GN Reference

This page is automatically generated from `gn help --markdown all`.

--args: Specifies build arguments overrides.

```
See "gn help buildargs" for an overview of how build arguments work.

Most operations take a build directory. The build arguments are taken
from the previous build done in that directory. If a command specifies
--args, it will override the previous arguments stored in the build
directory, and use the specified ones.

The args specified will be saved to the build directory for subsequent
commands. Specifying --args="" will clear all build arguments.
```

Formatting

```
The value of the switch is interpreted in GN syntax. For typical usage
of string arguments, you will need to be careful about escaping of
quotes.
```

Examples

```
gn gen out/Default --args="foo=\"bar\""

gn gen out/Default --args='foo="bar" enable=true blah=7'

gn check out/Default --args=""
  Clears existing build args from the directory.

gn desc out/Default --args="some_list=[1, false, \"foo\"]"
```

--[no]color: Forces colored output on or off.

```
Normally GN will try to detect whether it is outputting to a terminal
and will enable or disable color accordingly. Use of these switches
will override the default.
```

Examples

```
gn gen out/Default --color

gn gen out/Default --nocolor
```

--dotfile: Override the name of the ".gn" file.

```
Normally GN loads the ".gn"file  from the source root for some basic
configuration (see "gn help dotfile"). This flag allows you to
use a different file.

Note that this interacts with "--root" in a possibly incorrect way.
It would be nice to test the edge cases and document or fix.
```

--fail-on-unused-args: Treat unused build args as fatal errors.

```
If you set a value in a build's "gn args" and never use it in the
build (in a declare_args() block), GN will normally print an error
but not fail the build.

In many cases engineers would use build args to enable or disable
features that would sometimes get removed. It would by annoying to
block work for typically benign problems. In Chrome in particular,
flags might be configured for build bots in a separate infrastructure
repository, or a declare_args block might be changed in a third party
repository. Treating these errors as blocking forced complex multi-
way patches to land what would otherwise be simple changes.

In some cases, such concerns are not as important, and a mismatch
in build flags between the invoker of the build and the build files
represents a critical mismatch that should be immediately fixed. Such
users can set this flag to force GN to fail in that case.
```

--markdown: Write help output in the Markdown format.

--[no]color: Forces colored output on or off.

```
Normally GN will try to detect whether it is outputting to a terminal
and will enable or disable color accordingly. Use of these switches
will override the default.
```

Examples

```
gn gen out/Default --color

gn gen out/Default --nocolor
```

-q: Quiet mode. Don’t print output on success.

```
This is useful when running as a part of another script.
```

--root: Explicitly specify source root.

```
Normally GN will look up in the directory tree from the current
directory to find a ".gn" file. The source root directory specifies
the meaning of "//" beginning with paths, and the BUILD.gn file
in that directory will be the first thing loaded.

Specifying --root allows GN to do builds in a specific directory
regardless of the current directory.
```

Examples

```
gn gen //out/Default --root=/home/baracko/src
```

```
gn desc //out/Default --root="C:\Users\BObama\My Documents\foo"
```

--runtime-deps-list-file: Save runtime dependencies for targets in file.

```
--runtime-deps-list-file=<filename>

Where <filename> is a text file consisting of the labels, one per
line, of the targets for which runtime dependencies are desired.

See "gn help runtime_deps" for a description of how runtime
dependencies are computed.
```

Runtime deps output file

```
For each target requested, GN will write a separate runtime dependency
file. The runtime dependency file will be in the output directory
alongside the output file of the target, with a ".runtime_deps"
extension. For example, if the target "//foo:bar" is listed in the
input file, and that target produces an output file "bar.so", GN
will create a file "bar.so.runtime_deps" in the build directory.

If a source set, action, copy, or group is listed, the runtime deps
file will correspond to the .stamp file corresponding to that target.
This is probably not useful; the use-case for this feature is
generally executable targets.

The runtime dependency file will list one file per line, with no
escaping. The files will be relative to the root_build_dir. The first
line of the file will be the main output file of the target itself
(in the above example, "bar.so").
```

--script-executable: Set the executable used to execute scripts.

```
By default GN searches the PATH for Python to execute scripts in
action targets and exec_script calls. This flag allows the
specification of a specific Python executable or potentially
```

```
a different language interpreter.
```

--threads: Specify number of worker threads.

```
GN runs many threads to load and run build files. This can make
debugging challenging. Or you may want to experiment with different
values to see how it affects performance.

The parameter is the number of worker threads. This does not count the
main thread (so there are always at least two).
```

Examples

```
gen gen out/Default --threads=1
```

--time: Outputs a summary of how long everything took.

```
Hopefully self-explanatory.
```

Examples

```
gn gen out/Default --time
```

--tracelog: Writes a Chrome-compatible trace log to the given file.

```
The trace log will show file loads, executions, scripts, and writes.
This allows performance analysis of the generation step.

To view the trace, open Chrome and navigate to "chrome://tracing/",
then press "Load" and specify the file you passed to this parameter.
```

Examples

```
gn gen out/Default --tracelog=mytrace.trace
```

-v: Verbose logging.

```
This will spew logging events to the console for debugging issues.
Good luck!
```

gn args [--list] [--args]

```
See also "gn help buildargs" for a more high-level overview of how
build arguments work.
```

Usage

```
gn args <out_dir>
    Open the arguments for the given build directory in an editor
    (as specified by the EDITOR environment variable). If the given
    build directory doesn't exist, it will be created and an empty
    args file will be opened in the editor. You would type something
    like this into that file:
        enable_doom_melon=false
        os="android"

    Note: you can edit the build args manually by editing the file
    "args.gn" in the build directory and then running
    "gn gen <out_dir>".

gn args <out_dir> --list[=<exact_arg>] [--short]
    Lists all build arguments available in the current configuration,
    or, if an exact_arg is specified for the list flag, just that one
    build argument.

    The output will list the declaration location, default value, and
    comment preceeding the declaration. If --short is specified,
```

```
    only the names and values will be printed.

    If the out_dir is specified, the build configuration will be
    taken from that build directory. The reason this is needed is that
    the definition of some arguments is dependent on the build
    configuration, so setting some values might add, remove, or change
    the default values for other arguments. Specifying your exact
    configuration allows the proper arguments to be displayed.

    Instead of specifying the out_dir, you can also use the
    command-line flag to specify the build configuration:
      --args=<exact list of args to use>
```

Examples

```
gn args out/Debug
  Opens an editor with the args for out/Debug.

gn args out/Debug --list --short
  Prints all arguments with their default values for the out/Debug
  build.

gn args out/Debug --list=target_cpu
  Prints information about the "target_cpu" argument for the out/Debug
  build.

gn args --list --args="os=\"android\" enable_doom_melon=true"
  Prints all arguments with the default values for a build with the
  given arguments set (which may affect the values of other
  arguments).
```

gn check []

```
GN's include header checker validates that the includes for C-like
source files match the build dependency graph.

"gn check" is the same thing as "gn gen" with the "--check" flag
except that this command does not write out any build files. It's
intended to be an easy way to manually trigger include file checking.

The <label_pattern> can take exact labels or patterns that match more
than one (although not general regular expressions). If specified,
```

```
only those matching targets will be checked. See
"gn help label_pattern" for details.
```

Command-specific switches

```
--force
    Ignores specifications of "check_includes = false" and checks
    all target's files that match the target label.
```

What gets checked

```
The .gn file may specify a list of targets to be checked. Only these
targets will be checked if no label_pattern is specified on the
command line. Otherwise, the command-line list is used instead. See
"gn help dotfile".

Targets can opt-out from checking with "check_includes = false"
(see "gn help check_includes").

For targets being checked:

  - GN opens all C-like source files in the targets to be checked and
    scans the top for includes.

  - Includes with a "nogncheck" annotation are skipped (see
    "gn help nogncheck").

  - Only includes using "quotes" are checked. <brackets> are assumed
    to be system includes.

  - Include paths are assumed to be relative to either the source root
    or the "root_gen_dir" and must include all the path components.
    (It might be nice in the future to incorporate GN's knowledge of
    the include path to handle other include styles.)

  - GN does not run the preprocessor so will not understand
    conditional includes.

  - Only includes matching known files in the build are checked:
    includes matching unknown paths are ignored.

For an include to be valid:

  - The included file must be in the current target, or there must
```

```
    be a path following only public dependencies to a target with the
    file in it ("gn path" is a good way to diagnose problems).

  - There can be multiple targets with an included file: only one
    needs to be valid for the include to be allowed.

  - If there are only "sources" in a target, all are considered to
    be public and can be included by other targets with a valid public
    dependency path.

  - If a target lists files as "public", only those files are
    able to be included by other targets. Anything in the sources
    will be considered private and will not be includable regardless
    of dependency paths.

  - Ouptuts from actions are treated like public sources on that
    target.

  - A target can include headers from a target that depends on it
    if the other target is annotated accordingly. See
    "gn help allow_circular_includes_from".
```

Advice on fixing problems

```
If you have a third party project that uses relative includes,
it's generally best to exclude that target from checking altogether
via "check_includes = false".

If you have conditional includes, make sure the build conditions
and the preprocessor conditions match, and annotate the line with
"nogncheck" (see "gn help nogncheck" for an example).

If two targets are hopelessly intertwined, use the
"allow_circular_includes_from" annotation. Ideally each should have
identical dependencies so configs inherited from those dependencies
are consistent (see "gn help allow_circular_includes_from").

If you have a standalone header file or files that need to be shared
between a few targets, you can consider making a source_set listing
only those headers as public sources. With only header files, the
source set will be a no-op from a build perspective, but will give a
central place to refer to those headers. That source set's files
will still need to pass "gn check" in isolation.

In rare cases it makes sense to list a header in more than one
target if it could be considered conceptually a member of both.
```

Examples

```
gn check out/Debug
    Check everything.

gn check out/Default //foo:bar
    Check only the files in the //foo:bar target.

gn check out/Default "//foo/*
    Check only the files in targets in the //foo directory tree.
```

gn clean

```
Deletes the contents of the output directory except for args.gn and
creates a Ninja build environment sufficient to regenerate the build.
```

gn desc []

```
Displays information about a given target or config. The build
build parameters will be taken for the build in the given <out_dir>.

The <label or pattern> can be a target label, a config label, or a
label pattern (see "gn help label_pattern"). A label pattern will
only match targets.
```

Possibilities for

```
(If unspecified an overall summary will be displayed.)

all_dependent_configs
allow_circular_includes_from
arflags [--blame]
args
cflags [--blame]
cflags_cc [--blame]
```

```
cflags_cxx [--blame]
check_includes
configs [--tree] (see below)
defines [--blame]
depfile
deps [--all] [--tree] (see below)
include_dirs [--blame]
inputs
ldflags [--blame]
lib_dirs
libs
outputs
public_configs
public
script
sources
testonly
visibility

runtime_deps
    Compute all runtime deps for the given target. This is a
    computed list and does not correspond to any GN variable, unlike
    most other values here.

    The output is a list of file names relative to the build
    directory. See "gn help runtime_deps" for how this is computed.
    This also works with "--blame" to see the source of the
    dependency.
```

Shared flags

```
--all-toolchains
    Normally only inputs in the default toolchain will be included.
    This switch will turn on matching all toolchains.

    For example, a file is in a target might be compiled twice:
    once in the default toolchain and once in a secondary one. Without
    this flag, only the default toolchain one will be matched by
    wildcards. With this flag, both will be matched.
```

Target flags

```
--blame
    Used with any value specified on a config, this will name
```

```
    the config that cause that target to get the flag. This doesn't
    currently work for libs and lib_dirs because those are inherited
    and are more complicated to figure out the blame (patches
    welcome).
```

Configs

```
The "configs" section will list all configs that apply. For targets
this will include configs specified in the "configs" variable of
the target, and also configs pushed onto this target via public
or "all dependent" configs.

Configs can have child configs. Specifying --tree will show the
hierarchy.
```

Printing deps

```
Deps will include all public, private, and data deps (TODO this could
be clarified and enhanced) sorted in order applying. The following
may be used:

--all
    Collects all recursive dependencies and prints a sorted flat list.
    Also usable with --tree (see below).

--as=(buildfile|label|output)
    How to print targets.

    buildfile
        Prints the build files where the given target was declared as
        file names.
    label  (default)
        Prints the label of the target.
    output
        Prints the first output file for the target relative to the
        root build directory.

--testonly=(true|false)
    Restrict outputs to targets with the testonly flag set
    accordingly. When unspecified, the target's testonly flags are
    ignored.

--tree
    Print a dependency tree. By default, duplicates will be elided
```

```
    with "..." but when --all and -tree are used together, no
    eliding will be performed.

    The "deps", "public_deps", and "data_deps" will all be
    included in the tree.

    Tree output can not be used with the filtering or output flags:
    --as, --type, --testonly.

--type=(action|copy|executable|group|loadable_module|shared_library|
        source_set|static_library)
    Restrict outputs to targets matching the given type. If
    unspecified, no filtering will be performed.
```

Note

```
This command will show the full name of directories and source files,
but when directories and source paths are written to the build file,
they will be adjusted to be relative to the build directory. So the
values for paths displayed by this command won't match (but should
mean the same thing).
```

Examples

```
gn desc out/Debug //base:base
    Summarizes the given target.

gn desc out/Foo :base_unittests deps --tree
    Shows a dependency tree of the "base_unittests" project in
    the current directory.

gn desc out/Debug //base defines --blame
    Shows defines set for the //base:base target, annotated by where
    each one was set from.
```

gn format [--dump-tree] [--stdin] BUILD.gn

```
Formats .gn file to a standard format.

The contents of some lists ('sources', 'deps', etc.) will be sorted to
```

```
a canonical order. To suppress this, you can add a comment of the form
"# NOSORT" immediately preceeding the assignment. e.g.

# NOSORT
sources = [
  "z.cc",
  "a.cc",
]
```

Arguments

```
--dry-run
    Does not change or output anything, but sets the process exit code
    based on whether output would be different than what's on disk.
    This is useful for presubmit/lint-type checks.
    - Exit code 0: successful format, matches on disk.
    - Exit code 1: general failure (parse error, etc.)
    - Exit code 2: successful format, but differs from on disk.

--dump-tree
    For debugging only, dumps the parse tree.

--in-place
    Instead of writing the formatted file to stdout, replace the input
    file with the formatted output. If no reformatting is required,
    the input file will not be touched, and nothing printed.

--stdin
    Read input from stdin (and write to stdout). Not compatible with
    --in-place of course.
```

Examples

```
gn format //some/BUILD.gn
gn format some\BUILD.gn
gn format /abspath/some/BUILD.gn
gn format --stdin
```

gn gen: Generate ninja files.

```
gn gen [<ide options>] <out_dir>

Generates ninja files from the current tree and puts them in the given
output directory.

The output directory can be a source-repo-absolute path name such as:
    //out/foo
Or it can be a directory relative to the current directory such as:
    out/foo

See "gn help switches" for the common command-line switches.
```

IDE options

```
GN optionally generates files for IDE. Possibilities for <ide options>

--ide=<ide_name>
    Generate files for an IDE. Currently supported values:
    "eclipse" - Eclipse CDT settings file.
    "vs" - Visual Studio project/solution files.
           (default Visual Studio version: 2015)
    "vs2013" - Visual Studio 2013 project/solution files.
    "vs2015" - Visual Studio 2015 project/solution files.
    "xcode" - Xcode workspace/solution files.
    "qtcreator" - QtCreator project files.

--filters=<path_prefixes>
    Semicolon-separated list of label patterns used to limit the set
    of generated projects (see "gn help label_pattern"). Only
    matching targets will be included to the solution. Only used for
    Visual Studio and Xcode.
```

Visual Studio Flags

```
--sln=<file_name>
    Override default sln file name ("all"). Solution file is written
    to the root build directory.
```

Xcode Flags

```
--workspace=<file_name>
```

```
    Override defaut workspace file name ("all"). The workspace file
    is written to the root build directory.

--ninja-extra-args=<string>
    This string is passed without any quoting to the ninja invocation
    command-line. Can be used to configure ninja flags, like "-j" if
    using goma for example.

--root-target=<target_name>
    Name of the target corresponding to "All" target in Xcode.
    If unset, "All" invokes ninja without any target
    and builds everything.
```

QtCreator Flags

```
--root-target=<target_name>
    Name of the root target for which the QtCreator project will be
    generated to contain files of it and its dependencies. If unset,
    the whole build graph will be omitted.
```

Eclipse IDE Support

```
GN DOES NOT generate Eclipse CDT projects. Instead, it generates a
settings file which can be imported into an Eclipse CDT project. The
XML file contains a list of include paths and defines. Because GN does
not generate a full .cproject definition, it is not possible to
properly define includes/defines for each file individually.
Instead, one set of includes/defines is generated for the entire
project. This works fairly well but may still result in a few indexer
issues here and there.
```

gn help

```
Yo dawg, I heard you like help on your help so I put help on the help
in the help.

You can also use "all" as the parameter to get all help at once.
```

Switches

```
--markdown
    Format output in markdown syntax.
```

Example

```
gn help --markdown all
    Dump all help to stdout in markdown format.
```

gn ls [] [--as=...]

```
    [--type=...] [--testonly=...]

Lists all targets matching the given pattern for the given build
directory. By default, only targets in the default toolchain will
be matched unless a toolchain is explicitly supplied.

If the label pattern is unspecified, list all targets. The label
pattern is not a general regular expression (see
"gn help label_pattern"). If you need more complex expressions,
pipe the result through grep.
```

Options

```
--as=(buildfile|label|output)
    How to print targets.

    buildfile
        Prints the build files where the given target was declared as
        file names.
    label  (default)
        Prints the label of the target.
    output
        Prints the first output file for the target relative to the
        root build directory.

--all-toolchains
    Normally only inputs in the default toolchain will be included.
```

```
    This switch will turn on matching all toolchains.

    For example, a file is in a target might be compiled twice:
    once in the default toolchain and once in a secondary one. Without
    this flag, only the default toolchain one will be matched by
    wildcards. With this flag, both will be matched.

--testonly=(true|false)
    Restrict outputs to targets with the testonly flag set
    accordingly. When unspecified, the target's testonly flags are
    ignored.

--type=(action|copy|executable|group|loadable_module|shared_library|
        source_set|static_library)
    Restrict outputs to targets matching the given type. If
    unspecified, no filtering will be performed.
```

