GN Reference

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

Contents

- bundle_root_dir: Expansion of {{bundle_root_dir}} in create_bundle.
- cflags: [string list] Flags passed to all C compiler variants.
- cflags_c: [string list] Flags passed to the C compiler.
- cflags_cc: [string list] Flags passed to the C++ compiler.
- cflags_objc: [string list] Flags passed to the Objective C compiler.
- cflags_objcc: [string list] Flags passed to the Objective C++ compiler.
- check_includes: [boolean] Controls whether a target's files are checked.
- code_signing_args: [string list] Arguments passed to code signing script.
- code_signing_outputs: [file list] Output files for code signing step.
- code_signing_script: [file name] Script for code signing.
- code_signing_sources: [file list] Sources for code signing step.
- complete_static_lib: [boolean] Links all deps into a static library.
- configs: [label list] Configs applying to this target or config.
- contents: Contents to write to file.
- crate_name: [string] The name for the compiled crate.
- crate_root: [string] The root source file for a binary or library.
- crate_type: [string] The type of linkage to use on a shared_library.
- data: [file list] Runtime data file dependencies.
- data_deps: [label list] Non-linked dependencies.
- data_keys: [string list] Keys from which to collect metadata.
- defines: [string list] C preprocessor defines.
- depfile: [string] File name for input dependencies for actions.
- deps: [label list] Private linked dependencies.
- externs: [scope] Set of Rust crate-dependency pairs.
- framework_dirs: [directory list] Additional framework search directories.
- frameworks: [name list] Name of frameworks that must be linked.
- friend: [label pattern list] Allow targets to include private headers.
- gen_deps: [label list] Declares targets that should generate when this one does.
- include_dirs: [directory list] Additional include directories.
- inputs: [file list] Additional compile-time dependencies.
- ldflags: [string list] Flags passed to the linker.
- lib_dirs: [directory list] Additional library directories.
- libs: [string list] Additional libraries to link.
- metadata: [scope] Metadata of this target.
- module_name: [string] The name for the compiled module.
- output_conversion: Data format for generated_file targets.
- output_dir: [directory] Directory to put output file in.
- output_extension: [string] Value to use for the output's file extension.
- output_name: [string] Name for the output file other than the default.
- output_prefix_override: [boolean] Don't use prefix for output name.
- outputs: [file list] Output files for actions and copy targets.
- partial_info_plist: [filename] Path plist from asset catalog compiler.
- pool: [string] Label of the pool used by the action.
- precompiled_header: [string] Header file to precompile.
- precompiled_header_type: [string] "gcc" or "msvc".
- precompiled_source: [file name] Source file to precompile.
- product_type: [string] Product type for Xcode projects.
- public: [file list] Declare public header files for a target.
- public_configs: [label list] Configs applied to dependents.
- public_deps: [label list] Declare public dependencies.
- rebase: [boolean] Rebase collected metadata as files.
- response_file_contents: [string list] Contents of .rsp file for actions.

    - script: [file name] Script file for actions.
    - sources: [file list] Source files for a target.
    - swiftflags: [string list] Flags passed to the swift compiler.
    - testonly: [boolean] Declares a target must only be used for testing.
    - visibility: [label list] A list of labels that can depend on a target.
    - walk_keys: [string list] Key(s) for managing the metadata collection walk.
    - weak_frameworks: [name list] Name of frameworks that must be weak linked.
    - write_runtime_deps: Writes the target's runtime_deps to the given path.
    - xcasset_compiler_flags: [string list] Flags passed to xcassets compiler
    - xcode_extra_attributes: [scope] Extra attributes for Xcode projects.
    - xcode_test_application_name: [string] Name for Xcode test target.
- Other help topics
    - all: Print all the help at once
    - buildargs: How build arguments work.
    - dotfile: Info about the toplevel .gn file.
    - execution: Build graph and execution overview.
    - grammar: Language and grammar for GN build files.
    - input_conversion: Processing input from exec_script and read_file.
    - file_pattern: Matching more than one file.
    - label_pattern: Matching more than one label.
    - labels: About labels.
    - metadata_collection: About metadata and its collection.
    - ninja_rules: How Ninja build rules are named.
    - nogncheck: Annotating includes for checking.
    - output_conversion: Specifies how to transform a value to output.
    - runtime_deps: How runtime dependency computation works.
    - source_expansion: Map sources to outputs for scripts.
    - switches: Show available command-line switches.

