in identifiers. That is, there’s no way to use these symbols in portable ISO C or C++ and have all your identifiers in NFC. `-Wnormalized=id` suppresses the warning for these characters. It is hoped that future versions of the standards involved will correct this, which is why this option is not the default.

You can switch the warning off for all characters by writing `-Wnormalized=none` or `-Wno-normalized`. You should only do this if you are using some other normalization scheme (like “D”), because otherwise you can easily create bugs that are literally impossible to see.

Some characters in ISO 10646 have distinct meanings but look identical in some fonts or display methodologies, especially once formatting has been applied. For instance `\u207F`, “SUPERSCRIPT LATIN SMALL LETTER N”, displays just like a regular `n` that has been placed in a superscript. ISO 10646 defines the *NFKC* normalization scheme to convert all these into a standard form as well, and GCC warns if your code is not in NFKC if you use `-Wnormalized=nfkc`. This warning is comparable to warning about every identifier that contains the letter O because

it might be confused with the digit 0, and so is not the default, but may be useful as a local coding convention if the programming environment cannot be fixed to display these characters distinctly.

`-Wno-attribute-warning`

Do not warn about usage of functions (see Declaring Attributes of Functions) declared with `warning` attribute. By default, this warning is enabled. `-Wno-attribute-warning` can be used to disable the warning or `-Wno-error=attribute-warning` can be used to disable the error when compiled with `-Werror` flag.

`-Wno-deprecated`

Do not warn about usage of deprecated features. See Deprecated Features.

`-Wno-deprecated-declarations`

Do not warn about uses of functions (see Declaring Attributes of Functions), variables (see Specifying Attributes of Variables), and types (see Specifying Attributes of Types) marked as deprecated by using the `deprecated` attribute.

`-Wno-overflow`

Do not warn about compile-time overflow in constant expressions.

`-Wno-odr`

Warn about One Definition Rule violations during link-time optimization. Enabled by default.

`-Wopenacc-parallelism`

Warn about potentially suboptimal choices related to OpenACC parallelism.

`-Wno-openmp`

Warn about suspicious OpenMP code.

`-Wopenmp-simd`

Warn if the vectorizer cost model overrides the OpenMP simd directive set by user. The `-fsimd-cost-model=unlimited` option can be used to relax the cost model.

`-Woverride-init` (C and Objective-C only)

Warn if an initialized field without side effects is overridden when using designated initializers (see Designated Initializers).

This warning is included in `-Wextra`. To get other `-Wextra` warnings without this one, use `-Wextra -Wno-override-init`.

`-Wno-override-init-side-effects` (C and Objective-C only)

Do not warn if an initialized field with side effects is overridden when using designated initializers (see Designated Initializers). This warning is enabled by default.

`-Wpacked`

Warn if a structure is given the packed attribute, but the packed attribute has no effect on the layout or size of the structure. Such structures may be mis-aligned for little benefit. For instance, in this code, the variable `f.x` in `struct bar` is misaligned even though `struct bar` does not itself have the packed attribute:

```
struct foo {
  int x;
  char a, b, c, d;
} __attribute__((packed));
struct bar {
  char z;
  struct foo f;
};
```

`-Wnopacked-bitfield-compat`

The 4.1, 4.2 and 4.3 series of GCC ignore the `packed` attribute on bit-fields of type `char`. This was fixed in GCC 4.4 but the change can lead to differences in the structure layout. GCC informs you when the offset of such a field has changed in GCC 4.4. For example there is no longer a 4-bit padding between field `a` and `b` in this structure:

```
struct foo
{
  char a:4;
  char b:8;
} __attribute__ ((packed));
```

This warning is enabled by default. Use `-Wno-packed-bitfield-compat` to disable this warning.

`-Wpacked-not-aligned` (C, C++, Objective-C and Objective-C++ only)

Warn if a structure field with explicitly specified alignment in a packed struct or union is misaligned. For example, a warning will be issued on `struct S`, like, `warning: alignment 1 of 'struct S' is less than 8`, in this code:

```
struct __attribute__ ((aligned (8))) S8 { char a[8]; };
struct __attribute__ ((packed)) S {
  struct S8 s8;
};
```

This warning is enabled by `-Wall`.

`-Wpadded`

Warn if padding is included in a structure, either to align an element of the structure or to align the whole structure. Sometimes when this happens it is possible to rearrange the fields of the structure to reduce the padding and so make the structure smaller.

`-Wredundant-decls`

Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.