Examples

```
gn ls out/Debug
    Lists all targets in the default toolchain.

gn ls out/Debug "//base/*"
    Lists all targets in the directory base and all subdirectories.

gn ls out/Debug "//base:*"
    Lists all targets defined in //base/BUILD.gn.

gn ls out/Debug //base --as=output
    Lists the build output file for //base:base

gn ls out/Debug --type=executable
    Lists all executables produced by the build.

gn ls out/Debug "//base/*" --as=output | xargs ninja -C out/Debug
    Builds all targets in //base and all subdirectories.

gn ls out/Debug //base --all-toolchains
    Lists all variants of the target //base:base (it may be referenced
    in multiple toolchains).
```

gn path

```
Finds paths of dependencies between two targets. Each unique path
will be printed in one group, and groups will be separate by newlines.
The two targets can appear in either order: paths will be found going
in either direction.

By default, a single path will be printed. If there is a path with
only public dependencies, the shortest public path will be printed.
Otherwise, the shortest path using either public or private
dependencies will be printed. If --with-data is specified, data deps
will also be considered. If there are multiple shortest paths, an
arbitrary one will be selected.
```

Options

```
--all
   Prints all paths found rather than just the first one. Public paths
   will be printed first in order of increasing length, followed by
   non-public paths in order of increasing length.

--public
   Considers only public paths. Can't be used with --with-data.

--with-data
   Additionally follows data deps. Without this flag, only public and
   private linked deps will be followed. Can't be used with --public.
```

Example

```
gn path out/Default //base //tools/gn
```

gn refs (|||@)* [--all]

```
      [--all-toolchains] [--as=...] [--testonly=...] [--type=...]

Finds reverse dependencies (which targets reference something). The
input is a list containing:

 - Target label: The result will be which targets depend on it.
```

```
 - Config label: The result will be which targets list the given
   config in its "configs" or "public_configs" list.

 - Label pattern: The result will be which targets depend on any
   target matching the given pattern. Patterns will not match
   configs. These are not general regular expressions, see
   "gn help label_pattern" for details.

 - File name: The result will be which targets list the given file in
   its "inputs", "sources", "public", "data", or "outputs".
   Any input that does not contain wildcards and does not match a
   target or a config will be treated as a file.

 - Response file: If the input starts with an "@", it will be
   interpreted as a path to a file containing a list of labels or
   file names, one per line. This allows us to handle long lists
   of inputs without worrying about command line limits.
```

Options

```
--all
    When used without --tree, will recurse and display all unique
    dependencies of the given targets. For example, if the input is
    a target, this will output all targets that depend directly or
    indirectly on the input. If the input is a file, this will output
    all targets that depend directly or indirectly on that file.

    When used with --tree, turns off eliding to show a complete tree.

--all-toolchains
    Normally only inputs in the default toolchain will be included.
    This switch will turn on matching all toolchains.

    For example, a file is in a target might be compiled twice:
    once in the default toolchain and once in a secondary one. Without
    this flag, only the default toolchain one will be matched by
    wildcards. With this flag, both will be matched.

--as=(buildfile|label|output)
    How to print targets.

    buildfile
        Prints the build files where the given target was declared as
        file names.
    label  (default)
        Prints the label of the target.
```

```
    output
        Prints the first output file for the target relative to the
        root build directory.

-q
   Quiet. If nothing matches, don't print any output. Without this
   option, if there are no matches there will be an informational
   message printed which might interfere with scripts processing the
   output.

--testonly=(true|false)
    Restrict outputs to targets with the testonly flag set
    accordingly. When unspecified, the target's testonly flags are
    ignored.

--tree
    Outputs a reverse dependency tree from the given target.
    Duplicates will be elided. Combine with --all to see a full
    dependency tree.

    Tree output can not be used with the filtering or output flags:
    --as, --type, --testonly.

--type=(action|copy|executable|group|loadable_module|shared_library|
        source_set|static_library)
    Restrict outputs to targets matching the given type. If
    unspecified, no filtering will be performed.
```

Examples (target input)

```
gn refs out/Debug //tools/gn:gn
    Find all targets depending on the given exact target name.

gn refs out/Debug //base:i18n --as=buildfiles | xargs gvim
    Edit all .gn files containing references to //base:i18n

gn refs out/Debug //base --all
    List all targets depending directly or indirectly on //base:base.

gn refs out/Debug "//base/*"
    List all targets depending directly on any target in //base or
    its subdirectories.

gn refs out/Debug "//base:*"
    List all targets depending directly on any target in
    //base/BUILD.gn.
```

```
gn refs out/Debug //base --tree
    Print a reverse dependency tree of //base:base
```

Examples (file input)

```
gn refs out/Debug //base/macros.h
    Print target(s) listing //base/macros.h as a source.

gn refs out/Debug //base/macros.h --tree
    Display a reverse dependency tree to get to the given file. This
    will show how dependencies will reference that file.

gn refs out/Debug //base/macros.h //base/at_exit.h --all
    Display all unique targets with some dependency path to a target
    containing either of the given files as a source.

gn refs out/Debug //base/macros.h --testonly=true --type=executable
        --all --as=output
    Display the executable file names of all test executables
    potentially affected by a change to the given file.
```

action: Declare a target that runs a script a single time.

```
This target type allows you to run a script a single time to produce
or more output files. If you want to run a script once for each of a
set of input files, see "gn help action_foreach".
```

Inputs

```
In an action the "sources" and "inputs" are treated the same:
they're both input dependencies on script execution with no special
handling. If you want to pass the sources to your script, you must do
so explicitly by including them in the "args". Note also that this
means there is no special handling of paths since GN doesn't know
which of the args are paths and not. You will want to use
rebase_path() to convert paths to be relative to the root_build_dir.

You can dynamically write input dependencies (for incremental rebuilds
if an input file changes) by writing a depfile when the script is run
```

```
(see "gn help depfile"). This is more flexible than "inputs".

If the command line length is very long, you can use response files
to pass args to your script. See "gn help response_file_contents".

It is recommended you put inputs to your script in the "sources"
variable, and stuff like other Python files required to run your
script in the "inputs" variable.

The "deps" and "public_deps" for an action will always be
completed before any part of the action is run so it can depend on
the output of previous steps. The "data_deps" will be built if the
action is built, but may not have completed before all steps of the
action are started. This can give additional parallelism in the build
for runtime-only dependencies.
```

Outputs

```
You should specify files created by your script by specifying them in
the "outputs".

The script will be executed with the given arguments with the current
directory being that of the root build directory. If you pass files
to your script, see "gn help rebase_path" for how to convert
file names to be relative to the build directory (file names in the
sources, outputs, and inputs will be all treated as relative to the
current build file and converted as needed automatically).
```

File name handling

```
All output files must be inside the output directory of the build.
You would generally use |$target_out_dir| or |$target_gen_dir| to
reference the output or generated intermediate file directories,
respectively.
```

Variables

```
args, console, data, data_deps, depfile, deps, inputs, outputs*,
response_file_contents, script*, sources
* = required
```

Example

```
action("run_this_guy_once") {
  script = "doprocessing.py"
  sources = [ "my_configuration.txt" ]
  outputs = [ "$target_gen_dir/insightful_output.txt" ]

  # Our script imports this Python file so we want to rebuild if it
  # changes.
  inputs = [ "helper_library.py" ]

  # Note that we have to manually pass the sources to our script if
  # the script needs them as inputs.
  args = [ "--out", rebase_path(target_gen_dir, root_build_dir) ] +
         rebase_path(sources, root_build_dir)
}
```

action_foreach: Declare a target that runs a script over a set of files.

```
This target type allows you to run a script once-per-file over a set
of sources. If you want to run a script once that takes many files as
input, see "gn help action".
```

Inputs

```
The script will be run once per file in the "sources" variable. The
"outputs" variable should specify one or more files with a source
expansion pattern in it (see "gn help source_expansion"). The output
file(s) for each script invocation should be unique. Normally you
use "{{source_name_part}}" in each output file.

If your script takes additional data as input, such as a shared
configuration file or a Python module it uses, those files should be
listed in the "inputs" variable. These files are treated as
dependencies of each script invocation.

If the command line length is very long, you can use response files
to pass args to your script. See "gn help response_file_contents".

You can dynamically write input dependencies (for incremental rebuilds
```

```
if an input file changes) by writing a depfile when the script is run
(see "gn help depfile"). This is more flexible than "inputs".

The "deps" and "public_deps" for an action will always be
completed before any part of the action is run so it can depend on
the output of previous steps. The "data_deps" will be built if the
action is built, but may not have completed before all steps of the
action are started. This can give additional parallelism in the build
for runtime-only dependencies.
```

Outputs

```
The script will be executed with the given arguments with the current
directory being that of the root build directory. If you pass files
to your script, see "gn help rebase_path" for how to convert
file names to be relative to the build directory (file names in the
sources, outputs, and inputs will be all treated as relative to the
current build file and converted as needed automatically).
```

File name handling

```
All output files must be inside the output directory of the build.
You would generally use |$target_out_dir| or |$target_gen_dir| to
reference the output or generated intermediate file directories,
respectively.
```

Variables

```
args, console, data, data_deps, depfile, deps, inputs, outputs*,
response_file_contents, script*, sources*
* = required
```

Example

```
# Runs the script over each IDL file. The IDL script will generate
# both a .cc and a .h file for each input.
action_foreach("my_idl") {
  script = "idl_processor.py"
  sources = [ "foo.idl", "bar.idl" ]
```

```
  # Our script reads this file each time, so we need to list is as a
  # dependency so we can rebuild if it changes.
  inputs = [ "my_configuration.txt" ]

  # Transformation from source file name to output file names.
  outputs = [ "$target_gen_dir/{{source_name_part}}.h",
              "$target_gen_dir/{{source_name_part}}.cc" ]

  # Note that since "args" is opaque to GN, if you specify paths
  # here, you will need to convert it to be relative to the build
  # directory using "rebase_path()".
  args = [
    "{{source}}",
    "-o",
    rebase_path(relative_target_gen_dir, root_build_dir) +
      "/{{source_name_part}}.h" ]
}
```

assert: Assert an expression is true at generation time.

```
assert(<condition> [, <error string>])

If the condition is false, the build will fail with an error. If the
optional second argument is provided, that string will be printed
with the error message.
```

Examples:

```
assert(is_win)
assert(defined(sources), "Sources must be defined")
```

bundle_data: [iOS/OS X] Declare a target without output.

```
This target type allows to declare data that is required at runtime.
It is used to inform "create_bundle" targets of the files to copy
into generated bundle, see "gn help create_bundle" for help.
```

```
The target must define a list of files as "sources" and a single
"outputs". If there are multiple files, source expansions must be
used to express the output. The output must reference a file inside
of {{bundle_root_dir}}.

This target can be used on all platforms though it is designed only to
generate iOS/OS X bundle. In cross-platform projects, it is advised to
put it behind iOS/Mac conditionals.

See "gn help create_bundle" for more information.
```

Variables

```
sources*, outputs*, deps, data_deps, public_deps, visibility
* = required
```

Examples

```
bundle_data("icudata") {
  sources = [ "sources/data/in/icudtl.dat" ]
  outputs = [ "{{bundle_resources_dir}}/{{source_file_part}}" ]
}

bundle_data("base_unittests_bundle_data]") {
  sources = [ "test/data" ]
  outputs = [
    "{{bundle_resources_dir}}/{{source_root_relative_dir}}/" +
        "{{source_file_part}}"
  ]
}

bundle_data("material_typography_bundle_data") {
  sources = [
    "src/MaterialTypography.bundle/Roboto-Bold.ttf",
    "src/MaterialTypography.bundle/Roboto-Italic.ttf",
    "src/MaterialTypography.bundle/Roboto-Regular.ttf",
    "src/MaterialTypography.bundle/Roboto-Thin.ttf",
  ]
  outputs = [
    "{{bundle_resources_dir}}/MaterialTypography.bundle/"
        "{{source_file_part}}"
  ]
}
```

config: Defines a configuration object.

```
Configuration objects can be applied to targets and specify sets of
compiler flags, includes, defines, etc. They provide a way to
conveniently group sets of this configuration information.

A config is referenced by its label just like a target.

The values in a config are additive only. If you want to remove a flag
you need to remove the corresponding config that sets it. The final
set of flags, defines, etc. for a target is generated in this order:

 1. The values specified directly on the target (rather than using a
    config.
 2. The configs specified in the target's "configs" list, in order.
 3. Public_configs from a breadth-first traversal of the dependency
    tree in the order that the targets appear in "deps".
 4. All dependent configs from a breadth-first traversal of the
    dependency tree in the order that the targets appear in "deps".
```

Variables valid in a config definition

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, ldflags, lib_dirs, libs,
       precompiled_header, precompiled_source
Nested configs: configs
```

Variables on a target used to apply configs

```
all_dependent_configs, configs, public_configs
```

Example

```
config("myconfig") {
  includes = [ "include/common" ]
  defines = [ "ENABLE_DOOM_MELON" ]
```

```
}

executable("mything") {
  configs = [ ":myconfig" ]
}
```

copy: Declare a target that copies files.

File name handling

```
All output files must be inside the output directory of the build.
You would generally use |$target_out_dir| or |$target_gen_dir| to
reference the output or generated intermediate file directories,
respectively.

Both "sources" and "outputs" must be specified. Sources can include
as many files as you want, but there can only be one item in the
outputs list (plural is used for the name for consistency with
other target types).

If there is more than one source file, your output name should specify
a mapping from each source file to an output file name using source
expansion (see "gn help source_expansion"). The placeholders will
look like "{{source_name_part}}", for example.
```

Examples

```
# Write a rule that copies a checked-in DLL to the output directory.
copy("mydll") {
  sources = [ "mydll.dll" ]
  outputs = [ "$target_out_dir/mydll.dll" ]
}

# Write a rule to copy several files to the target generated files
# directory.
copy("myfiles") {
  sources = [ "data1.dat", "data2.dat", "data3.dat" ]

  # Use source expansion to generate output files with the
  # corresponding file names in the gen dir. This will just copy each
  # file.
  outputs = [ "$target_gen_dir/{{source_file_part}}" ]
```

```
}
```

create_bundle: [iOS/OS X] Build an OS X / iOS bundle.

```
This target generates an iOS/OS X bundle (which is a directory with a
well-know structure). This target does not define any sources, instead
they are computed from all "bundle_data" target this one depends on
transitively (the recursion stops at "create_bundle" targets).

The "bundle_*_dir" properties must be defined. They will be used for
the expansion of {{bundle_*_dir}} rules in "bundle_data" outputs.

This target can be used on all platforms though it is designed only to
generate iOS/OS X bundle. In cross-platform projects, it is advised to
put it behind iOS/Mac conditionals.
```

Variables

```
bundle_root_dir*, bundle_resources_dir*, bundle_executable_dir*,
bundle_plugins_dir*, deps, data_deps, public_deps, visibility,
product_type
* = required
```

Example

```
# Defines a template to create an application. On most platform, this
# is just an alias for an "executable" target, but on iOS/OS X, it
# builds an application bundle.
template("app") {
  if (!is_ios && !is_mac) {
    executable(target_name) {
      forward_variables_from(invoker, "*")
    }
  } else {
    app_name = target_name
    gen_path = target_gen_dir

    action("${app_name}_generate_info_plist") {
      script = [ "//build/ios/ios_gen_plist.py" ]
      sources = [ "templates/Info.plist" ]
```

```
  outputs = [ "$gen_path/Info.plist" ]
  args = rebase_path(sources, root_build_dir) +
         rebase_path(outputs, root_build_dir)
}

bundle_data("${app_name}_bundle_info_plist") {
  deps = [ ":${app_name}_generate_info_plist" ]
  sources = [ "$gen_path/Info.plist" ]
  outputs = [ "{{bundle_root_dir}}/Info.plist" ]
}

executable("${app_name}_generate_executable") {
  forward_variables_from(invoker, "*", [
                                          "output_name",
                                          "visibility",
                                        ])
  output_name =
      rebase_path("$gen_path/$app_name", root_build_dir)
}

bundle_data("${app_name}_bundle_executable") {
  deps = [ ":${app_name}_generate_executable" ]
  sources = [ "$gen_path/$app_name" ]
  outputs = [ "{{bundle_executable_dir}}/$app_name" ]
}

create_bundle("${app_name}.app") {
  product_type = "com.apple.product-type.application"
  deps = [
    ":${app_name}_bundle_executable",
    ":${app_name}_bundle_info_plist",
  ]
  if (is_ios) {
    bundle_root_dir = "${root_build_dir}/$target_name"
    bundle_resources_dir = bundle_root_dir
    bundle_executable_dir = bundle_root_dir
    bundle_plugins_dir = bundle_root_dir + "/Plugins"
  } else {
    bundle_root_dir = "${root_build_dir}/target_name/Contents"
    bundle_resources_dir = bundle_root_dir + "/Resources"
    bundle_executable_dir = bundle_root_dir + "/MacOS"
    bundle_plugins_dir = bundle_root_dir + "/Plugins"
  }
}

group(target_name) {
  forward_variables_from(invoker, ["visibility"])
  deps = [ ":${app_name}.app" ]
}
```

```
  }
}
```

declare_args: Declare build arguments.

```
Introduces the given arguments into the current scope. If they are
not specified on the command line or in a toolchain's arguments,
the default values given in the declare_args block will be used.
However, these defaults will not override command-line values.

See also "gn help buildargs" for an overview.

The precise behavior of declare args is:

 1. The declare_arg block executes. Any variables in the enclosing
    scope are available for reading.

 2. At the end of executing the block, any variables set within that
    scope are saved globally as build arguments, with their current
    values being saved as the "default value" for that argument.