Commands

gn analyze <out_dir> <input_path> <output_path>

```
Analyze which targets are affected by a list of files.

This command takes three arguments:

out_dir is the path to the build directory.

input_path is a path to a file containing a JSON object with three fields:

 - "files": A list of the filenames to check.

 - "test_targets": A list of the labels for targets that are needed to run
   the tests we wish to run.

 - "additional_compile_targets" (optional): A list of the labels for targets
   that we wish to rebuild, but aren't necessarily needed for testing. The
   important difference between this field and "test_targets" is that if an
   item in the additional_compile_targets list refers to a group, then any
   dependencies of that group will be returned if they are out of date, but
   the group itself does not need to be. If the dependencies themselves are
   groups, the same filtering is repeated. This filtering can be used to
```

```
    avoid rebuilding dependencies of a group that are unaffected by the input
    files. The list may also contain the string "all" to refer to a
    pseudo-group that contains every root target in the build graph.

    This filtering behavior is also known as "pruning" the list of compile
    targets.

    If "additional_compile_targets" is absent, it defaults to the empty list.

If input_path is -, input is read from stdin.

output_path is a path indicating where the results of the command are to be
written. The results will be a file containing a JSON object with one or more
of following fields:

 - "compile_targets": A list of the labels derived from the input
    compile_targets list that are affected by the input files. Due to the way
    the filtering works for compile targets as described above, this list may
    contain targets that do not appear in the input list.

 - "test_targets": A list of the labels from the input test_targets list that
    are affected by the input files. This list will be a proper subset of the
    input list.

 - "invalid_targets": A list of any names from the input that do not exist in
    the build graph. If this list is non-empty, the "error" field will also be
    set to "Invalid targets".

 - "status": A string containing one of three values:

     - "Found dependency"
     - "No dependency"
     - "Found dependency (all)"

    In the first case, the lists returned in compile_targets and test_targets
    should be passed to ninja to build. In the second case, nothing was
    affected and no build is necessary. In the third case, GN could not
    determine the correct answer and returned the input as the output in order
    to be safe.

 - "error": This will only be present if an error occurred, and will contain
    a string describing the error. This includes cases where the input file is
    not in the right format, or contains invalid targets.

If output_path is -, output is written to stdout.

The command returns 1 if it is unable to read the input file or write the
output file, or if there is something wrong with the build such that gen
would also fail, and 0 otherwise. In particular, it returns 0 even if the
"error" key is non-empty and a non-fatal error occurred. In other words, it
tries really hard to always write something to the output JSON and convey
errors that way rather than via return codes.
```

gn args: (command-line tool)

```
Display or configure arguments declared by the build.

  gn args <out_dir> [--list] [--short] [--args] [--overrides-only]

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. 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"

    To find your editor on Posix, GN will search the environment variables in
    order: GN_EDITOR, VISUAL, and EDITOR. On Windows GN will open the command
    associated with .txt files.

    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] [--overrides-only] [--json]
    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, current value for the
    build, default value (if different than the current value), and comment
    preceding the declaration.

    If --short is specified, only the names and current values will be
    printed.

    If --overrides-only is specified, only the names and current values of
    arguments that have been overridden (i.e. non-default arguments) will
    be printed. Overrides come from the <out_dir>/args.gn file and //.gn

    If --json is specified, the output will be emitted in json format.
    JSON schema for output:
    [
      {
        "name": variable_name,
        "current": {
          "value": overridden_value,
          "file": file_name,
          "line": line_no
        },
        "default": {
          "value": default_value,
          "file": file_name,
          "line": line_no
```

```
        },
        "comment": comment_string
      },
      ...
    ]
```

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 --short --overrides-only
  Prints overridden arguments 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 <out_dir> [<label_pattern>] [--force] [--check-generated]

```
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

```
--check-generated
    Generated files are normally not checked since they do not exist
    until after a build. With this flag, those generated files that
    can be found on disk are also checked.