`-Wrestrict`

Warn when an object referenced by a `restrict`-qualified parameter (or, in C++, a `__restrict`-qualified parameter) is aliased by another argument, or when copies between such objects overlap. For example, the call to the `strcpy` function below attempts to truncate the string by replacing its initial characters with the last four. However, because the call writes the terminating NUL into `a[4]`, the copies overlap and the call is diagnosed.

```
void foo (void)
{
  char a[] = "abcd1234";
  strcpy (a, a + 4);
  …
}
```

The `-Wrestrict` option detects some instances of simple overlap even without optimization but works best at `-O2` and above. It is included in `-Wall`.

`-Wnested-externs` (C and Objective-C only)

Warn if an `extern` declaration is encountered within a function.

`-Winline`

Warn if a function that is declared as inline cannot be inlined. Even with this option, the compiler does not warn about failures to inline functions declared in system headers.

The compiler uses a variety of heuristics to determine whether or not to inline a function. For example, the compiler takes into account the size of the function being inlined and the amount of inlining that has already been done in the current function. Therefore, seemingly insignificant changes in the source program can cause the warnings produced by `-Winline` to appear or disappear.

`-Winterference-size`

Warn about use of C++17 `std::hardware_destructive_interference_size` without specifying its value with `--param destructive-interference-size`. Also warn about questionable values for that option.

This variable is intended to be used for controlling class layout, to avoid false sharing in concurrent code:

```
struct independent_fields {
  alignas(std::hardware_destructive_interference_size)
    std::atomic<int> one;
  alignas(std::hardware_destructive_interference_size)
    std::atomic<int> two;
};
```

Here ‘`one`’ and ‘`two`’ are intended to be far enough apart that stores to one won’t require accesses to the other to reload the cache line.

By default, `--param destructive-interference-size` and `--param constructive-interference-size` are set based on the current `-mtune` option, typically to the L1 cache line size for the particular target CPU, sometimes to a range if tuning for a generic target. So all translation units that depend on ABI compatibility for the use of these variables must be compiled with the same `-mtune` (or `-mcpu`).

If ABI stability is important, such as if the use is in a header for a library, you should probably not use the hardware interference size variables at all. Alternatively, you can force a particular value with `--param`.

If you are confident that your use of the variable does not affect ABI outside a single build of your project, you can turn off the warning with `-Wno-interference-size`.

`-Wint-in-bool-context`

Warn for suspicious use of integer values where boolean values are expected, such as conditional expressions (?:) using non-boolean integer constants in boolean context, like `if (a <= b ? 2 : 3)`. Or left shifting of signed integers in boolean context, like `for (a = 0; 1 << a; a++);`. Likewise for all kinds of multiplications regardless of the data type. This warning is enabled by `-Wall`.

`-Wno-int-to-pointer-cast`

Suppress warnings from casts to pointer type of an integer of a different size. In C++, casting to a pointer type of smaller size is an error. `Wint-to-pointer-cast` is enabled by default.

`-Wno-pointer-to-int-cast` (C and Objective-C only)

Suppress warnings from casts from a pointer to an integer type of a different size.

`-Winvalid-pch`

Warn if a precompiled header (see Using Precompiled Headers) is found in the search path but cannot be used.

`-Winvalid-utf8`

Warn if an invalid UTF-8 character is found. This warning is on by default for C++23 if `-finput-charset=UTF-8` is used and turned into error with `-pedantic-errors`.

`-Wno-unicode`

Don't diagnose invalid forms of delimited or named escape sequences which are treated as separate tokens. `Wunicode` is enabled by default.

`-Wlong-long`

Warn if `long long` type is used. This is enabled by either `-Wpedantic` or `-Wtraditional` in ISO C90 and C++98 modes. To inhibit the warning messages, use `-Wno-long-long`.

This warning is upgraded to an error by `-pedantic-errors`.

`-Wvariadic-macros`

Warn if variadic macros are used in ISO C90 mode, or if the GNU alternate syntax is used in ISO C99 mode. This is enabled by either `-Wpedantic` or `-Wtraditional`. To inhibit the warning messages, use `-Wno-variadic-macros`.

`-Wno-varargs`

Do not warn upon questionable usage of the macros used to handle variable arguments like `va_start`. These warnings are enabled by default.

`-Wvector-operation-performance`

Warn if vector operation is not implemented via SIMD capabilities of the architecture. Mainly useful for the performance tuning. Vector operation can be implemented `piecewise`, which means that the scalar operation is performed on every vector element; `in parallel`, which means that the vector operation is implemented using scalars of wider type, which normally is more performance efficient; and `as a single scalar`, which means that vector fits into a scalar type.

`-Wvla`

Warn if a variable-length array is used in the code. `-Wno-vla` prevents the `-Wpedantic` warning of the variable-length array.

This warning is upgraded to an error by `-pedantic-errors`.