 3. User-defined overrides are applied. Anything set in "gn args"
    now overrides any default values. The resulting set of variables
    is promoted to be readable from the following code in the file.

This has some ramifications that may not be obvious:

  - You should not perform difficult work inside a declare_args block
    since this only sets a default value that may be discarded. In
    particular, don't use the result of exec_script() to set the
    default value. If you want to have a script-defined default, set
    some default "undefined" value like [], "", or -1, and after
    the declare_args block, call exec_script if the value is unset by
    the user.

  - Any code inside of the declare_args block will see the default
    values of previous variables defined in the block rather than
    the user-overridden value. This can be surprising because you will
    be used to seeing the overridden value. If you need to make the
    default value of one arg dependent on the possibly-overridden
    value of another, write two separate declare_args blocks:

      declare_args() {
        enable_foo = true
      }
```

```
      declare_args() {
        # Bar defaults to same user-overridden state as foo.
        enable_bar = enable_foo
      }
```

Example

```
declare_args() {
  enable_teleporter = true
  enable_doom_melon = false
}

If you want to override the (default disabled) Doom Melon:
  gn --args="enable_doom_melon=true enable_teleporter=false"
This also sets the teleporter, but it's already defaulted to on so
it will have no effect.
```

defined: Returns whether an identifier is defined.

```
Returns true if the given argument is defined. This is most useful in
templates to assert that the caller set things up properly.

You can pass an identifier:
  defined(foo)
which will return true or false depending on whether foo is defined in
the current scope.

You can also check a named scope:
  defined(foo.bar)
which will return true or false depending on whether bar is defined in
the named scope foo. It will throw an error if foo is not defined or
is not a scope.
```

Example:

```
template("mytemplate") {
  # To help users call this template properly...
  assert(defined(invoker.sources), "Sources must be defined")

  # If we want to accept an optional "values" argument, we don't
```

```
  # want to dereference something that may not be defined.
  if (defined(invoker.values)) {
    values = invoker.values
  } else {
    values = "some default value"
  }
}
```

exec_script: Synchronously run a script and return the output.

```
exec_script(filename,
            arguments = [],
            input_conversion = "",
            file_dependencies = [])

Runs the given script, returning the stdout of the script. The build
generation will fail if the script does not exist or returns a nonzero
exit code.

The current directory when executing the script will be the root
build directory. If you are passing file names, you will want to use
the rebase_path() function to make file names relative to this
path (see "gn help rebase_path").
```

Arguments:

```
filename:
    File name of python script to execute. Non-absolute names will
    be treated as relative to the current build file.

arguments:
    A list of strings to be passed to the script as arguments.
    May be unspecified or the empty list which means no arguments.

input_conversion:
    Controls how the file is read and parsed.
    See "gn help input_conversion".

    If unspecified, defaults to the empty string which causes the
    script result to be discarded. exec script will return None.

dependencies:
```

```
    (Optional) A list of files that this script reads or otherwise
    depends on. These dependencies will be added to the build result
    such that if any of them change, the build will be regenerated and
    the script will be re-run.

    The script itself will be an implicit dependency so you do not
    need to list it.
```

Example:

```
all_lines = exec_script(
    "myscript.py", [some_input], "list lines",
    [ rebase_path("data_file.txt", root_build_dir) ])

# This example just calls the script with no arguments and discards
# the result.
exec_script("//foo/bar/myscript.py")
```

executable: Declare an executable target.

Variables

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, ldflags, lib_dirs, libs,
       precompiled_header, precompiled_source
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, inputs, output_name,
         output_extension, public, sources, testonly, visibility
```

foreach: Iterate over a list.

```
foreach(<loop_var>, <list>) {
  <loop contents>
}

Executes the loop contents block over each item in the list,
assigning the loop_var to each item in sequence. The loop_var will be
```

```
a copy so assigning to it will not mutate the list.

The block does not introduce a new scope, so that variable assignments
inside the loop will be visible once the loop terminates.

The loop variable will temporarily shadow any existing variables with
the same name for the duration of the loop. After the loop terminates
the loop variable will no longer be in scope, and the previous value
(if any) will be restored.
```

Example

```
mylist = [ "a", "b", "c" ]
foreach(i, mylist) {
  print(i)
}

Prints:
a
b
c
```

forward_variables_from: Copies variables from a different scope.

```
forward_variables_from(from_scope, variable_list_or_star,
                       variable_to_not_forward_list = [])

Copies the given variables from the given scope to the local scope
if they exist. This is normally used in the context of templates to
use the values of variables defined in the template invocation to
a template-defined target.

The variables in the given variable_list will be copied if they exist
in the given scope or any enclosing scope. If they do not exist,
nothing will happen and they be left undefined in the current scope.

As a special case, if the variable_list is a string with the value of
"*", all variables from the given scope will be copied. "*" only
copies variables set directly on the from_scope, not enclosing ones.
Otherwise it would duplicate all global variables.
```

```
When an explicit list of variables is supplied, if the variable exists
in the current (destination) scope already, an error will be thrown.
If "*" is specified, variables in the current scope will be
clobbered (the latter is important because most targets have an
implicit configs list, which means it wouldn't work at all if it
didn't clobber).

The sources assignment filter (see "gn help set_sources_assignment_filter")
is never applied by this function. It's assumed than any desired
filtering was already done when sources was set on the from_scope.

If variables_to_not_forward_list is non-empty, then it must contains
a list of variable names that will not be forwarded. This is mostly
useful when variable_list_or_star has a value of "*".
```

Examples

```
# This is a common action template. It would invoke a script with
# some given parameters, and wants to use the various types of deps
# and the visibility from the invoker if it's defined. It also injects
# an additional dependency to all targets.
template("my_test") {
  action(target_name) {
    forward_variables_from(invoker, [ "data_deps", "deps",
                                      "public_deps", "visibility" ])
    # Add our test code to the dependencies.
    # "deps" may or may not be defined at this point.
    if (defined(deps)) {
      deps += [ "//tools/doom_melon" ]
    } else {
      deps = [ "//tools/doom_melon" ]
    }
  }
}

# This is a template around either a target whose type depends on a
# global variable. It forwards all values from the invoker.
template("my_wrapper") {
  target(my_wrapper_target_type, target_name) {
    forward_variables_from(invoker, "*")
  }
}

# A template that wraps another. It adds behavior based on one
# variable, and forwards all others to the nested target.
```

```
template("my_ios_test_app") {
  ios_test_app(target_name) {
    forward_variables_from(invoker, "*", ["test_bundle_name"])
    if (!defined(extra_substitutions)) {
      extra_substitutions = []
    }
    extra_substitutions += [ "BUNDLE_ID_TEST_NAME=$test_bundle_name" ]
  }
}
```

get_label_info: Get an attribute from a target's label.

```
get_label_info(target_label, what)

Given the label of a target, returns some attribute of that target.
The target need not have been previously defined in the same file,
since none of the attributes depend on the actual target definition,
only the label itself.

See also "gn help get_target_outputs".
```

Possible values for the "what" parameter

```
"name"
    The short name of the target. This will match the value of the
    "target_name" variable inside that target's declaration. For the
    label "//foo/bar:baz" this will return "baz".

"dir"
    The directory containing the target's definition, with no slash at
    the end. For the label "//foo/bar:baz" this will return
    "//foo/bar".

"target_gen_dir"
    The generated file directory for the target. This will match the
    value of the "target_gen_dir" variable when inside that target's
    declaration.

"root_gen_dir"
    The root of the generated file tree for the target. This will
    match the value of the "root_gen_dir" variable when inside that
    target's declaration.
```

```
"target_out_dir
    The output directory for the target. This will match the
    value of the "target_out_dir" variable when inside that target's
    declaration.

"root_out_dir"
    The root of the output file tree for the target. This will
    match the value of the "root_out_dir" variable when inside that
    target's declaration.

"label_no_toolchain"
    The fully qualified version of this label, not including the
    toolchain. For the input ":bar" it might return
    "//foo:bar".

"label_with_toolchain"
    The fully qualified version of this label, including the
    toolchain. For the input ":bar" it might return
    "//foo:bar(//toolchain:x64)".

"toolchain"
    The label of the toolchain. This will match the value of the
    "current_toolchain" variable when inside that target's
    declaration.
```

Examples

```
get_label_info(":foo", "name")
# Returns string "foo".

get_label_info("//foo/bar:baz", "gen_dir")
# Returns string "//out/Debug/gen/foo/bar".
```

get_path_info: Extract parts of a file or directory name.

```
get_path_info(input, what)

The first argument is either a string representing a file or
directory name, or a list of such strings. If the input is a list
the return value will be a list containing the result of applying the
rule to each item in the input.
```

Possible values for the “what” parameter

```
"file"
    The substring after the last slash in the path, including the name
    and extension. If the input ends in a slash, the empty string will
    be returned.
      "foo/bar.txt" => "bar.txt"
      "bar.txt" => "bar.txt"
      "foo/" => ""
      "" => ""

"name"
   The substring of the file name not including the extension.
      "foo/bar.txt" => "bar"
      "foo/bar" => "bar"
      "foo/" => ""

"extension"
    The substring following the last period following the last slash,
    or the empty string if not found. The period is not included.
      "foo/bar.txt" => "txt"
      "foo/bar" => ""

"dir"
    The directory portion of the name, not including the slash.
      "foo/bar.txt" => "foo"
      "//foo/bar" => "//foo"
      "foo" => "."

    The result will never end in a slash, so if the resulting
    is empty, the system ("/") or source ("//") roots, a "."
    will be appended such that it is always legal to append a slash
    and a filename and get a valid path.

"out_dir"
    The output file directory corresponding to the path of the
    given file, not including a trailing slash.
      "//foo/bar/baz.txt" => "//out/Default/obj/foo/bar"
"gen_dir"
    The generated file directory corresponding to the path of the
    given file, not including a trailing slash.
      "//foo/bar/baz.txt" => "//out/Default/gen/foo/bar"

"abspath"
    The full absolute path name to the file or directory. It will be
```

```
    resolved relative to the current directory, and then the source-
    absolute version will be returned. If the input is system-
    absolute, the same input will be returned.
      "foo/bar.txt" => "//mydir/foo/bar.txt"
      "foo/" => "//mydir/foo/"
      "//foo/bar" => "//foo/bar"  (already absolute)
      "/usr/include" => "/usr/include"  (already absolute)

    If you want to make the path relative to another directory, or to
    be system-absolute, see rebase_path().
```

Examples

```
sources = [ "foo.cc", "foo.h" ]
result = get_path_info(source, "abspath")
# result will be [ "//mydir/foo.cc", "//mydir/foo.h" ]

result = get_path_info("//foo/bar/baz.cc", "dir")
# result will be "//foo/bar"

# Extract the source-absolute directory name,
result = get_path_info(get_path_info(path, "dir"), "abspath")
```

get_target_outputs: [file list] Get the list of outputs from a target.

```
get_target_outputs(target_label)

Returns a list of output files for the named target. The named target
must have been previously defined in the current file before this
function is called (it can't reference targets in other files because
there isn't a defined execution order, and it obviously can't
reference targets that are defined after the function call).

Only copy and action targets are supported. The outputs from binary
targets will depend on the toolchain definition which won't
necessarily have been loaded by the time a given line of code has run,
and source sets and groups have no useful output file.
```

Return value

```
The names in the resulting list will be absolute file paths (normally
like "//out/Debug/bar.exe", depending on the build directory).

action targets: this will just return the files specified in the
"outputs" variable of the target.

action_foreach targets: this will return the result of applying
the output template to the sources (see "gn help source_expansion").
This will be the same result (though with guaranteed absolute file
paths), as process_file_template will return for those inputs
(see "gn help process_file_template").

binary targets (executables, libraries): this will return a list
of the resulting binary file(s). The "main output" (the actual
binary or library) will always be the 0th element in the result.
Depending on the platform and output type, there may be other output
files as well (like import libraries) which will follow.

source sets and groups: this will return a list containing the path of
the "stamp" file that Ninja will produce once all outputs are
generated. This probably isn't very useful.
```

Example

```
# Say this action generates a bunch of C source files.
action_foreach("my_action") {
  sources = [ ... ]
  outputs = [ ... ]
}

# Compile the resulting source files into a source set.
source_set("my_lib") {
  sources = get_target_outputs(":my_action")
}
```

getenv: Get an environment variable.

```
value = getenv(env_var_name)

Returns the value of the given enironment variable. If the value is
```

```
not found, it will try to look up the variable with the "opposite"
case (based on the case of the first letter of the variable), but
is otherwise case-sensitive.

If the environment variable is not found, the empty string will be
returned. Note: it might be nice to extend this if we had the concept
of "none" in the language to indicate lookup failure.
```

Example:

```
home_dir = getenv("HOME")
```

group: Declare a named group of targets.

```
This target type allows you to create meta-targets that just collect a
set of dependencies into one named target. Groups can additionally
specify configs that apply to their dependents.

Depending on a group is exactly like depending directly on that
group's deps.
```

Variables

```
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
```

Example

```
group("all") {
  deps = [
    "//project:runner",
    "//project:unit_tests",
  ]
}
```

import: Import a file into the current scope.

```
The import command loads the rules and variables resulting from
executing the given file into the current scope.

By convention, imported files are named with a .gni extension.

An import is different than a C++ "include". The imported file is
executed in a standalone environment from the caller of the import
command. The results of this execution are cached for other files that
import the same .gni file.

Note that you can not import a BUILD.gn file that's otherwise used
in the build. Files must either be imported or implicitly loaded as
a result of deps rules, but not both.

The imported file's scope will be merged with the scope at the point
import was called. If there is a conflict (both the current scope and
the imported file define some variable or rule with the same name but
different value), a runtime error will be thrown. Therefore, it's good
practice to minimize the stuff that an imported file defines.

Variables and templates beginning with an underscore '_' are
considered private and will not be imported. Imported files can use
such variables for internal computation without affecting other files.
```

Examples:

```
import("//build/rules/idl_compilation_rule.gni")

# Looks in the current directory.
import("my_vars.gni")
```

loadable_module: Declare a loadable module target.

```
This target type allows you to create an object file that is (and can
only be) loaded and unloaded at runtime.

A loadable module will be specified on the linker line for targets
listing the loadable module in its "deps". If you don't want this
(if you don't need to dynamically load the library at runtime), then
```

```
you should use a "shared_library" target type instead.
```

Variables

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, ldflags, lib_dirs, libs,
       precompiled_header, precompiled_source
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, inputs, output_name,
         output_extension, public, sources, testonly, visibility
```

print: Prints to the console.

```
Prints all arguments to the console separated by spaces. A newline is
automatically appended to the end.

This function is intended for debugging. Note that build files are run
in parallel so you may get interleaved prints. A buildfile may also
be executed more than once in parallel in the context of different
toolchains so the prints from one file may be duplicated or
interleaved with itself.
```

Examples:

```
print("Hello world")

print(sources, deps)
```

process_file_template: Do template expansion over a list of files.

```
process_file_template(source_list, template)

process_file_template applies a template list to a source file list,
```

```
  returning the result of applying each template to each source. This is
  typically used for computing output file names from input files.

  In most cases, get_target_outputs() will give the same result with
  shorter, more maintainable code. This function should only be used
  when that function can't be used (like there's no target or the target
  is defined in another build file).
```

Arguments:

```
  The source_list is a list of file names.

  The template can be a string or a list. If it is a list, multiple
  output strings are generated for each input.

  The template should contain source expansions to which each name in
  the source list is applied. See "gn help source_expansion".
```

Example:

```
  sources = [
    "foo.idl",
    "bar.idl",
  ]
  myoutputs = process_file_template(
      sources,
      [ "$target_gen_dir/{{source_name_part}}.cc",
        "$target_gen_dir/{{source_name_part}}.h" ])

The result in this case will be:
   [ "//out/Debug/foo.cc"
     "//out/Debug/foo.h"
     "//out/Debug/bar.cc"
     "//out/Debug/bar.h" ]
```

read_file: Read a file into a variable.

```
  read_file(filename, input_conversion)

  Whitespace will be trimmed from the end of the file. Throws an error
```

```
if the file can not be opened.
```

Arguments:

```
filename
    Filename to read, relative to the build file.

input_conversion
    Controls how the file is read and parsed.
    See "gn help input_conversion".
```

Example

```
lines = read_file("foo.txt", "list lines")
```

rebase_path: Rebase a file or directory to another location.

```
converted = rebase_path(input,
                        new_base = "",
                        current_base = ".")

Takes a string argument representing a file name, or a list of such
strings and converts it/them to be relative to a different base
directory.

When invoking the compiler or scripts, GN will automatically convert
sources and include directories to be relative to the build directory.
However, if you're passing files directly in the "args" array or
doing other manual manipulations where GN doesn't know something is
a file name, you will need to convert paths to be relative to what
your tool is expecting.

The common case is to use this to convert paths relative to the
current directory to be relative to the build directory (which will
be the current directory when executing scripts).

If you want to convert a file path to be source-absolute (that is,
beginning with a double slash like "//foo/bar"), you should use
the get_path_info() function. This function won't work because it will
always make relative paths, and it needs to support making paths
```

```
relative to the source root, so can't also generate source-absolute
paths without more special-cases.
```

Arguments

```
input
    A string or list of strings representing file or directory names
    These can be relative paths ("foo/bar.txt"), system absolute
    paths ("/foo/bar.txt"), or source absolute paths
    ("//foo/bar.txt").

new_base
    The directory to convert the paths to be relative to. This can be
    an absolute path or a relative path (which will be treated
    as being relative to the current BUILD-file's directory).

    As a special case, if new_base is the empty string (the default),
    all paths will be converted to system-absolute native style paths
    with system path separators. This is useful for invoking external
    programs.

current_base
    Directory representing the base for relative paths in the input.
    If this is not an absolute path, it will be treated as being
    relative to the current build file. Use "." (the default) to
    convert paths from the current BUILD-file's directory.
```

Return value

```
The return value will be the same type as the input value (either a
string or a list of strings). All relative and source-absolute file
names will be converted to be relative to the requested output
System-absolute paths will be unchanged.