--check-system
   Check system style includes (using <angle brackets>) in addition to
   "double quote" includes.

--default-toolchain
    Normally wildcard targets are matched in all toolchains. This
```

```
    switch makes wildcard labels with no explicit toolchain reference
    only match targets in the default toolchain.

    Non-wildcard inputs with no explicit toolchain specification will
    always match only a target in the default toolchain if one exists.

--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 in the list
check_targets (see "gn help dotfile"). Alternatively, the .gn file may
specify a list of targets not to be checked in no_check_targets. If a label
pattern is specified on the command line, neither check_targets or
no_check_targets is used.

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.

  - Generated files (that might not exist yet) are ignored unless
    the --check-generated flag is provided.

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

  - Includes using "quotes" are always checked.
      If system style checking is enabled, includes using <angle brackets>
      are also checked.

  - Include paths are assumed to be relative to any of the "include_dirs" for
    the target (including the implicit current dir).

  - 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.

  - Outputs 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 is difficult to fix or doesn't care
about include checks 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 <out_dir>...

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

gn clean_stale [--ninja-executable=...] <out_dir>...

```
Removes the no longer needed output files from the build directory and prunes
their records from the ninja build log and dependency database. These are
output files that were generated from previous builds, but the current build
graph no longer references them.

This command requires a ninja executable of at least version 1.10.0. The
executable must be provided by the --ninja-executable switch.
```

Options

```
--ninja-executable=<string>
    Can be used to specify the ninja executable to use.
```

gn desc

```
gn desc <out_dir> <label or pattern> [<what to show>] [--blame]
        [--format=json]

Displays information about a given target or config. The 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 <what to show>

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

all_dependent_configs
allow_circular_includes_from
arflags [--blame]
args
cflags [--blame]
cflags_c [--blame]
cflags_cc [--blame]
check_includes
configs [--tree] (see below)
data_keys
defines [--blame]
depfile
deps [--all] [--tree] (see below)
framework_dirs
frameworks
include_dirs [--blame]
inputs
ldflags [--blame]
lib_dirs
libs
metadata
output_conversion
outputs
```

```
public_configs
public
rebase
script
sources
testonly
visibility
walk_keys
weak_frameworks

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

```
--default-toolchain
    Normally wildcard targets are matched in all toolchains. This
    switch makes wildcard labels with no explicit toolchain reference
    only match targets in the default toolchain.

    Non-wildcard inputs with no explicit toolchain specification will
    always match only a target in the default toolchain if one exists.

--format=json
    Format the output as JSON instead of text.
```

Target flags

```
--blame
    Used with any value specified on a config, this will name the config that
    causes that target to get the flag. This doesn't currently work for libs,
    lib_dirs, frameworks, weak_frameworks and framework_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 outputs

```
The "outputs" section will list all outputs that apply, including the outputs
computed from the tool definition (eg for "executable", "static_library", ...
targets).
```

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 | <list of build_files...>)

```
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 preceding the assignment. e.g.

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

Arguments

```
--dry-run
    Prints the list of files that would be reformatted but does not write
    anything to 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[=( text | json )]
    Dumps the parse tree to stdout and does not update the file or print
    formatted output. If no format is specified, text format will be used.

--stdin
    Read input from stdin and write to stdout rather than update a file
    in-place.