`-Wvla-larger-than=`*`byte-size`*

If this option is used, the compiler warns for declarations of variable-length arrays whose size is either unbounded, or bounded by an argument that allows the array size to exceed *byte-size* bytes. This is similar to how `-Walloca-larger-than=`*byte-size* works, but with variable-length arrays.

Note that GCC may optimize small variable-length arrays of a known value into plain arrays, so this warning may not get triggered for such arrays.

`-Wvla-larger-than=‘PTRDIFF_MAX’` is enabled by default but is typically only effective when `-ftree-vrp` is active (default for `-O2` and above).

See also `-Walloca-larger-than=`*`byte-size`*.

`-Wno-vla-larger-than`

Disable `-Wvla-larger-than=` warnings. The option is equivalent to `-Wvla-larger-than=‘SIZE_MAX’` or larger.

`-Wvla-parameter`

Warn about redeclarations of functions involving arguments of Variable Length Array types of inconsistent kinds or forms, and enable the detection of out-of-bounds accesses to such parameters by warnings such as `-Warray-bounds`.

If the first function declaration uses the VLA form the bound specified in the array is assumed to be the minimum number of elements expected to be provided in calls to the function and the maximum number of elements accessed by it. Failing to provide arguments of sufficient size or accessing more than the maximum number of elements may be diagnosed.

For example, the warning triggers for the following redeclarations because the first one allows an array of any size to be passed to `f` while the second one specifies that the array argument must have at least `n` elements. In addition, calling `f` with the associated VLA bound parameter in excess of the actual VLA bound triggers a warning as well.

```
void f (int n, int[n]);
// warning: argument 2 previously declared as a VLA
void f (int, int[]);

void g (int n)
{
    if (n > 4)
      return;
    int a[n];
    // warning: access to a by f may be out of bounds
    f (sizeof a, a);
  …
}
```

`-Wvla-parameter` is included in `-Wall`. The `-Warray-parameter` option triggers warnings for similar problems involving ordinary array arguments.

`-Wvolatile-register-var`

Warn if a register variable is declared volatile. The volatile modifier does not inhibit all optimizations that may eliminate reads and/or writes to register variables. This warning is enabled by `-Wall`.

`-Wno-xor-used-as-pow` (C, C++, Objective-C and Objective-C++ only)

Disable warnings about uses of ^, the exclusive or operator, where it appears the code meant exponentiation. Specifically, the warning occurs when the left-hand side is the decimal constant 2 or 10 and the right-hand side is also a decimal constant.

In C and C++, ^ means exclusive or, whereas in some other languages (e.g. TeX and some versions of BASIC) it means exponentiation.

This warning can be silenced by converting one of the operands to hexadecimal as well as by compiling with `-Wno-xor-used-as-pow`.

`-Wdisabled-optimization`

Warn if a requested optimization pass is disabled. This warning does not generally indicate that there is anything wrong with your code; it merely indicates that GCC's optimizers are unable to handle the code effectively. Often, the problem is that your code is too big or too complex; GCC refuses to optimize programs when the optimization itself is likely to take inordinate amounts of time.

`-Wpointer-sign` (C and Objective-C only)

Warn for pointer argument passing or assignment with different signedness. This option is only supported for C and Objective-C. It is implied by `-Wall` and by `-Wpedantic`, which can be disabled with `-Wno-pointer-sign`.

This warning is upgraded to an error by `-pedantic-errors`.

`-Wstack-protector`

This option is only active when `-fstack-protector` is active. It warns about functions that are not protected against stack smashing.

`-Woverlength-strings`

Warn about string constants that are longer than the "minimum maximum" length specified in the C standard. Modern compilers generally allow string constants that are much longer than the standard's minimum limit, but very portable programs should avoid using longer strings.

The limit applies *after* string constant concatenation, and does not count the trailing NUL. In C90, the limit was 509 characters; in C99, it was raised to 4095. C++98 does not specify a normative minimum maximum, so we do not diagnose overlength strings in C++.

This option is implied by `-Wpedantic`, and can be disabled with `-Wno-overlength-strings`.

`-Wunsuffixed-float-constants` (C and Objective-C only)

Issue a warning for any floating constant that does not have a suffix. When used together with `-Wsystem-headers` it warns about such constants in system header files. This can be useful when preparing code to use with the `FLOAT_CONST_DECIMAL64` pragma from the decimal floating-point extension to C99.

`-Wno-lto-type-mismatch`

During the link-time optimization, do not warn about type mismatches in global declarations from different compilation units. Requires `-flto` to be enabled. Enabled by default.

`-Wno-designated-init` (C and Objective-C only)

Suppress warnings when a positional initializer is used to initialize a structure that has been marked with the `designated_init` attribute.

Next: Options That Control Static Analysis, Previous: Options to Control Diagnostic Messages Formatting, Up: GCC Command Options [Contents] [Index]