Whether an output path will end in a slash will match whether the
corresponding input path ends in a slash. It will return "." or
"./" (depending on whether the input ends in a slash) to avoid
returning empty strings. This means if you want a root path
("//" or "/") not ending in a slash, you can add a dot ("//.").
```

Example

```
# Convert a file in the current directory to be relative to the build
# directory (the current dir when executing compilers and scripts).
foo = rebase_path("myfile.txt", root_build_dir)
# might produce "../../project/myfile.txt".

# Convert a file to be system absolute:
foo = rebase_path("myfile.txt")
# Might produce "D:\source\project\myfile.txt" on Windows or
# "/home/you/source/project/myfile.txt" on Linux.

# Typical usage for converting to the build directory for a script.
action("myscript") {
  # Don't convert sources, GN will automatically convert these to be
  # relative to the build directory when it constructs the command
  # line for your script.
  sources = [ "foo.txt", "bar.txt" ]

  # Extra file args passed manually need to be explicitly converted
  # to be relative to the build directory:
  args = [
    "--data",
    rebase_path("//mything/data/input.dat", root_build_dir),
    "--rel",
    rebase_path("relative_path.txt", root_build_dir)
  ] + rebase_path(sources, root_build_dir)
}
```

set_default_toolchain: Sets the default toolchain name.

```
set_default_toolchain(toolchain_label)

The given label should identify a toolchain definition (see
"help toolchain"). This toolchain will be used for all targets
unless otherwise specified.

This function is only valid to call during the processing of the build
configuration file. Since the build configuration file is processed
separately for each toolchain, this function will be a no-op when
called under any non-default toolchains.

For example, the default toolchain should be appropriate for the
current environment. If the current environment is 32-bit and
somebody references a target with a 64-bit toolchain, we wouldn't
want processing of the build config file for the 64-bit toolchain to
```

```
reset the default toolchain to 64-bit, we want to keep it 32-bits.
```

Argument:

```
toolchain_label
    Toolchain name.
```

Example:

```
set_default_toolchain("//build/config/win:vs32")
```

set_defaults: Set default values for a target type.

```
set_defaults(<target_type_name>) { <values...> }

Sets the default values for a given target type. Whenever
target_type_name is seen in the future, the values specified in
set_default's block will be copied into the current scope.

When the target type is used, the variable copying is very strict.
If a variable with that name is already in scope, the build will fail
with an error.

set_defaults can be used for built-in target types ("executable",
"shared_library", etc.) and custom ones defined via the "template"
command.
```

Example:

```
set_defaults("static_library") {
  configs = [ "//tools/mything:settings" ]
}

static_library("mylib")
  # The configs will be auto-populated as above. You can remove it if
  # you don't want the default for a particular default:
  configs -= "//tools/mything:settings"
}
```

set_sources_assignment_filter: Set a pattern to filter source files.

```
The sources assignment filter is a list of patterns that remove files
from the list implicitly whenever the "sources" variable is
assigned to. This is intended to be used to globally filter out files
with platform-specific naming schemes when they don't apply, for
example, you may want to filter out all "*_win.cc" files on non-
Windows platforms.

Typically this will be called once in the master build config script
to set up the filter for the current platform. Subsequent calls will
overwrite the previous values.

If you want to bypass the filter and add a file even if it might
be filtered out, call set_sources_assignment_filter([]) to clear the
list of filters. This will apply until the current scope exits
```

How to use patterns

```
File patterns are VERY limited regular expressions. They must match
the entire input string to be counted as a match. In regular
expression parlance, there is an implicit "^...$" surrounding your
input. If you want to match a substring, you need to use wildcards at
the beginning and end.

There are only two special tokens understood by the pattern matcher.
Everything else is a literal.

 * Matches zero or more of any character. It does not depend on the
   preceding character (in regular expression parlance it is
   equivalent to ".*").

\b Matches a path boundary. This will match the beginning or end of
   a string, or a slash.
```

Pattern examples

```
"*asdf*"
    Matches a string containing "asdf" anywhere.

"asdf"
    Matches only the exact string "asdf".

"*.cc"
    Matches strings ending in the literal ".cc".

"\bwin/*"
    Matches "win/foo" and "foo/win/bar.cc" but not "iwin/foo".
```

Sources assignment example

```
# Filter out all _win files.
set_sources_assignment_filter([ "*_win.cc", "*_win.h" ])
sources = [ "a.cc", "b_win.cc" ]
print(sources)
# Will print [ "a.cc" ]. b_win one was filtered out.
```

shared_library: Declare a shared library target.

```
A shared library will be specified on the linker line for targets
listing the shared library in its "deps". If you don't want this
(say you dynamically load the library at runtime), then you should
depend on the shared library via "data_deps" or, on Darwin
platforms, use a "loadable_module" target type instead.
```

Variables

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, ldflags, lib_dirs, libs,
       precompiled_header, precompiled_source
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, inputs, output_name,
         output_extension, public, sources, testonly, visibility
```

source_set: Declare a source set target.

```
A source set is a collection of sources that get compiled, but are not
linked to produce any kind of library. Instead, the resulting object
files are implicitly added to the linker line of all targets that
depend on the source set.

In most cases, a source set will behave like a static library, except
no actual library file will be produced. This will make the build go
a little faster by skipping creation of a large static library, while
maintaining the organizational benefits of focused build targets.

The main difference between a source set and a static library is
around handling of exported symbols. Most linkers assume declaring
a function exported means exported from the static library. The linker
can then do dead code elimination to delete code not reachable from
exported functions.

A source set will not do this code elimination since there is no link
step. This allows you to link many sources sets into a shared library
and have the "exported symbol" notation indicate "export from the
final shared library and not from the intermediate targets." There is
no way to express this concept when linking multiple static libraries
into a shared library.
```

Variables

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, ldflags, lib_dirs, libs,
       precompiled_header, precompiled_source
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, inputs, output_name,
         output_extension, public, sources, testonly, visibility
```

static_library: Declare a static library target.

```
Make a ".a" / ".lib" file.

If you only need the static library for intermediate results in the
build, you should consider a source_set instead since it will skip
```

```
the (potentially slow) step of creating the intermediate library file.
```

Variables

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, ldflags, lib_dirs, libs,
       precompiled_header, precompiled_source
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, inputs, output_name,
         output_extension, public, sources, testonly, visibility
```

target: Declare an target with the given programmatic type.

```
target(target_type_string, target_name_string) { ... }

The target() function is a way to invoke a built-in target or template
with a type determined at runtime. This is useful for cases where the
type of a target might not be known statically.

Only templates and built-in target functions are supported for the
target_type_string parameter. Arbitrary functions, configs, and
toolchains are not supported.

The call:
  target("source_set", "doom_melon") {
Is equivalent to:
  source_set("doom_melon") {
```

Example

```
if (foo_build_as_shared) {
  my_type = "shared_library"
} else {
  my_type = "source_set"
}

target(my_type, "foo") {
  ...
}
```

template: Define a template rule.

```
A template defines a custom name that acts like a function. It
provides a way to add to the built-in target types.

The template() function is used to declare a template. To invoke the
template, just use the name of the template like any other target
type.

Often you will want to declare your template in a special file that
other files will import (see "gn help import") so your template
rule can be shared across build files.
```

Variables and templates:

```
When you call template() it creates a closure around all variables
currently in scope with the code in the template block. When the
template is invoked, the closure will be executed.

When the template is invoked, the code in the caller is executed and
passed to the template code as an implicit "invoker" variable. The
template uses this to read state out of the invoking code.

One thing explicitly excluded from the closure is the "current
directory" against which relative file names are resolved. The
current directory will be that of the invoking code, since typically
that code specifies the file names. This means all files internal
to the template should use absolute names.

A template will typically forward some or all variables from the
invoking scope to a target that it defines. Often, such variables
might be optional. Use the pattern:

  if (defined(invoker.deps)) {
    deps = invoker.deps
  }

The function forward_variables_from() provides a shortcut to forward
one or more or possibly all variables in this manner:

  forward_variables_from(invoker, ["deps", "public_deps"])
```

Target naming:

```
Your template should almost always define a built-in target with the
name the template invoker specified. For example, if you have an IDL
template and somebody does:
  idl("foo") {...
you will normally want this to expand to something defining a
source_set or static_library named "foo" (among other things you may
need). This way, when another target specifies a dependency on
"foo", the static_library or source_set will be linked.

It is also important that any other targets your template expands to
have globally unique names, or you will get collisions.

Access the invoking name in your template via the implicit
"target_name" variable. This should also be the basis for how other
targets that a template expands to ensure uniqueness.

A typical example would be a template that defines an action to
generate some source files, and a source_set to compile that source.
Your template would name the source_set "target_name" because
that's what you want external targets to depend on to link your code.
And you would name the action something like "${target_name}_action"
to make it unique. The source set would have a dependency on the
action to make it run.
```

Example of defining a template:

```
template("my_idl") {
  # Be nice and help callers debug problems by checking that the
  # variables the template requires are defined. This gives a nice
  # message rather than giving the user an error about an
  # undefined variable in the file defining the template
  #
  # You can also use defined() to give default values to variables
  # unspecified by the invoker.
  assert(defined(invoker.sources),
         "Need sources in $target_name listing the idl files.")

  # Name of the intermediate target that does the code gen. This must
  # incorporate the target name so it's unique across template
  # instantiations.
  code_gen_target_name = target_name + "_code_gen"
```

```
  # Intermediate target to convert IDL to C source. Note that the name
  # is based on the name the invoker of the template specified. This
  # way, each time the template is invoked we get a unique
  # intermediate action name (since all target names are in the global
  # scope).
  action_foreach(code_gen_target_name) {
    # Access the scope defined by the invoker via the implicit
    # "invoker" variable.
    sources = invoker.sources

    # Note that we need an absolute path for our script file name.
    # The current directory when executing this code will be that of
    # the invoker (this is why we can use the "sources" directly
    # above without having to rebase all of the paths). But if we need
    # to reference a script relative to the template file, we'll need
    # to use an absolute path instead.
    script = "//tools/idl/idl_code_generator.py"

    # Tell GN how to expand output names given the sources.
    # See "gn help source_expansion" for more.
    outputs = [ "$target_gen_dir/{{source_name_part}}.cc",
                "$target_gen_dir/{{source_name_part}}.h" ]
  }

  # Name the source set the same as the template invocation so
  # instancing this template produces something that other targets
  # can link to in their deps.
  source_set(target_name) {
    # Generates the list of sources, we get these from the
    # action_foreach above.
    sources = get_target_outputs(":$code_gen_target_name")

    # This target depends on the files produced by the above code gen
    # target.
    deps = [ ":$code_gen_target_name" ]
  }
}
```

Example of invoking the resulting template:

```
# This calls the template code above, defining target_name to be
# "foo_idl_files" and "invoker" to be the set of stuff defined in
# the curly brackets.
my_idl("foo_idl_files") {
  # Goes into the template as "invoker.sources".
```

```
  sources = [ "foo.idl", "bar.idl" ]
}

# Here is a target that depends on our template.
executable("my_exe") {
  # Depend on the name we gave the template call above. Internally,
  # this will produce a dependency from executable to the source_set
  # inside the template (since it has this name), which will in turn
  # depend on the code gen action.
  deps = [ ":foo_idl_files" ]
}
```

tool: Specify arguments to a toolchain tool.

Usage:

```
tool(<tool type>) {
  <tool variables...>
}
```

Tool types

```
  Compiler tools:
    "cc": C compiler
    "cxx": C++ compiler
    "objc": Objective C compiler
    "objcxx": Objective C++ compiler
    "rc": Resource compiler (Windows .rc files)
    "asm": Assembler

  Linker tools:
    "alink": Linker for static libraries (archives)
    "solink": Linker for shared libraries
    "link": Linker for executables

  Other tools:
    "stamp": Tool for creating stamp files
    "copy": Tool to copy files.

  Platform specific tools:
    "copy_bundle_data": [iOS, OS X] Tool to copy files in a bundle.
    "compile_xcassets": [iOS, OS X] Tool to compile asset catalogs.
```

Tool variables

```
command  [string with substitutions]
    Valid for: all tools (required)

    The command to run.

default_output_dir  [string with substitutions]
    Valid for: linker tools

    Default directory name for the output file relative to the
    root_build_dir. It can contain other substitution patterns.
    This will be the default value for the {{output_dir}} expansion
    (discussed below) but will be overridden by the "output_dir"
    variable in a target, if one is specified.

    GN doesn't do anything with this string other than pass it
    along, potentially with target-specific overrides. It is the
    tool's job to use the expansion so that the files will be in
    the right place.

default_output_extension  [string]
    Valid for: linker tools

    Extension for the main output of a linkable tool. It includes
    the leading dot. This will be the default value for the
    {{output_extension}} expansion (discussed below) but will be
    overridden by by the "output extension" variable in a target,
    if one is specified. Empty string means no extension.

    GN doesn't actually do anything with this extension other than
    pass it along, potentially with target-specific overrides. One
    would typically use the {{output_extension}} value in the
    "outputs" to read this value.

    Example: default_output_extension = ".exe"

depfile  [string with substitutions]
    Valid for: compiler tools (optional)

    If the tool can write ".d" files, this specifies the name of
    the resulting file. These files are used to list header file
    dependencies (or other implicit input dependencies) that are
    discovered at build time. See also "depsformat".
```

```
      Example: depfile = "{{output}}.d"

  depsformat  [string]
      Valid for: compiler tools (when depfile is specified)

      Format for the deps outputs. This is either "gcc" or "msvc".
      See the ninja documentation for "deps" for more information.

      Example: depsformat = "gcc"

  description  [string with substitutions, optional]
      Valid for: all tools

      What to print when the command is run.

      Example: description = "Compiling {{source}}"

  lib_switch  [string, optional, link tools only]
  lib_dir_switch  [string, optional, link tools only]
      Valid for: Linker tools except "alink"

      These strings will be prepended to the libraries and library
      search directories, respectively, because linkers differ on how
      specify them. If you specified:
        lib_switch = "-l"
        lib_dir_switch = "-L"
      then the "{{libs}}" expansion for [ "freetype", "expat"]
      would be "-lfreetype -lexpat".

  outputs  [list of strings with substitutions]
      Valid for: Linker and compiler tools (required)

      An array of names for the output files the tool produces. These
      are relative to the build output directory. There must always be
      at least one output file. There can be more than one output (a
      linker might produce a library and an import library, for
      example).

      This array just declares to GN what files the tool will
      produce. It is your responsibility to specify the tool command
      that actually produces these files.

      If you specify more than one output for shared library links,
      you should consider setting link_output, depend_output, and
      runtime_link_output. Otherwise, the first entry in the
      outputs list should always be the main output which will be
      linked to.

      Example for a compiler tool that produces .obj files:
```

```
      outputs = [
        "{{source_out_dir}}/{{source_name_part}}.obj"
      ]

    Example for a linker tool that produces a .dll and a .lib. The
    use of {{target_output_name}}, {{output_extension}} and
    {{output_dir}} allows the target to override these values.
      outputs = [
        "{{output_dir}}/{{target_output_name}}{{output_extension}}",
        "{{output_dir}}/{{target_output_name}}.lib",
      ]

  link_output  [string with substitutions]
  depend_output  [string with substitutions]
  runtime_link_output  [string with substitutions]
    Valid for: "solink" only (optional)

    These three files specify which of the outputs from the solink
    tool should be used for linking and dependency tracking. These
    should match entries in the "outputs". If unspecified, the
    first item in the "outputs" array will be used for all. See
    "Separate linking and dependencies for shared libraries"
    below for more. If link_output is set but runtime_link_output
    is not set, runtime_link_output defaults to link_output.

    On Windows, where the tools produce a .dll shared library and
    a .lib import library, you will want the first two to be the
    import library and the third one to be the .dll file.
    On Linux, if you're not doing the separate linking/dependency
    optimization, all of these should be the .so output.

  output_prefix  [string]
    Valid for: Linker tools (optional)

    Prefix to use for the output name. Defaults to empty. This
    prefix will be prepended to the name of the target (or the
    output_name if one is manually specified for it) if the prefix
    is not already there. The result will show up in the
    {{output_name}} substitution pattern.

    Individual targets can opt-out of the output prefix by setting:
      output_prefix_override = true
    (see "gn help output_prefix_override").

    This is typically used to prepend "lib" to libraries on
    Posix systems:
      output_prefix = "lib"

  precompiled_header_type  [string]
```

```
      Valid for: "cc", "cxx", "objc", "objcxx"

      Type of precompiled headers. If undefined or the empty string,
      precompiled headers will not be used for this tool. Otherwise
      use "gcc" or "msvc".

      For precompiled headers to be used for a given target, the
      target (or a config applied to it) must also specify a
      "precompiled_header" and, for "msvc"-style headers, a
      "precompiled_source" value. If the type is "gcc", then both
      "precompiled_header" and "precompiled_source" must resolve
      to the same file, despite the different formats required for each.
      See "gn help precompiled_header" for more.

  restat  [boolean]
      Valid for: all tools (optional, defaults to false)

      Requests that Ninja check the file timestamp after this tool has
      run to determine if anything changed. Set this if your tool has
      the ability to skip writing output if the output file has not
      changed.

      Normally, Ninja will assume that when a tool runs the output
      be new and downstream dependents must be rebuild. When this is
      set to trye, Ninja can skip rebuilding downstream dependents for
      input changes that don't actually affect the output.

      Example:
        restat = true

  rspfile  [string with substitutions]
      Valid for: all tools (optional)

      Name of the response file. If empty, no response file will be
      used. See "rspfile_content".

  rspfile_content  [string with substitutions]
      Valid for: all tools (required when "rspfile" is specified)

      The contents to be written to the response file. This may
      include all or part of the command to send to the tool which
      allows you to get around OS command-line length limits.

      This example adds the inputs and libraries to a response file,
      but passes the linker flags directly on the command line:
        tool("link") {
          command = "link -o {{output}} {{ldflags}} @{{output}}.rsp"
          rspfile = "{{output}}.rsp"
          rspfile_content = "{{inputs}} {{solibs}} {{libs}}"
```

```
}
```