--read-tree=json
    Reads an AST from stdin in the format output by --dump-tree=json and
    uses that as the parse tree. (The only read-tree format currently
    supported is json.) The given .gn file will be overwritten. This can be
    used to programmatically transform .gn files.
```

Examples

```
gn format //some/BUILD.gn //some/other/BUILD.gn //and/another/BUILD.gn
gn format some\\BUILD.gn
gn format /abspath/some/BUILD.gn
gn format --stdin
gn format --read-tree=json //rewritten/BUILD.gn
```

gn gen [--check] [<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

"gn gen --check" is the same as running "gn check". "gn gen --check=system" is
the same as running "gn check --check-system".  See "gn help check" for
documentation on that mode.

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

General options

```
--ninja-executable=<string>
    Can be used to specify the ninja executable to use. This executable will
    be used as an IDE option to indicate which ninja to use for building. This
    executable will also be used as part of the gen process for triggering a
    restat on generated ninja files and for use with --clean-stale.

--clean-stale
    This option will cause no longer needed output files to be removed from
    the build directory, and their records pruned from the ninja build log and
    dependency database after the ninja build graph has been generated. This
    option requires a ninja executable of at least version 1.10.0. It can be
    provided by the --ninja-executable switch. Also see "gn help clean_stale".
```

IDE options

```
GN optionally generates files for IDE. Files won't be overwritten if their
contents don't change. 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: 2019)
    "vs2013" - Visual Studio 2013 project/solution files.
    "vs2015" - Visual Studio 2015 project/solution files.
    "vs2017" - Visual Studio 2017 project/solution files.
    "vs2019" - Visual Studio 2019 project/solution files.
    "vs2022" - Visual Studio 2022 project/solution files.
```

```
    "xcode" - Xcode workspace/solution files.
    "qtcreator" - QtCreator project files.
    "json" - JSON file containing target information

--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
    and their dependencies will be included in the solution. Only used for
    Visual Studio, Xcode and JSON.
```

Visual Studio Flags

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

--no-deps
    Don't include targets dependencies to the solution. Changes the way how
    --filters option works. Only directly matching targets are included.

--winsdk=<sdk_version>
    Use the specified Windows 10 SDK version to generate project files.
    As an example, "10.0.15063.0" can be specified to use Creators Update SDK
    instead of the default one.

--ninja-executable=<string>
    Can be used to specify the ninja executable to use when building.

--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".
```

Xcode Flags

```
--xcode-project=<file_name>
    Override default Xcode project file name ("all"). The project file is
    written to the root build directory.

--xcode-build-system=<value>
    Configure the build system to use for the Xcode project. Supported
    values are (default to "legacy"):
    "legacy" - Legacy Build system
    "new" - New Build System

--ninja-executable=<string>
    Can be used to specify the ninja executable to use when building.

--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".

--ide-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

```
  --ide-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 emitted.
```

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.
```

Generic JSON Output

```
  Dumps target information to a JSON file and optionally invokes a
  python script on the generated file. See the comments at the beginning
  of json_project_writer.cc and desc_builder.cc for an overview of the JSON
  file format.

  --json-file-name=<json_file_name>
      Overrides default file name (project.json) of generated JSON file.

  --json-ide-script=<path_to_python_script>
      Executes python script after the JSON file is generated or updated with
      new content. Path can be project absolute (//), system absolute (/) or
      relative, in which case the output directory will be base. Path to
      generated JSON file will be first argument when invoking script.

  --json-ide-script-args=<argument>
      Optional second argument that will passed to executed script.
```