Expansions for tool variables

```
All paths are relative to the root build directory, which is the
current directory for running all tools. These expansions are
available to all tools:

  {{label}}
      The label of the current target. This is typically used in the
      "description" field for link tools. The toolchain will be
      omitted from the label for targets in the default toolchain, and
      will be included for targets in other toolchains.

  {{label_name}}
      The short name of the label of the target. This is the part
      after the colon. For "//foo/bar:baz" this will be "baz".
      Unlike {{target_output_name}}, this is not affected by the
      "output_prefix" in the tool or the "output_name" set
      on the target.

  {{output}}
      The relative path and name of the output(s) of the current
      build step. If there is more than one output, this will expand
      to a list of all of them.
      Example: "out/base/my_file.o"

  {{target_gen_dir}}
  {{target_out_dir}}
      The directory of the generated file and output directories,
      respectively, for the current target. There is no trailing
      slash. See also {{output_dir}} for linker tools.
      Example: "out/base/test"

  {{target_output_name}}
      The short name of the current target with no path information,
      or the value of the "output_name" variable if one is specified
      in the target. This will include the "output_prefix" if any.
      See also {{label_name}}.
      Example: "libfoo" for the target named "foo" and an
      output prefix for the linker tool of "lib".

Compiler tools have the notion of a single input and a single output,
along with a set of compiler-specific flags. The following expansions
are available:
```

```
    {{asmflags}}
    {{cflags}}
    {{cflags_c}}
    {{cflags_cc}}
    {{cflags_objc}}
    {{cflags_objcc}}
    {{defines}}
    {{include_dirs}}
        Strings correspond that to the processed flags/defines/include
        directories specified for the target.
        Example: "--enable-foo --enable-bar"

        Defines will be prefixed by "-D" and include directories will
        be prefixed by "-I" (these work with Posix tools as well as
        Microsoft ones).

    {{source}}
        The relative path and name of the current input file.
        Example: "../../base/my_file.cc"

    {{source_file_part}}
        The file part of the source including the extension (with no
        directory information).
        Example: "foo.cc"

    {{source_name_part}}
        The filename part of the source file with no directory or
        extension.
        Example: "foo"

    {{source_gen_dir}}
    {{source_out_dir}}
        The directory in the generated file and output directories,
        respectively, for the current input file. If the source file
        is in the same directory as the target is declared in, they will
        will be the same as the "target" versions above.
        Example: "gen/base/test"

  Linker tools have multiple inputs and (potentially) multiple outputs
  The static library tool ("alink") is not considered a linker tool.
  The following expansions are available:

    {{inputs}}
    {{inputs_newline}}
        Expands to the inputs to the link step. This will be a list of
        object files and static libraries.
        Example: "obj/foo.o obj/bar.o obj/somelibrary.a"

        The "_newline" version will separate the input files with
```

newlines instead of spaces. This is useful in response files: some linkers can take a "-filelist" flag which expects newline separated files, and some Microsoft tools have a fixed-sized buffer for parsing each line of a response file.

{{ldflags}}
Expands to the processed set of ldflags and library search paths specified for the target.
Example: "-m64 -fPIC -pthread -L/usr/local/mylib"

{{libs}}
Expands to the list of system libraries to link to. Each will be prefixed by the "lib_prefix".

As a special case to support Mac, libraries with names ending in ".framework" will be added to the {{libs}} with "-framework" preceeding it, and the lib prefix will be ignored.

Example: "-lfoo -lbar"

{{output_dir}}
The value of the "output_dir" variable in the target, or the the value of the "default_output_dir" value in the tool if the target does not override the output directory. This will be relative to the root_build_dir and will not end in a slash. Will be "." for output to the root_build_dir.

This is subtly different than {{target_out_dir}} which is defined by GN based on the target's path and not overridable. {{output_dir}} is for the final output, {{target_out_dir}} is generally for object files and other outputs.

Usually {{output_dir}} would be defined in terms of either {{target_out_dir}} or {{root_out_dir}}

{{output_extension}}
The value of the "output_extension" variable in the target, or the value of the "default_output_extension" value in the tool if the target does not specify an output extension.
Example: ".so"

{{solibs}}
Extra libraries from shared library dependencide not specified in the {{inputs}}. This is the list of link_output files from shared libraries (if the solink tool specifies a "link_output" variable separate from the "depend_output").

These should generally be treated the same as libs by your tool.
Example: "libfoo.so libbar.so"

The static library ("alink") tool allows {{arflags}} plus the common tool substitutions.

The copy tool allows the common compiler/linker substitutions, plus {{source}} which is the source of the copy. The stamp tool allows only the common tool substitutions.

The copy_bundle_data and compile_xcassets tools only allows the common tool substitutions. Both tools are required to create iOS/OS X bundles and need only be defined on those platforms.

The copy_bundle_data tool will be called with one source and needs to copy (optionally optimizing the data representation) to its output. It may be called with a directory as input and it needs to be recursively copied.

The compile_xcassets tool will be called with one or more source (each an asset catalog) that needs to be compiled to a single output.

Separate linking and dependencies for shared libraries

Shared libraries are special in that not all changes to them require that dependent targets be re-linked. If the shared library is changed but no imports or exports are different, dependent code needn't be relinked, which can speed up the build.

If your link step can output a list of exports from a shared library and writes the file only if the new one is different, the timestamp of this file can be used for triggering re-links, while the actual shared library would be used for linking.

You will need to specify
  restat = true
in the linker tool to make this work, so Ninja will detect if the timestamp of the dependency file has changed after linking (otherwise it will always assume that running a command updates the output):

```
  tool("solink") {
    command = "..."
    outputs = [
      "{{output_dir}}/{{target_output_name}}{{output_extension}}",
      "{{output_dir}}/{{target_output_name}}{{output_extension}}.TOC",
    ]
    link_output =
      "{{output_dir}}/{{target_output_name}}{{output_extension}}"
```

```
    depend_output =
      "{{output_dir}}/{{target_output_name}}{{output_extension}}.TOC"
    restat = true
  }
```

Example

```
toolchain("my_toolchain") {
  # Put these at the top to apply to all tools below.
  lib_prefix = "-l"
  lib_dir_prefix = "-L"

  tool("cc") {
    command = "gcc {{source}} -o {{output}}"
    outputs = [ "{{source_out_dir}}/{{source_name_part}}.o" ]
    description = "GCC {{source}}"
  }
  tool("cxx") {
    command = "g++ {{source}} -o {{output}}"
    outputs = [ "{{source_out_dir}}/{{source_name_part}}.o" ]
    description = "G++ {{source}}"
  }
}
```

toolchain: Defines a toolchain.

```
A toolchain is a set of commands and build flags used to compile the
source code. You can have more than one toolchain in use at once in
a build.
```

Functions and variables

```
tool()
  The tool() function call specifies the commands commands to run for
  a given step. See "gn help tool".

toolchain_args()
  List of arguments to pass to the toolchain when invoking this
  toolchain. This applies only to non-default toolchains. See
  "gn help toolchain_args" for more.
```

```
deps
  Dependencies of this toolchain. These dependencies will be resolved
  before any target in the toolchain is compiled. To avoid circular
  dependencies these must be targets defined in another toolchain.

  This is expressed as a list of targets, and generally these targets
  will always specify a toolchain:
    deps = [ "//foo/bar:baz(//build/toolchain:bootstrap)" ]

  This concept is somewhat inefficient to express in Ninja (it
  requires a lot of duplicate of rules) so should only be used when
  absolutely necessary.

concurrent_links
  In integer expressing the number of links that Ninja will perform in
  parallel. GN will create a pool for shared library and executable
  link steps with this many processes. Since linking is memory- and
  I/O-intensive, projects with many large targets may want to limit
  the number of parallel steps to avoid overloading the computer.
  Since creating static libraries is generally not as intensive
  there is no limit to "alink" steps.

  Defaults to 0 which Ninja interprets as "no limit".

  The value used will be the one from the default toolchain of the
  current build.
```

Invoking targets in toolchains:

```
By default, when a target depends on another, there is an implicit
toolchain label that is inherited, so the dependee has the same one
as the dependent.

You can override this and refer to any other toolchain by explicitly
labeling the toolchain to use. For example:
  data_deps = [ "//plugins:mine(//toolchains:plugin_toolchain)" ]
The string "//build/toolchains:plugin_toolchain" is a label that
identifies the toolchain declaration for compiling the sources.

To load a file in an alternate toolchain, GN does the following:

 1. Loads the file with the toolchain definition in it (as determined
    by the toolchain label).
 2. Re-runs the master build configuration file, applying the
    arguments specified by the toolchain_args section of the toolchain
```

```
    definition (see "gn help toolchain_args").
 3. Loads the destination build file in the context of the
    configuration file in the previous step.
```

Example:

```
toolchain("plugin_toolchain") {
  concurrent_links = 8

  tool("cc") {
    command = "gcc {{source}}"
    ...
  }

  toolchain_args() {
    is_plugin = true
    is_32bit = true
    is_64bit = false
  }
}
```

toolchain_args: Set build arguments for toolchain build setup.

```
Used inside a toolchain definition to pass arguments to an alternate
toolchain's invocation of the build.

When you specify a target using an alternate toolchain, the master
build configuration file is re-interpreted in the context of that
toolchain (see "gn help toolchain"). The toolchain_args function
allows you to control the arguments passed into this alternate
invocation of the build.

Any default system arguments or arguments passed in on the command-
line will also be passed to the alternate invocation unless explicitly
overridden by toolchain_args.

The toolchain_args will be ignored when the toolchain being defined
is the default. In this case, it's expected you want the default
argument values.

See also "gn help buildargs" for an overview of these arguments.
```

Example:

```
toolchain("my_weird_toolchain") {
  ...
  toolchain_args() {
    # Override the system values for a generic Posix system.
    is_win = false
    is_posix = true

    # Pass this new value for specific setup for my toolchain.
    is_my_weird_system = true
  }
}
```

write_file: Write a file to disk.

```
write_file(filename, data)

If data is a list, the list will be written one-item-per-line with no
quoting or brackets.

If the file exists and the contents are identical to that being
written, the file will not be updated. This will prevent unnecessary
rebuilds of targets that depend on this file.

One use for write_file is to write a list of inputs to an script
that might be too long for the command line. However, it is
preferrable to use response files for this purpose. See
"gn help response_file_contents".

TODO(brettw) we probably need an optional third argument to control
list formatting.
```

Arguments

```
filename
    Filename to write. This must be within the output directory.

data:
    The list or string to write.
```

current_cpu: The processor architecture of the current toolchain.

```
The build configuration usually sets this value based on the value
of "host_cpu" (see "gn help host_cpu") and then threads
this through the toolchain definitions to ensure that it always
reflects the appropriate value.

This value is not used internally by GN for any purpose. It is
set it to the empty string ("") by default but is declared so
that it can be overridden on the command line if so desired.

See "gn help target_cpu" for a list of common values returned.
```

current_os: The operating system of the current toolchain.

```
The build configuration usually sets this value based on the value
of "target_os" (see "gn help target_os"), and then threads this
through the toolchain definitions to ensure that it always reflects
the appropriate value.

This value is not used internally by GN for any purpose. It is
set it to the empty string ("") by default but is declared so
that it can be overridden on the command line if so desired.

See "gn help target_os" for a list of common values returned.
```

current_toolchain: Label of the current toolchain.

```
A fully-qualified label representing the current toolchain. You can
use this to make toolchain-related decisions in the build. See also
"default_toolchain".
```

Example

```
if (current_toolchain == "//build:64_bit_toolchain") {
  executable("output_thats_64_bit_only") {
    ...
```

default_toolchain: [string] Label of the default toolchain.

```
A fully-qualified label representing the default toolchain, which may
not necessarily be the current one (see "current_toolchain").
```

host_cpu: The processor architecture that GN is running on.

```
This is value is exposed so that cross-compile toolchains can
access the host architecture when needed.

The value should generally be considered read-only, but it can be
overriden in order to handle unusual cases where there might
be multiple plausible values for the host architecture (e.g., if
you can do either 32-bit or 64-bit builds). The value is not used
internally by GN for any purpose.
```

Some possible values:

```
- "x64"
- "x86"
```

host_os: [string] The operating system that GN is running on.

```
This value is exposed so that cross-compiles can access the host
build system's settings.

This value should generally be treated as read-only. It, however,
is not used internally by GN for any purpose.
```

Some possible values:

```
- "linux"
- "mac"
- "win"
```

python_path: Absolute path of Python.

```
Normally used in toolchain definitions if running some command
requires Python. You will normally not need this when invoking scripts
since GN automatically finds it for you.
```

root_build_dir: [string] Directory where build commands are run.

```
This is the root build output directory which will be the current
directory when executing all compilers and scripts.

Most often this is used with rebase_path (see "gn help rebase_path")
to convert arguments to be relative to a script's current directory.
```

root_gen_dir: Directory for the toolchain's generated files.

```
Absolute path to the root of the generated output directory tree for
the current toolchain. An example would be "//out/Debug/gen" for the
default toolchain, or "//out/Debug/arm/gen" for the "arm"
toolchain.

This is primarily useful for setting up include paths for generated
files. If you are passing this to a script, you will want to pass it
through rebase_path() (see "gn help rebase_path") to convert it
to be relative to the build directory.

See also "target_gen_dir" which is usually a better location for
generated files. It will be inside the root generated dir.
```

root_out_dir: [string] Root directory for toolchain output files.

```
Absolute path to the root of the output directory tree for the current
toolchain. It will not have a trailing slash.

For the default toolchain this will be the same as the root_build_dir.
An example would be "//out/Debug" for the default toolchain, or
"//out/Debug/arm" for the "arm" toolchain.

This is primarily useful for setting up script calls. If you are
passing this to a script, you will want to pass it through
rebase_path() (see "gn help rebase_path") to convert it
to be relative to the build directory.

See also "target_out_dir" which is usually a better location for
output files. It will be inside the root output dir.
```

Example

```
action("myscript") {
  # Pass the output dir to the script.
  args = [ "-o", rebase_path(root_out_dir, root_build_dir) ]
}
```

target_cpu: The desired cpu architecture for the build.

```
This value should be used to indicate the desired architecture for
the primary objects of the build. It will match the cpu architecture
of the default toolchain.

In many cases, this is the same as "host_cpu", but in the case
of cross-compiles, this can be set to something different. This
value is different from "current_cpu" in that it can be referenced
from inside any toolchain. This value can also be ignored if it is
not needed or meaningful for a project.

This value is not used internally by GN for any purpose, so it
may be set to whatever value is needed for the build.
```

```
GN defaults this value to the empty string ("") and the
configuration files should set it to an appropriate value
(e.g., setting it to the value of "host_cpu") if it is not
overridden on the command line or in the args.gn file.

Where practical, use one of the following list of common values:
```

Possible values:

```
- "x86"
- "x64"
- "arm"
- "arm64"
- "mipsel"
```

target_gen_dir: Directory for a target's generated files.

```
Absolute path to the target's generated file directory. This will be
the "root_gen_dir" followed by the relative path to the current
build file. If your file is in "//tools/doom_melon" then
target_gen_dir would be "//out/Debug/gen/tools/doom_melon". It will
not have a trailing slash.

This is primarily useful for setting up include paths for generated
files. If you are passing this to a script, you will want to pass it
through rebase_path() (see "gn help rebase_path") to convert it
to be relative to the build directory.

See also "gn help root_gen_dir".
```

Example

```
action("myscript") {
  # Pass the generated output dir to the script.
  args = [ "-o", rebase_path(target_gen_dir, root_build_dir) ]
}
```

target_os: The desired operating system for the build.

```
This value should be used to indicate the desired operating system
for the primary object(s) of the build. It will match the OS of
the default toolchain.

In many cases, this is the same as "host_os", but in the case of
cross-compiles, it may be different. This variable differs from
"current_os" in that it can be referenced from inside any
toolchain and will always return the initial value.

This should be set to the most specific value possible. So,
"android" or "chromeos" should be used instead of "linux"
where applicable, even though Android and ChromeOS are both Linux
variants. This can mean that one needs to write

    if (target_os == "android" || target_os == "linux") {
        # ...
    }

and so forth.

This value is not used internally by GN for any purpose, so it
may be set to whatever value is needed for the build.
GN defaults this value to the empty string ("") and the
configuration files should set it to an appropriate value
(e.g., setting it to the value of "host_os") if it is not
set via the command line or in the args.gn file.

Where practical, use one of the following list of common values:
```

Possible values:

```
- "android"
- "chromeos"
- "ios"
- "linux"
- "nacl"
- "mac"
- "win"
```

target_out_dir: [string] Directory for target output files.

```
Absolute path to the target's generated file directory. If your
current target is in "//tools/doom_melon" then this value might be
"//out/Debug/obj/tools/doom_melon". It will not have a trailing
slash.

This is primarily useful for setting up arguments for calling
scripts. If you are passing this to a script, you will want to pass it
through rebase_path() (see "gn help rebase_path") to convert it
to be relative to the build directory.

See also "gn help root_out_dir".
```

Example

```
action("myscript") {
  # Pass the output dir to the script.
  args = [ "-o", rebase_path(target_out_dir, root_build_dir) ]
}
```

all_dependent_configs: Configs to be forced on dependents.

```
A list of config labels.

All targets depending on this one, and recursively, all targets
depending on those, will have the configs listed in this variable
added to them. These configs will also apply to the current target.

This addition happens in a second phase once a target and all of its
dependencies have been resolved. Therefore, a target will not see
these force-added configs in their "configs" variable while the
script is running, and then can not be removed. As a result, this
capability should generally only be used to add defines and include
directories necessary to compile a target's headers.

See also "public_configs".
```

Ordering of flags and values

```
1. Those set on the current target (not in a config).
```

```
2. Those set on the "configs" on the target in order that the
   configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order
   that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that
   those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of
   the "deps" list. This is done recursively. If a config appears
   more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.
```

allow_circular_includes_from: Permit includes from deps.

```
A list of target labels. Must be a subset of the target's "deps".
These targets will be permitted to include headers from the current
target despite the dependency going in the opposite direction.

When you use this, both targets must be included in a final binary
for it to link. To keep linker errors from happening, it is good
practice to have all external dependencies depend only on one of
the two targets, and to set the visibility on the other to enforce
this. Thus the targets will always be linked together in any output.
```

Details

```
Normally, for a file in target A to include a file from target B,
A must list B as a dependency. This invariant is enforced by the
"gn check" command (and the --check flag to "gn gen" -- see
"gn help check").

Sometimes, two targets might be the same unit for linking purposes
(two source sets or static libraries that would always be linked
together in a final executable or shared library) and they each
include headers from the other: you want A to be able to include B's
headers, and B to include A's headers. This is not an ideal situation
but is sometimes unavoidable.

This list, if specified, lists which of the dependencies of the
current target can include header files from the current target.
That is, if A depends on B, B can only include headers from A if it is
```

```
in A's allow_circular_includes_from list. Normally includes must
follow the direction of dependencies, this flag allows them to go
in the opposite direction.
```

Danger

```
In the above example, A's headers are likely to include headers from
A's dependencies. Those dependencies may have public_configs that
apply flags, defines, and include paths that make those headers work
properly.

With allow_circular_includes_from, B can include A's headers, and
transitively from A's dependencies, without having the dependencies
that would bring in the public_configs those headers need. The result
may be errors or inconsistent builds.

So when you use allow_circular_includes_from, make sure that any
compiler settings, flags, and include directories are the same between
both targets (consider putting such things in a shared config they can
both reference). Make sure the dependencies are also the same (you
might consider a group to collect such dependencies they both
depend on).
```

Example

```
source_set("a") {
  deps = [ ":b", ":a_b_shared_deps" ]
  allow_circular_includes_from = [ ":b" ]
  ...
}

source_set("b") {
  deps = [ ":a_b_shared_deps" ]
  # Sources here can include headers from a despite lack of deps.
  ...
}

group("a_b_shared_deps") {
  public_deps = [ ":c" ]
}
```

arflags: Arguments passed to static_library archiver.

```
A list of flags passed to the archive/lib command that creates static
libraries.

arflags are NOT pushed to dependents, so applying arflags to source
sets or any other target type will be a no-op. As with ldflags,
you could put the arflags in a config and set that as a public or
"all dependent" config, but that will likely not be what you want.
If you have a chain of static libraries dependent on each other,
this can cause the flags to propagate up to other static libraries.
Due to the nature of how arflags are typically used, you will normally
want to apply them directly on static_library targets themselves.
```

Ordering of flags and values

1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the
   configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order
   that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that
   those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of
   the "deps" list. This is done recursively. If a config appears
   more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.

args: Arguments passed to an action.

```
For action and action_foreach targets, args is the list of arguments
to pass to the script. Typically you would use source expansion (see
"gn help source_expansion") to insert the source file names.

See also "gn help action" and "gn help action_foreach".
```

asmflags: Flags passed to the assembler.

```
A list of strings.

"asmflags" are passed to any invocation of a tool that takes an
.asm or .S file as input.
```

Ordering of flags and values

```
1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the
   configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order
   that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that
   those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of
   the "deps" list. This is done recursively. If a config appears
   more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.
```

assert_no_deps: Ensure no deps on these targets.

```
A list of label patterns.

This list is a list of patterns that must not match any of the
transitive dependencies of the target. These include all public,
private, and data dependencies, and cross shared library boundaries.
This allows you to express that undesirable code isn't accidentally
added to downstream dependencies in a way that might otherwise be
difficult to notice.

Checking does not cross executable boundaries. If a target depends on
an executable, it's assumed that the executable is a tool that is
producing part of the build rather than something that is linked and
distributed. This allows assert_no_deps to express what is distributed
in the final target rather than depend on the internal build steps
(which may include non-distributable code).
```

```
See "gn help label_pattern" for the format of the entries in the
list. These patterns allow blacklisting individual targets or whole
directory hierarchies.

Sometimes it is desirable to enforce that many targets have no
dependencies on a target or set of targets. One efficient way to
express this is to create a group with the assert_no_deps rule on
it, and make that group depend on all targets you want to apply that
assertion to.
```

Example

```
executable("doom_melon") {
  deps = [ "//foo:bar" ]
  ...
  assert_no_deps = [
    "//evil/*",  # Don't link any code from the evil directory.
    "//foo:test_support",  # This target is also disallowed.
  ]
}
```

bundle_executable_dir: Expansion of {{bundle_executable_dir}} in create_bundle.

```
A string corresponding to a path in $root_build_dir.

This string is used by the "create_bundle" target to expand the
{{bundle_executable_dir}} of the "bundle_data" target it depends on.
This must correspond to a path under "bundle_root_dir".

See "gn help bundle_root_dir" for examples.
```

bundle_plugins_dir: Expansion of {{bundle_plugins_dir}} in create_bundle.

```
A string corresponding to a path in $root_build_dir.
```

```
This string is used by the "create_bundle" target to expand the
{{bundle_plugins_dir}} of the "bundle_data" target it depends on.
This must correspond to a path under "bundle_root_dir".

See "gn help bundle_root_dir" for examples.
```

bundle_resources_dir: Expansion of {{bundle_resources_dir}} in create_bundle.

```
A string corresponding to a path in $root_build_dir.

This string is used by the "create_bundle" target to expand the
{{bundle_resources_dir}} of the "bundle_data" target it depends on.
This must correspond to a path under "bundle_root_dir".

See "gn help bundle_root_dir" for examples.
```

bundle_root_dir: Expansion of {{bundle_root_dir}} in create_bundle.

```
A string corresponding to a path in root_build_dir.

This string is used by the "create_bundle" target to expand the
{{bundle_root_dir}} of the "bundle_data" target it depends on.
This must correspond to a path under root_build_dir.
```

Example

```
bundle_data("info_plist") {
  sources = [ "Info.plist" ]
  outputs = [ "{{bundle_root_dir}}/Info.plist" ]
}

create_bundle("doom_melon.app") {
  deps = [ ":info_plist" ]
  bundle_root_dir = root_build_dir + "/doom_melon.app/Contents"
  bundle_resources_dir = bundle_root_dir + "/Resources"
```

```
  bundle_executable_dir = bundle_root_dir + "/MacOS"
  bundle_plugins_dir = bundle_root_dir + "/PlugIns"
}
```

cflags*: Flags passed to the C compiler.

```
A list of strings.

"cflags" are passed to all invocations of the C, C++, Objective C,
and Objective C++ compilers.

To target one of these variants individually, use "cflags_c",
"cflags_cc", "cflags_objc", and "cflags_objcc",
respectively. These variant-specific versions of cflags* will be
appended on the compiler command line after "cflags".

See also "asmflags" for flags for assembly-language files.
```

Ordering of flags and values

```
1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the
   configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order
   that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that
   those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of
   the "deps" list. This is done recursively. If a config appears
   more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.
```

cflags*: Flags passed to the C compiler.

```
A list of strings.
```

```
"cflags" are passed to all invocations of the C, C++, Objective C,
and Objective C++ compilers.

To target one of these variants individually, use "cflags_c",
"cflags_cc", "cflags_objc", and "cflags_objcc",
respectively. These variant-specific versions of cflags* will be
appended on the compiler command line after "cflags".

See also "asmflags" for flags for assembly-language files.
```

Ordering of flags and values

1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of the "deps" list. This is done recursively. If a config appears more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the "deps" list. If a dependency is public, they will be applied recursively.

cflags*: Flags passed to the C compiler.

```
A list of strings.

"cflags" are passed to all invocations of the C, C++, Objective C,
and Objective C++ compilers.

To target one of these variants individually, use "cflags_c",
"cflags_cc", "cflags_objc", and "cflags_objcc",
respectively. These variant-specific versions of cflags* will be
appended on the compiler command line after "cflags".

See also "asmflags" for flags for assembly-language files.
```

Ordering of flags and values

1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of the "deps" list. This is done recursively. If a config appears more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the "deps" list. If a dependency is public, they will be applied recursively.

cflags*: Flags passed to the C compiler.

A list of strings.

"cflags" are passed to all invocations of the C, C++, Objective C, and Objective C++ compilers.

To target one of these variants individually, use "cflags_c", "cflags_cc", "cflags_objc", and "cflags_objcc", respectively. These variant-specific versions of cflags* will be appended on the compiler command line after "cflags".

See also "asmflags" for flags for assembly-language files.

Ordering of flags and values

1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of the "deps" list. This is done recursively. If a config appears more than once, only the first occurance will be used.

```
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.
```

cflags*: Flags passed to the C compiler.

```
A list of strings.

"cflags" are passed to all invocations of the C, C++, Objective C,
and Objective C++ compilers.

To target one of these variants individually, use "cflags_c",
"cflags_cc", "cflags_objc", and "cflags_objcc",
respectively. These variant-specific versions of cflags* will be
appended on the compiler command line after "cflags".

See also "asmflags" for flags for assembly-language files.
```

Ordering of flags and values

```
1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the
   configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order
   that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that
   those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of
   the "deps" list. This is done recursively. If a config appears
   more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.
```

check_includes: [boolean] Controls whether a target's files are checked.

```
When true (the default), the "gn check" command (as well as
```

```
"gn gen" with the --check flag) will check this target's sources
and headers for proper dependencies.

When false, the files in this target will be skipped by default.
This does not affect other targets that depend on the current target,
it just skips checking the includes of the current target's files.

If there are a few conditionally included headers that trip up
checking, you can exclude headers individually by annotating them with
"nogncheck" (see "gn help nogncheck").

The topic "gn help check" has general information on how checking
works and advice on how to pass a check in problematic cases.
```

Example

```
source_set("busted_includes") {
  # This target's includes are messed up, exclude it from checking.
  check_includes = false
  ...
}
```

complete_static_lib: [boolean] Links all deps into a static library.

```
A static library normally doesn't include code from dependencies, but
instead forwards the static libraries and source sets in its deps up
the dependency chain until a linkable target (an executable or shared
library) is reached. The final linkable target only links each static
library once, even if it appears more than once in its dependency
graph.

In some cases the static library might be the final desired output.
For example, you may be producing a static library for distribution to
third parties. In this case, the static library should include code
for all dependencies in one complete package. However, complete static
libraries themselves are never linked into other complete static
libraries. All complete static libraries are for distribution and
linking them in would cause code duplication in this case. If the
static library is not for distribution, it should not be complete.

GN treats non-complete static libraries as source sets when they are
```

```
linked into complete static libraries. This is done because some tools
like AR do not handle dependent static libraries properly. This makes
it easier to write "alink" rules.

In rare cases it makes sense to list a header in more than one
target if it could be considered conceptually a member of both.
libraries.
```

Example

```
static_library("foo") {
  complete_static_lib = true
  deps = [ "bar" ]
}
```

configs: Configs applying to this target or config.

```
A list of config labels.
```

Configs on a target

```
When used on a target, the include_dirs, defines, etc. in each config
are appended in the order they appear to the compile command for each
file in the target. They will appear after the include_dirs, defines,
etc. that the target sets directly.

Since configs apply after the values set on a target, directly setting
a compiler flag will prepend it to the command line. If you want to
append a flag instead, you can put that flag in a one-off config and
append that config to the target's configs list.

The build configuration script will generally set up the default
configs applying to a given target type (see "set_defaults").
When a target is being defined, it can add to or remove from this
list.
```

Configs on a config

```
It is possible to create composite configs by specifying configs on a
config. One might do this to forward values, or to factor out blocks
of settings from very large configs into more manageable named chunks.

In this case, the composite config is expanded to be the concatenation
of its own values, and in order, the values from its sub-configs
*before* anything else happens. This has some ramifications:

 - A target has no visibility into a config's sub-configs. Target
   code only sees the name of the composite config. It can't remove
   sub-configs or opt in to only parts of it. The composite config may
   not even be defined before the target is.

 - You can get duplication of values if a config is listed twice, say,
   on a target and in a sub-config that also applies. In other cases,
   the configs applying to a target are de-duped. It's expected that
   if a config is listed as a sub-config that it is only used in that
   context. (Note that it's possible to fix this and de-dupe, but it's
   not normally relevant and complicates the implementation.)
```

Ordering of flags and values

```
1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the
   configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order
   that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that
   those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of
   the "deps" list. This is done recursively. If a config appears
   more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.
```

Example

```
# Configs on a target.
source_set("foo") {
  # Don't use the default RTTI config that BUILDCONFIG applied to us.
  configs -= [ "//build:no_rtti" ]
```

```
  # Add some of our own settings.
  configs += [ ":mysettings" ]
}

# Create a default_optimization config that forwards to one of a set
# of more specialized configs depending on build flags. This pattern
# is useful because it allows a target to opt in to either a default
# set, or a more specific set, while avoid duplicating the settings in
# two places.
config("super_optimization") {
  cflags = [ ... ]
}
config("default_optimization") {
  if (optimize_everything) {
    configs = [ ":super_optimization" ]
  } else {
    configs = [ ":no_optimization" ]
  }
}
```

console: Run this action in the console pool.

```
Boolean. Defaults to false.

Actions marked "console = true" will be run in the built-in ninja
"console" pool. They will have access to real stdin and stdout, and
output will not be buffered by ninja. This can be useful for
long-running actions with progress logs, or actions that require user
input.

Only one console pool target can run at any one time in Ninja. Refer
to the Ninja documentation on the console pool for more info.
```

Example

```
action("long_action_with_progress_logs") {
  console = true
}
```

data: Runtime data file dependencies.

```
Lists files or directories required to run the given target. These are
typically data files or directories of data files. The paths are
interpreted as being relative to the current build file. Since these
are runtime dependencies, they do not affect which targets are built
or when. To declare input files to a script, use "inputs".

Appearing in the "data" section does not imply any special handling
such as copying them to the output directory. This is just used for
declaring runtime dependencies. Runtime dependencies can be queried
using the "runtime_deps" category of "gn desc" or written during
build generation via "--runtime-deps-list-file".

GN doesn't require data files to exist at build-time. So actions that
produce files that are in turn runtime dependencies can list those
generated files both in the "outputs" list as well as the "data"
list.

By convention, directories are listed with a trailing slash:
  data = [ "test/data/" ]
However, no verification is done on these so GN doesn't enforce this.
The paths are just rebased and passed along when requested.

See "gn help runtime_deps" for how these are used.
```

data_deps: Non-linked dependencies.

```
A list of target labels.

Specifies dependencies of a target that are not actually linked into
the current target. Such dependencies will be built and will be
available at runtime.

This is normally used for things like plugins or helper programs that
a target needs at runtime.

See also "gn help deps" and "gn help data".
```

Example

```
executable("foo") {
  deps = [ "//base" ]
  data_deps = [ "//plugins:my_runtime_plugin" ]
}
```

defines: C preprocessor defines.

```
A list of strings

These strings will be passed to the C/C++ compiler as #defines. The
strings may or may not include an "=" to assign a value.
```

Ordering of flags and values

```
1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the
   configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order
   that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that
   those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of
   the "deps" list. This is done recursively. If a config appears
   more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.
```

Example

```
defines = [ "AWESOME_FEATURE", "LOG_LEVEL=3" ]
```

depfile: [string] File name for input dependencies for actions.

```
If nonempty, this string specifies that the current action or
```

```
action_foreach target will generate the given ".d" file containing
the dependencies of the input. Empty or unset means that the script
doesn't generate the files.

The .d file should go in the target output directory. If you have more
than one source file that the script is being run over, you can use
the output file expansions described in "gn help action_foreach" to
name the .d file according to the input.
The format is that of a Makefile, and all of the paths should be
relative to the root build directory.
```

Example

```
action_foreach("myscript_target") {
  script = "myscript.py"
  sources = [ ... ]

  # Locate the depfile in the output directory named like the
  # inputs but with a ".d" appended.
  depfile = "$relative_target_output_dir/{{source_name}}.d"

  # Say our script uses "-o <d file>" to indicate the depfile.
  args = [ "{{source}}", "-o", depfile ]
}
```

deps: Private linked dependencies.

```
A list of target labels.

Specifies private dependencies of a target. Private dependencies are
propagated up the dependency tree and linked to dependant targets, but
do not grant the ability to include headers from the dependency.
Public configs are not forwarded.
```

Details of dependency propagation

```
Source sets, shared libraries, and non-complete static libraries
will be propagated up the dependency tree across groups, non-complete
static libraries and source sets.
```

```
Executables, shared libraries, and complete static libraries will
link all propagated targets and stop propagation. Actions and copy
steps also stop propagation, allowing them to take a library as an
input but not force dependants to link to it.

Propagation of all_dependent_configs and public_configs happens
independently of target type. all_dependent_configs are always
propagated across all types of targets, and public_configs
are always propagated across public deps of all types of targets.

Data dependencies are propagated differently. See
"gn help data_deps" and "gn help runtime_deps".

See also "public_deps".
```

include_dirs: Additional include directories.

```
A list of source directories.

The directories in this list will be added to the include path for
the files in the affected target.
```

Ordering of flags and values

```
1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the
   configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order
   that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that
   those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of
   the "deps" list. This is done recursively. If a config appears
   more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.
```

Example

```
include_dirs = [ "src/include", "//third_party/foo" ]
```

inputs: Additional compile-time dependencies.

```
Inputs are compile-time dependencies of the current target. This means
that all inputs must be available before compiling any of the sources
or executing any actions.

Inputs are typically only used for action and action_foreach targets.
```