Compilation Database

```
  --export-rust-project
      Produces a rust-project.json file in the root of the build directory
      This is used for various tools in the Rust ecosystem allowing for the
      replay of individual compilations independent of the build system.
      This is an unstable format and likely to change without warning.

  --export-compile-commands[=<target_name1,target_name2...>]
      Produces a compile_commands.json file in the root of the build directory
      containing an array of "command objects", where each command object
      specifies one way a translation unit is compiled in the project. If a list
      of target_name is supplied, only targets that are reachable from any
      target in any build file whose name is target_name will be used for
      "command objects" generation, otherwise all available targets will be
used.
      This is used for various Clang-based tooling, allowing for the replay of
      individual compilations independent of the build system.
      e.g. "foo" will match:
```

```
    - "//path/to/src:foo"
    - "//other/path:foo"
    - "//foo:foo"
    and not match:
    - "//foo:bar"
```

gn help <anything>

```
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 <out_dir> [<label_pattern>] [--default-toolchain] [--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.

--default-toolchain
    Normally wildcard targets are matched in all toolchains. This
    switch makes wildcard labels with no explicit toolchain reference
    only match targets in the default toolchain.
```

```
      Non-wildcard inputs with no explicit toolchain specification will
      always match only a target in the default toolchain if one exists.

  --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 meta

```
  gn meta <out_dir> <target>* --data=<key>[,<key>*]* [--walk=<key>[,<key>*]*]
          [--rebase=<dest dir>]

  Lists collected metaresults of all given targets for the given data key(s),
  collecting metadata dependencies as specified by the given walk key(s).

  See `gn help generated_file` for more information on the walk.
```

Arguments

```
  <target(s)>
    A list of target labels from which to initiate the walk.

  --data
    A list of keys from which to extract data. In each target walked, its
metadata
    scope is checked for the presence of these keys. If present, the contents of
    those variable in the scope are appended to the results list.
```

```
  --walk (optional)
    A list of keys from which to control the walk. In each target walked, its
    metadata scope is checked for the presence of any of these keys. If present,
    the contents of those variables is checked to ensure that it is a label of
    a valid dependency of the target and then added to the set of targets to
walk.
    If the empty string ("") is present in any of these keys, all deps and
data_deps
    are added to the walk set.

  --rebase (optional)
    A destination directory onto which to rebase any paths found. If set, all
    collected metadata will be rebased onto this path. This option will throw
errors
    if collected metadata is not a list of strings.
```

Examples

```
  gn meta out/Debug "//base/foo" --data=files
      Lists collected metaresults for the `files` key in the //base/foo:foo
      target and all of its dependency tree.

  gn meta out/Debug "//base/foo" --data=files --data=other
      Lists collected metaresults for the `files` and `other` keys in the
      //base/foo:foo target and all of its dependency tree.

  gn meta out/Debug "//base/foo" --data=files --walk=stop
      Lists collected metaresults for the `files` key in the //base/foo:foo
      target and all of the dependencies listed in the `stop` key (and so on).

  gn meta out/Debug "//base/foo" --data=files --rebase="/"
      Lists collected metaresults for the `files` key in the //base/foo:foo
      target and all of its dependency tree, rebasing the strings in the `files`
      key onto the source directory of the target's declaration relative to "/".
```

gn outputs <out_dir> <list of target or file names...>

```
  Lists the output files corresponding to the given target(s) or file name(s).
  There can be multiple outputs because there can be more than one output
  generated by a build step, and there can be more than one toolchain matched.
  You can also list multiple inputs which will generate a union of all the
  outputs from those inputs.

   - The input target/file names are relative to the current directory.

   - The output file names are relative to the root build directory.

   This command is useful for finding a ninja command that will build only a
   portion of the build.
```

Target outputs

```
If the parameter is a target name that includes a toolchain, it will match
only that target in that toolchain. If no toolchain is specified, it will
match all targets with that name in any toolchain.

The result will be the outputs specified by that target which could be a
library, executable, output of an action, a stamp file, etc.
```

File outputs

```
If the parameter is a file name it will compute the output for that compile
step for all targets in all toolchains that contain that file as a source
file.

If the source is not compiled (e.g. a header or text file), the command will
produce no output.