Inputs for actions

```
For action and action_foreach targets, inputs should be the inputs to
script that don't vary. These should be all .py files that the script
uses via imports (the main script itself will be an implicit dependency
of the action so need not be listed).

For action targets, inputs and sources are treated the same, but from
a style perspective, it's recommended to follow the same rule as
action_foreach and put helper files in the inputs, and the data used
by the script (if any) in sources.

Note that another way to declare input dependencies from an action
is to have the action write a depfile (see "gn help depfile"). This
allows the script to dynamically write input dependencies, that might
not be known until actually executing the script. This is more
efficient than doing processing while running GN to determine the
inputs, and is easier to keep in-sync than hardcoding the list.
```

Script input gotchas

```
It may be tempting to write a script that enumerates all files in a
directory as inputs. Don't do this! Even if you specify all the files
in the inputs or sources in the GN target (or worse, enumerate the
files in an exec_script call when running GN, which will be slow), the
dependencies will be broken.

The problem happens if a file is ever removed because the inputs are
not listed on the command line to the script. Because the script
```

```
hasn't changed and all inputs are up-to-date, the script will not
re-run and you will get a stale build. Instead, either list all
inputs on the command line to the script, or if there are many, create
a separate list file that the script reads. As long as this file is
listed in the inputs, the build will detect when it has changed in any
way and the action will re-run.
```

Inputs for binary targets

```
Any input dependencies will be resolved before compiling any sources.
Normally, all actions that a target depends on will be run before any
files in a target are compiled. So if you depend on generated headers,
you do not typically need to list them in the inputs section.

Inputs for binary targets will be treated as order-only dependencies,
meaning that they will be forced up-to-date before compiling or
any files in the target, but changes in the inputs will not
necessarily force the target to compile. This is because it is
expected that the compiler will report the precise list of input
dependencies required to recompile each file once the initial build
is done.
```

Example

```
action("myscript") {
  script = "domything.py"
  inputs = [ "input.data" ]
}
```

ldflags: Flags passed to the linker.

```
A list of strings.

These flags are passed on the command-line to the linker and generally
specify various linking options. Most targets will not need these and
will use "libs" and "lib_dirs" instead.

ldflags are NOT pushed to dependents, so applying ldflags to source
sets or static libraries will be a no-op. If you want to apply ldflags
to dependent targets, put them in a config and set it in the
```

```
all_dependent_configs or public_configs.
```

Ordering of flags and values

```
1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the
   configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order
   that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that
   those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of
   the "deps" list. This is done recursively. If a config appears
   more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.
```

lib_dirs: Additional library directories.

```
A list of directories.

Specifies additional directories passed to the linker for searching
for the required libraries. If an item is not an absolute path, it
will be treated as being relative to the current build file.

libs and lib_dirs work differently than other flags in two respects.
First, then are inherited across static library boundaries until a
shared library or executable target is reached. Second, they are
uniquified so each one is only passed once (the first instance of it
will be the one used).
```

Ordering of flags and values

```
1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the
   configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order
   that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that
```

```
   those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of
   the "deps" list. This is done recursively. If a config appears
   more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.

For "libs" and "lib_dirs" only, the values propagated from
dependencies (as described above) are applied last assuming they
are not already in the list.
```

Example

```
lib_dirs = [ "/usr/lib/foo", "lib/doom_melon" ]
```

libs: Additional libraries to link.

```
A list of library names or library paths.

These libraries will be linked into the final binary (executable or
shared library) containing the current target.

libs and lib_dirs work differently than other flags in two respects.
First, then are inherited across static library boundaries until a
shared library or executable target is reached. Second, they are
uniquified so each one is only passed once (the first instance of it
will be the one used).
```

Types of libs

```
There are several different things that can be expressed in libs:

File paths
    Values containing '/' will be treated as references to files in
    the checkout. They will be rebased to be relative to the build
    directory and specified in the "libs" for linker tools. This
    facility should be used for libraries that are checked in to the
    version control. For libraries that are generated by the build,
    use normal GN deps to link them.
```

```
System libraries
    Values not containing '/' will be treated as system library names.
    These will be passed unmodified to the linker and prefixed with
    the "lib_prefix" attribute of the linker tool. Generally you
    would set the "lib_dirs" so the given library is found. Your
    BUILD.gn file should not specify the switch (like "-l"): this
    will be encoded in the "lib_prefix" of the tool.

Apple frameworks
    System libraries ending in ".framework" will be special-cased:
    the switch "-framework" will be prepended instead of the
    lib_prefix, and the ".framework" suffix will be trimmed. This is
    to support the way Mac links framework dependencies.
```

Ordering of flags and values

```
1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the
   configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order
   that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that
   those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of
   the "deps" list. This is done recursively. If a config appears
   more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the
   "deps" list. If a dependency is public, they will be applied
   recursively.

For "libs" and "lib_dirs" only, the values propagated from
dependencies (as described above) are applied last assuming they
are not already in the list.
```

Examples

```
On Windows:
  libs = [ "ctl3d.lib" ]

On Linux:
  libs = [ "ld" ]
```

output_dir: [directory] Directory to put output file in.

```
For library and executable targets, overrides the directory for the
final output. This must be in the root_build_dir or a child thereof.

This should generally be in the root_out_dir or a subdirectory thereof
(the root_out_dir will be the same as the root_build_dir for the
default toolchain, and will be a subdirectory for other toolchains).
Not putting the output in a subdirectory of root_out_dir can result
in collisions between different toolchains, so you will need to take
steps to ensure that your target is only present in one toolchain.

Normally the toolchain specifies the output directory for libraries
and executables (see "gn help tool"). You will have to consult that
for the default location. The default location will be used if
output_dir is undefined or empty.
```

Example

```
shared_library("doom_melon") {
  output_dir = "$root_out_dir/plugin_libs"
  ...
}
```

output_extension: Value to use for the output's file extension.

```
Normally the file extension for a target is based on the target
type and the operating system, but in rare cases you will need to
override the name (for example to use "libfreetype.so.6" instead
of libfreetype.so on Linux).

This value should not include a leading dot. If undefined, the default
specified on the tool will be used. If set to the empty string, no
output extension will be used.

The output_extension will be used to set the "{{output_extension}}"
expansion which the linker tool will generally use to specify the
output file name. See "gn help tool".
```

Example

```
shared_library("freetype") {
  if (is_linux) {
    # Call the output "libfreetype.so.6"
    output_extension = "so.6"
  }
  ...
}

# On Windows, generate a "mysettings.cpl" control panel applet.
# Control panel applets are actually special shared libraries.
if (is_win) {
  shared_library("mysettings") {
    output_extension = "cpl"
    ...
  }
}
```

output_name: Define a name for the output file other than the default.

```
Normally the output name of a target will be based on the target name,
so the target "//foo/bar:bar_unittests" will generate an output
file such as "bar_unittests.exe" (using Windows as an example).

Sometimes you will want an alternate name to avoid collisions or
if the internal name isn't appropriate for public distribution.

The output name should have no extension or prefixes, these will be
added using the default system rules. For example, on Linux an output
name of "foo" will produce a shared library "libfoo.so". There
is no way to override the output prefix of a linker tool on a per-
target basis. If you need more flexibility, create a copy target
to produce the file you want.

This variable is valid for all binary output target types.
```

Example

```
static_library("doom_melon") {
```

```
  output_name = "fluffy_bunny"
}
```

output_prefix_override: Don’t use prefix for output name.

```
A boolean that overrides the output prefix for a target. Defaults to
false.

Some systems use prefixes for the names of the final target output
file. The normal example is "libfoo.so" on Linux for a target
named "foo".

The output prefix for a given target type is specified on the linker
tool (see "gn help tool"). Sometimes this prefix is undesired.

See also "gn help output_extension".
```

Example

```
shared_library("doom_melon") {
  # Normally this will produce "libdoom_melon.so" on Linux, setting
  # Setting this flag will produce "doom_melon.so".
  output_prefix_override = true
  ...
}
```

outputs: Output files for actions and copy targets.

```
Outputs is valid for "copy", "action", and "action_foreach"
target types and indicates the resulting files. Outputs must always
refer to files in the build directory.

copy
  Copy targets should have exactly one entry in the outputs list. If
  there is exactly one source, this can be a literal file name or a
  source expansion. If there is more than one source, this must
  contain a source expansion to map a single input name to a single
  output name. See "gn help copy".
```

```
action_foreach
  Action_foreach targets must always use source expansions to map
  input files to output files. There can be more than one output,
  which means that each invocation of the script will produce a set of
  files (presumably based on the name of the input file). See
  "gn help action_foreach".

action
  Action targets (excluding action_foreach) must list literal output
  file(s) with no source expansions. See "gn help action".
```

precompiled_header: [string] Header file to precompile.

```
Precompiled headers will be used when a target specifies this
value, or a config applying to this target specifies this value.
In addition, the tool corresponding to the source files must also
specify precompiled headers (see "gn help tool"). The tool
will also specify what type of precompiled headers to use.

The precompiled header/source variables can be specified on a target
or a config, but must be the same for all configs applying to a given
target since a target can only have one precompiled header.
```

MSVC precompiled headers

```
When using MSVC-style precompiled headers, the "precompiled_header"
value is a string corresponding to the header. This is NOT a path
to a file that GN recognises, but rather the exact string that appears
in quotes after an #include line in source code. The compiler will
match this string against includes or forced includes (/FI).

MSVC also requires a source file to compile the header with. This must
be specified by the "precompiled_source" value. In contrast to the
header value, this IS a GN-style file name, and tells GN which source
file to compile to make the .pch file used for subsequent compiles.

If you use both C and C++ sources, the precompiled header and source
file will be compiled using both tools. You will want to make sure
to wrap C++ includes in __cplusplus #ifdefs so the file will compile
in C mode.
```

```
For example, if the toolchain specifies MSVC headers:

  toolchain("vc_x64") {
    ...
    tool("cxx") {
      precompiled_header_type = "msvc"
      ...

You might make a config like this:

  config("use_precompiled_headers") {
    precompiled_header = "build/precompile.h"
    precompiled_source = "//build/precompile.cc"

    # Either your source files should #include "build/precompile.h"
    # first, or you can do this to force-include the header.
    cflags = [ "/FI$precompiled_header" ]
  }

And then define a target that uses the config:

  executable("doom_melon") {
    configs += [ ":use_precompiled_headers" ]
    ...
```

precompiled_source: [file name] Source file to precompile.

```
The source file that goes along with the precompiled_header when
using "msvc"-style precompiled headers. It will be implicitly added
to the sources of the target. See "gn help precompiled_header".
```

product_type: Product type for Xcode projects.

```
Correspond to the type of the product of a create_bundle target. Only
meaningful to Xcode (used as part of the Xcode project generation).

When generating Xcode project files, only create_bundle target with
a non-empty product_type will have a corresponding target in Xcode
project.
```

public: Declare public header files for a target.

```
A list of files that other targets can include. These permissions are
checked via the "check" command (see "gn help check").

If no public files are declared, other targets (assuming they have
visibility to depend on this target can include any file in the
sources list. If this variable is defined on a target, dependent
targets may only include files on this whitelist.

Header file permissions are also subject to visibility. A target
must be visible to another target to include any files from it at all
and the public headers indicate which subset of those files are
permitted. See "gn help visibility" for more.

Public files are inherited through the dependency tree. So if there is
a dependency A -> B -> C, then A can include C's public headers.
However, the same is NOT true of visibility, so unless A is in C's
visibility list, the include will be rejected.

GN only knows about files declared in the "sources" and "public"
sections of targets. If a file is included that is not known to the
build, it will be allowed.
```

Examples

```
These exact files are public:
  public = [ "foo.h", "bar.h" ]

No files are public (no targets may include headers from this one):
  public = []
```

public_configs: Configs to be applied on dependents.

```
A list of config labels.

Targets directly depending on this one will have the configs listed in
this variable added to them. These configs will also apply to the
```

```
current target.

This addition happens in a second phase once a target and all of its
dependencies have been resolved. Therefore, a target will not see
these force-added configs in their "configs" variable while the
script is running, and then can not be removed. As a result, this
capability should generally only be used to add defines and include
directories necessary to compile a target's headers.

See also "all_dependent_configs".
```

Ordering of flags and values

1. Those set on the current target (not in a config).
2. Those set on the "configs" on the target in order that the configs appear in the list.
3. Those set on the "all_dependent_configs" on the target in order that the configs appear in the list.
4. Those set on the "public_configs" on the target in order that those configs appear in the list.
5. all_dependent_configs pulled from dependencies, in the order of the "deps" list. This is done recursively. If a config appears more than once, only the first occurance will be used.
6. public_configs pulled from dependencies, in the order of the "deps" list. If a dependency is public, they will be applied recursively.

public_deps: Declare public dependencies.

```
Public dependencies are like private dependencies (see
"gn help deps") but additionally express that the current target
exposes the listed deps as part of its public API.

This has several ramifications:

  - public_configs that are part of the dependency are forwarded
    to direct dependents.

  - Public headers in the dependency are usable by dependents
    (includes do not require a direct dependency or visibility).

  - If the current target is a shared library, other shared libraries
```

```
    that it publicly depends on (directly or indirectly) are
    propagated up the dependency tree to dependents for linking.
```

Discussion

```
Say you have three targets: A -> B -> C. C's visibility may allow
B to depend on it but not A. Normally, this would prevent A from
including any headers from C, and C's public_configs would apply
only to B.

If B lists C in its public_deps instead of regular deps, A will now
inherit C's public_configs and the ability to include C's public
headers.

Generally if you are writing a target B and you include C's headers
as part of B's public headers, or targets depending on B should
consider B and C to be part of a unit, you should use public_deps
instead of deps.
```

Example

```
# This target can include files from "c" but not from
# "super_secret_implementation_details".
executable("a") {
  deps = [ ":b" ]
}

shared_library("b") {
  deps = [ ":super_secret_implementation_details" ]
  public_deps = [ ":c" ]
}
```

response_file_contents: Contents of a response file for actions.

```
Sometimes the arguments passed to a script can be too long for the
system's command-line capabilities. This is especially the case on
Windows where the maximum command-line length is less than 8K. A
response file allows you to pass an unlimited amount of data to a
script in a temporary file for an action or action_foreach target.
```

```
If the response_file_contents variable is defined and non-empty, the
list will be treated as script args (including possibly substitution
patterns) that will be written to a temporary file at build time.
The name of the temporary file will be substituted for
"{{response_file_name}}" in the script args.

The response file contents will always be quoted and escaped
according to Unix shell rules. To parse the response file, the Python
script should use "shlex.split(file_contents)".
```

Example

```
action("process_lots_of_files") {
  script = "process.py",
  inputs = [ ... huge list of files ... ]

  # Write all the inputs to a response file for the script. Also,
  # make the paths relative to the script working directory.
  response_file_contents = rebase_path(inputs, root_build_dir)

  # The script expects the name of the response file in --file-list.
  args = [
    "--enable-foo",
    "--file-list={{response_file_name}}",
  ]
}
```

script: Script file for actions.

```
An absolute or buildfile-relative file name of a Python script to run
for a action and action_foreach targets (see "gn help action" and
"gn help action_foreach").
```

sources: Source files for a target

```
A list of files. Non-absolute paths will be resolved relative to the
current build file.
```

Sources for binary targets

```
For binary targets (source sets, executables, and libraries), the
known file types will be compiled with the associated tools. Unknown
file types and headers will be skipped. However, you should still
list all C/C+ header files so GN knows about the existance of those
files for the purposes of include checking.

As a special case, a file ending in ".def" will be treated as a
Windows module definition file. It will be appended to the link
line with a preceeding "/DEF:" string. There must be at most one
.def file in a target and they do not cross dependency boundaries
(so specifying a .def file in a static library or source set will have
no effect on the executable or shared library they're linked into).
```

Sources for non-binary targets

```
action_foreach
  The sources are the set of files that the script will be executed
  over. The script will run once per file.

action
  The sources will be treated the same as inputs. See "gn help inputs"
  for more information and usage advice.

copy
  The source are the source files to copy.
```

testonly: Declares a target must only be used for testing.

```
Boolean. Defaults to false.

When a target is marked "testonly = true", it must only be depended
on by other test-only targets. Otherwise, GN will issue an error
that the depenedency is not allowed.

This feature is intended to prevent accidentally shipping test code
in a final product.
```

Example

```
source_set("test_support") {
  testonly = true
  ...
}
```

visibility: A list of labels that can depend on a target.

```
A list of labels and label patterns that define which targets can
depend on the current one. These permissions are checked via the
"check" command (see "gn help check").

If visibility is not defined, it defaults to public ("*").

If visibility is defined, only the targets with labels that match it
can depend on the current target. The empty list means no targets
can depend on the current target.

Tip: Often you will want the same visibility for all targets in a
BUILD file. In this case you can just put the definition at the top,
outside of any target, and the targets will inherit that scope and see
the definition.
```

Patterns

```
See "gn help label_pattern" for more details on what types of
patterns are supported. If a toolchain is specified, only targets
in that toolchain will be matched. If a toolchain is not specified on
a pattern, targets in all toolchains will be matched.
```

Examples

```
Only targets in the current buildfile ("private"):
  visibility = [ ":*" ]

No targets (used for targets that should be leaf nodes):
  visibility = []
```

```
Any target ("public", the default):
  visibility = [ "*" ]

All targets in the current directory and any subdirectory:
  visibility = [ "./*" ]

Any target in "//bar/BUILD.gn":
  visibility = [ "//bar:*" ]

Any target in "//bar/" or any subdirectory thereof:
  visibility = [ "//bar/*" ]

Just these specific targets:
  visibility = [ ":mything", "//foo:something_else" ]

Any target in the current directory and any subdirectory thereof, plus
any targets in "//bar/" and any subdirectory thereof.
  visibility = [ "./*", "//bar/*" ]
```

write_runtime_deps: Writes the target's runtime_deps to the given path.

```
Does not synchronously write the file, but rather schedules it
to be written at the end of generation.

If the file exists and the contents are identical to that being
written, the file will not be updated. This will prevent unnecessary
rebuilds of targets that depend on this file.

Path must be within the output directory.

See "gn help runtime_deps" for how the runtime dependencies are
computed.

The format of this file will list one file per line with no escaping.
The files will be relative to the root_build_dir. The first line of
the file will be the main output file of the target itself. The file
contents will be the same as requesting the runtime deps be written on
the command line (see "gn help --runtime-deps-list-file").
```

Build Arguments Overview

```
Build arguments are variables passed in from outside of the build
that build files can query to determine how the build works.
```

How build arguments are set

```
First, system default arguments are set based on the current system.
The built-in arguments are:
 - host_cpu
 - host_os
 - current_cpu
 - current_os
 - target_cpu
 - target_os

If specified, arguments from the --args command line flag are used. If
that flag is not specified, args from previous builds in the build
directory will be used (this is in the file args.gn in the build
directory).

Last, for targets being compiled with a non-default toolchain, the
toolchain overrides are applied. These are specified in the
toolchain_args section of a toolchain definition. The use-case for
this is that a toolchain may be building code for a different
platform, and that it may want to always specify Posix, for example.
See "gn help toolchain_args" for more.

If you specify an override for a build argument that never appears in
a "declare_args" call, a nonfatal error will be displayed.
```

Examples

```
gn args out/FooBar
    Create the directory out/FooBar and open an editor. You would type
    something like this into that file:
        enable_doom_melon=false
        os="android"

gn gen out/FooBar --args="enable_doom_melon=true os=\"android\""
    This will overwrite the build directory with the given arguments.
    (Note that the quotes inside the args command will usually need to
    be escaped for your shell to pass through strings values.)
```

How build arguments are used

```
If you want to use an argument, you use declare_args() and specify
default values. These default values will apply if none of the steps
listed in the "How build arguments are set" section above apply to
the given argument, but the defaults will not override any of these.

Often, the root build config file will declare global arguments that
will be passed to all buildfiles. Individual build files can also
specify arguments that apply only to those files. It is also useful
to specify build args in an "import"-ed file if you want such
arguments to apply to multiple buildfiles.
```

.gn file

```
When gn starts, it will search the current directory and parent
directories for a file called ".gn". This indicates the source root.
You can override this detection by using the --root command-line
argument

The .gn file in the source root will be executed. The syntax is the
same as a buildfile, but with very limited build setup-specific
meaning.

If you specify --root, by default GN will look for the file .gn in
that directory. If you want to specify a different file, you can
additionally pass --dotfile:

  gn gen out/Debug --root=/home/build --dotfile=/home/my_gn_file.gn
```

Variables

```
buildconfig [required]
    Label of the build config file. This file will be used to set up
    the build file execution environment for each toolchain.

check_targets [optional]
    A list of labels and label patterns that should be checked when
    running "gn check" or "gn gen --check". If unspecified, all
    targets will be checked. If it is the empty list, no targets will
    be checked.
```

```
    The format of this list is identical to that of "visibility"
    so see "gn help visibility" for examples.

exec_script_whitelist [optional]
    A list of .gn/.gni files (not labels) that have permission to call
    the exec_script function. If this list is defined, calls to
    exec_script will be checked against this list and GN will fail if
    the current file isn't in the list.

    This is to allow the use of exec_script to be restricted since
    is easy to use inappropriately. Wildcards are not supported.
    Files in the secondary_source tree (if defined) should be
    referenced by ignoring the secondary tree and naming them as if
    they are in the main tree.

    If unspecified, the ability to call exec_script is unrestricted.

    Example:
      exec_script_whitelist = [
        "//base/BUILD.gn",
        "//build/my_config.gni",
      ]

root [optional]
    Label of the root build target. The GN build will start by loading
    the build file containing this target name. This defaults to
    "//:" which will cause the file //BUILD.gn to be loaded.

secondary_source [optional]
    Label of an alternate directory tree to find input files. When
    searching for a BUILD.gn file (or the build config file discussed
    above), the file will first be looked for in the source root.
    If it's not found, the secondary source root will be checked
    (which would contain a parallel directory hierarchy).

    This behavior is intended to be used when BUILD.gn files can't be
    checked in to certain source directories for whatever reason.

    The secondary source root must be inside the main source tree.
```

Example .gn file contents

```
buildconfig = "//build/config/BUILDCONFIG.gn"

check_targets = [
  "//doom_melon/*",  # Check everything in this subtree.
```

```
  "//tools:mind_controlling_ant",  # Check this specific target.
]

root = "//:root"

secondary_source = "//build/config/temporary_buildfiles/"
```

GN build language grammar

Tokens

```
GN build files are read as sequences of tokens.  While splitting the
file into tokens, the next token is the longest sequence of characters
that form a valid token.
```

White space and comments

```
White space is comprised of spaces (U+0020), horizontal tabs (U+0009),
carriage returns (U+000D), and newlines (U+000A).

Comments start at the character "#" and stop at the next newline.

White space and comments are ignored except that they may separate
tokens that would otherwise combine into a single token.
```

Identifiers

```
Identifiers name variables and functions.

    identifier = letter { letter | digit } .
    letter     = "A" ... "Z" | "a" ... "z" | "_" .
    digit      = "0" ... "9" .
```

Keywords

```
The following keywords are reserved and may not be used as
identifiers:
```

```
        else    false   if      true
```

Integer literals

```
An integer literal represents a decimal integer value.

    integer = [ "-" ] digit { digit } .

Leading zeros and negative zero are disallowed.
```

String literals

```
A string literal represents a string value consisting of the quoted
characters with possible escape sequences and variable expansions.

    string           = `"` { char | escape | expansion } `"` .
    escape           = `\` ( "$" | `"` | char ) .
    BracketExpansion = "{" ( identifier | ArrayAccess | ScopeAccess ) "}" .
    Hex              = "0x" [0-9A-Fa-f][0-9A-Fa-f]
    expansion        = "$" ( identifier | BracketExpansion | Hex ) .
    char             = /* any character except "$", `"`, or newline */ .

After a backslash, certain sequences represent special characters:

        \"    U+0022    quotation mark
        \$    U+0024    dollar sign
        \\    U+005C    backslash

All other backslashes represent themselves.

To insert an arbitrary byte value, use $0xFF. For example, to
insert a newline character: "Line one$0x0ALine two".
```

Punctuation

```
The following character sequences represent punctuation:

        +       +=      ==      !=      (       )
        -       -=      <       <=      [       ]
```

```
          !       =       >       >=      {       }
                          &&      ||      .       ,
```

Grammar

```
The input tokens form a syntax tree following a context-free grammar:

    File = StatementList .

    Statement     = Assignment | Call | Condition .
    Assignment    = identifier AssignOp Expr .
    Call          = identifier "(" [ ExprList ] ")" [ Block ] .
    Condition     = "if" "(" Expr ")" Block
                    [ "else" ( Condition | Block ) ] .
    Block         = "{" StatementList "}" .
    StatementList = { Statement } .

    ArrayAccess = identifier "[" { identifier | integer } "]" .
    ScopeAccess = identifier "." identifier .
    Expr        = UnaryExpr | Expr BinaryOp Expr .
    UnaryExpr   = PrimaryExpr | UnaryOp UnaryExpr .
    PrimaryExpr = identifier | integer | string | Call
                | ArrayAccess | ScopeAccess
                | "(" Expr ")"
                | "[" [ ExprList [ "," ] ] "]" .
    ExprList    = Expr { "," Expr } .

    AssignOp = "=" | "+=" | "-=" .
    UnaryOp  = "!" .
    BinaryOp = "+" | "-"                  // highest priority
             | "<" | "<=" | ">" | ">="
             | "==" | "!="
             | "&&"
             | "||" .                     // lowest priority

All binary operators are left-associative.
```

input_conversion: Specifies how to transform input to a variable.

```
input_conversion is an argument to read_file and exec_script that
specifies how the result of the read operation should be converted
```

```
into a variable.

"" (the default)
    Discard the result and return None.

"list lines"
    Return the file contents as a list, with a string for each line.
    The newlines will not be present in the result. The last line may
    or may not end in a newline.

    After splitting, each individual line will be trimmed of
    whitespace on both ends.

"scope"
    Execute the block as GN code and return a scope with the
    resulting values in it. If the input was:
      a = [ "hello.cc", "world.cc" ]
      b = 26
    and you read the result into a variable named "val", then you
    could access contents the "." operator on "val":
      sources = val.a
      some_count = val.b

"string"
    Return the file contents into a single string.

"value"
    Parse the input as if it was a literal rvalue in a buildfile.
    Examples of typical program output using this mode:
      [ "foo", "bar" ]     (result will be a list)
    or
      "foo bar"            (result will be a string)
    or
      5                    (result will be an integer)

    Note that if the input is empty, the result will be a null value
    which will produce an error if assigned to a variable.

"trim ..."
    Prefixing any of the other transformations with the word "trim"
    will result in whitespace being trimmed from the beginning and end
    of the result before processing.

    Examples: "trim string" or "trim list lines"

    Note that "trim value" is useless because the value parser skips
    whitespace anyway.
```

Label patterns

```
A label pattern is a way of expressing one or more labels in a portion
of the source tree. They are not general regular expressions.

They can take the following forms only:

 - Explicit (no wildcard):
     "//foo/bar:baz"
     ":baz"

 - Wildcard target names:
     "//foo/bar:*" (all targets in the //foo/bar/BUILD.gn file)
     ":*"  (all targets in the current build file)

 - Wildcard directory names ("*" is only supported at the end)
     "*"  (all targets)
     "//foo/bar/*"  (all targets in any subdir of //foo/bar)
     "./*"  (all targets in the current build file or sub dirs)

Any of the above forms can additionally take an explicit toolchain.
In this case, the toolchain must be fully qualified (no wildcards
are supported in the toolchain name).

  "//foo:bar(//build/toochain:mac)"
      An explicit target in an explicit toolchain.

  ":*(//build/toolchain/linux:32bit)"
      All targets in the current build file using the 32-bit Linux
      toolchain.

  "//foo/*(//build/toolchain:win)"
      All targets in //foo and any subdirectory using the Windows
      toolchain.
```

nogncheck: Skip an include line from checking.

```
GN's header checker helps validate that the includes match the build
dependency graph. Sometimes an include might be conditional or
otherwise problematic, but you want to specifically allow it. In this
case, it can be whitelisted.
```

```
Include lines containing the substring "nogncheck" will be excluded
from header checking. The most common case is a conditional include:

  #if defined(ENABLE_DOOM_MELON)
  #include "tools/doom_melon/doom_melon.h"  // nogncheck
  #endif

If the build file has a conditional dependency on the corresponding
target that matches the conditional include, everything will always
link correctly:

  source_set("mytarget") {
    ...
    if (enable_doom_melon) {
      defines = [ "ENABLE_DOOM_MELON" ]
      deps += [ "//tools/doom_melon" ]
    }

But GN's header checker does not understand preprocessor directives,
won't know it matches the build dependencies, and will flag this
include as incorrect when the condition is false.
```

More information

```
The topic "gn help check" has general information on how checking
works and advice on fixing problems. Targets can also opt-out of
checking, see "gn help check_includes".
```

Runtime dependencies

```
Runtime dependencies of a target are exposed via the "runtime_deps"
category of "gn desc" (see "gn help desc") or they can be written
at build generation time via write_runtime_deps(), or
--runtime-deps-list-file (see "gn help --runtime-deps-list-file").

To a first approximation, the runtime dependencies of a target are
the set of "data" files, data directories, and the shared libraries
from all transitive dependencies. Executables, shared libraries, and
loadable modules are considered runtime dependencies of themselves.
```

Executables

Executable targets and those executable targets' transitive dependencies are not considered unless that executable is listed in "data_deps". Otherwise, GN assumes that the executable (and everything it requires) is a build-time dependency only.

Actions and copies

Action and copy targets that are listed as "data_deps" will have all of their outputs and data files considered as runtime dependencies. Action and copy targets that are "deps" or "public_deps" will have only their data files considered as runtime dependencies. These targets can list an output file in both the "outputs" and "data" lists to force an output file as a runtime dependency in all cases.

The different rules for deps and data_deps are to express build-time (deps) vs. run-time (data_deps) outputs. If GN counted all build-time copy steps as data dependencies, there would be a lot of extra stuff, and if GN counted all run-time dependencies as regular deps, the build's parallelism would be unnecessarily constrained.

This rule can sometimes lead to unintuitive results. For example, given the three targets:

```
  A  --[data_deps]-->  B  --[deps]-->  ACTION
```

GN would say that A does not have runtime deps on the result of the ACTION, which is often correct. But the purpose of the B target might be to collect many actions into one logic unit, and the "data"-ness of A's dependency is lost. Solutions:

- List the outputs of the action in it's data section (if the results of that action are always runtime files).
- Have B list the action in data_deps (if the outputs of the actions are always runtime files).
- Have B list the action in both deps and data deps (if the outputs might be used in both contexts and you don't care about unnecessary entries in the list of files required at runtime).
- Split B into run-time and build-time versions with the appropriate "deps" for each.

Static libraries and source sets

The results of static_library or source_set targets are not considered

runtime dependencies since these are assumed to be intermediate
targets only. If you need to list a static library as a runtime
dependency, you can manually compute the .a/.lib file name for the
current platform and list it in the "data" list of a target
(possibly on the static library target itself).

Multiple outputs

When a tool produces more than one output, only the first output
is considered. For example, a shared library target may produce a
.dll and a .lib file on Windows. Only the .dll file will be considered
a runtime dependency. This applies only to linker tools, scripts and
copy steps with multiple outputs will also get all outputs listed.

How Source Expansion Works

Source expansion is used for the action_foreach and copy target types
to map source file names to output file names or arguments.

To perform source expansion in the outputs, GN maps every entry in the
sources to every entry in the outputs list, producing the cross
product of all combinations, expanding placeholders (see below).

Source expansion in the args works similarly, but performing the
placeholder substitution produces a different set of arguments for
each invocation of the script.

If no placeholders are found, the outputs or args list will be treated
as a static list of literal file names that do not depend on the
sources.

See "gn help copy" and "gn help action_foreach" for more on how
this is applied.

Placeholders

This section discusses only placeholders for actions. There are other
placeholders used in the definition of tools. See "gn help tool" for
those.

```
{{source}}
    The name of the source file including directory (*). This will
    generally be used for specifying inputs to a script in the
    "args" variable.
      "//foo/bar/baz.txt" => "../../foo/bar/baz.txt"

{{source_file_part}}
    The file part of the source including the extension.
      "//foo/bar/baz.txt" => "baz.txt"

{{source_name_part}}
    The filename part of the source file with no directory or
    extension. This will generally be used for specifying a
    transformation from a source file to a destination file with the
    same name but different extension.
      "//foo/bar/baz.txt" => "baz"

{{source_dir}}
    The directory (*) containing the source file with no
    trailing slash.
      "//foo/bar/baz.txt" => "../../foo/bar"

{{source_root_relative_dir}}
    The path to the source file's directory relative to the source
    root, with no leading "//" or trailing slashes. If the path is
    system-absolute, (beginning in a single slash) this will just
    return the path with no trailing slash. This value will always
    be the same, regardless of whether it appears in the "outputs"
    or "args" section.
      "//foo/bar/baz.txt" => "foo/bar"

{{source_gen_dir}}
    The generated file directory (*) corresponding to the source
    file's path. This will be different than the target's generated
    file directory if the source file is in a different directory
    than the BUILD.gn file.
      "//foo/bar/baz.txt" => "gen/foo/bar"

{{source_out_dir}}
    The object file directory (*) corresponding to the source file's
    path, relative to the build directory. this us be different than
    the target's out directory if the source file is in a different
    directory than the build.gn file.
      "//foo/bar/baz.txt" => "obj/foo/bar"
```

(*) Note on directories

```
Paths containing directories (except the source_root_relative_dir)
will be different depending on what context the expansion is evaluated
in. Generally it should "just work" but it means you can't
concatenate strings containing these values with reasonable results.

Details: source expansions can be used in the "outputs" variable,
the "args" variable, and in calls to "process_file_template". The
"args" are passed to a script which is run from the build directory,
so these directories will relative to the build directory for the
script to find. In the other cases, the directories will be source-
absolute (begin with a "//") because the results of those expansions
will be handled by GN internally.
```

Examples

```
Non-varying outputs:
  action("hardcoded_outputs") {
    sources = [ "input1.idl", "input2.idl" ]
    outputs = [ "$target_out_dir/output1.dat",
                "$target_out_dir/output2.dat" ]
  }
The outputs in this case will be the two literal files given.

Varying outputs:
  action_foreach("varying_outputs") {
    sources = [ "input1.idl", "input2.idl" ]
    outputs = [ "{{source_gen_dir}}/{{source_name_part}}.h",
                "{{source_gen_dir}}/{{source_name_part}}.cc" ]
  }
Performing source expansion will result in the following output names:
  //out/Debug/obj/mydirectory/input1.h
  //out/Debug/obj/mydirectory/input1.cc
  //out/Debug/obj/mydirectory/input2.h
  //out/Debug/obj/mydirectory/input2.cc
```

**Available global switches ** Do “gn help –the_switch_you_want_help_on” for more. Individual commands may take command-specific switches not listed here. See the help on your specific command for more.

```
**  \--args**: Specifies build arguments overrides.
**  \--color**: Force colored output.
**  \--dotfile**: Override the name of the ".gn" file.
```

```
**  \--fail-on-unused-args**: Treat unused build args as fatal errors.
**  \--markdown**: Write help output in the Markdown format.
**  \--nocolor**: Force non-colored output.
**  -q**: Quiet mode. Don't print output on success.
**  \--root**: Explicitly specify source root.
**  \--runtime-deps-list-file**: Save runtime dependencies for targets in file
**  \--script-executable**: Set the executable used to execute scripts.
**  \--threads**: Specify number of worker threads.
**  \--time**: Outputs a summary of how long everything took.
**  \--tracelog**: Writes a Chrome-compatible trace log to the given file.
**  -v**: Verbose logging.
**  \--version**: Prints the GN version number and exits.
```