If the source is listed as an "input" to a binary target or action will
resolve to that target's outputs.
```

Example

```
gn outputs out/debug some/directory:some_target
    Find the outputs of a given target.

gn outputs out/debug src/project/my_file.cc | xargs ninja -C out/debug
    Compiles just the given source file in all toolchains it's referenced in.

git diff --name-only | xargs gn outputs out/x64 | xargs ninja -C out/x64
    Compiles all files changed in git.
```

gn path <out_dir> <target_one> <target_two>

```
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.
```

Interesting paths

```
In a large project, there can be 100's of millions of unique paths between a
very high level and a common low-level target. To make the output more useful
(and terminate in a reasonable time), GN will not revisit sub-paths
previously known to lead to the target.
```

Options

```
--all
   Prints all "interesting" 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 //gn
```

gn refs

```
gn refs <out_dir> (<label_pattern>|<label>|<file>|@<response_file>)* [--all]
        [--default-toolchain] [--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.

--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.

--default-toolchain
    Normally wildcard targets are matched in all toolchains. This
    switch makes wildcard labels with no explicit toolchain reference
    only match targets in the default toolchain.

    Non-wildcard inputs with no explicit toolchain specification will
    always match only a target in the default toolchain if one exists.

-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 //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.
```

Target declarations

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 one 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).

GN sets Ninja's flag 'restat = 1` for all action commands. This means
that Ninja will check the timestamp of the output after the action
completes. If output timestamp is unchanged, the step will be treated
as if it never needed to be rebuilt, potentially eliminating some
downstream steps for incremental builds. Scripts can improve build
performance by taking care not to change the timstamp of the output
file(s) if the contents have not changed.
```

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, data, data_deps, depfile, deps, inputs, metadata, outputs*, pool,
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).

GN sets Ninja's flag 'restat = 1` for all action commands. This means
that Ninja will check the timestamp of the output after the action
completes. If output timestamp is unchanged, the step will be treated
as if it never needed to be rebuilt, potentially eliminating some
downstream steps for incremental builds. Scripts can improve build
performance by taking care not to change the timstamp of the output
file(s) if the contents have not changed.
```

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, data, data_deps, depfile, deps, inputs, metadata, outputs*, pool,
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 it 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(target_gen_dir, root_build_dir) +
      "/{{source_name_part}}.h" ]
}
```

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

```
This target type allows one 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/macOS bundle. In cross-platform projects, it is advised to put it
behind iOS/macOS conditionals.

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

Variables

```
sources*, outputs*, deps, data_deps, metadata, 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}}"
  ]
}
```

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/macOS] Build an iOS or macOS bundle.

```
This target generates an iOS or macOS 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" are be used for the expansion of {{bundle_*_dir}} rules in
"bundle_data" outputs. The properties are optional but must be defined if any
of the "bundle_data" target use them.

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

If a create_bundle is specified as a data_deps for another target, the bundle
is considered a leaf, and its public and private dependencies will not
contribute to any data or data_deps. Required runtime dependencies should be
placed in the bundle. A create_bundle can declare its own explicit data and
data_deps, however.
```

Code signing

```
Some bundle needs to be code signed as part of the build (on iOS all
application needs to be code signed to run on a device). The code signature
can be configured via the code_signing_script variable.

If set, code_signing_script is the path of a script that invoked after all
files have been moved into the bundle. The script must not change any file in
the bundle, but may add new files.

If code_signing_script is defined, then code_signing_outputs must also be
defined and non-empty to inform when the script needs to be re-run. The
code_signing_args will be passed as is to the script (so path have to be
rebased) and additional inputs may be listed with the variable
code_signing_sources.
```

Variables

```
bundle_root_dir, bundle_contents_dir, bundle_resources_dir,
bundle_executable_dir, bundle_deps_filter, deps, data_deps, public_deps,
visibility, product_type, code_signing_args, code_signing_script,
code_signing_sources, code_signing_outputs, xcode_extra_attributes,
xcode_test_application_name, partial_info_plist, metadata
```

Example

```
# Defines a template to create an application. On most platform, this is just
# an alias for an "executable" target, but on iOS/macOS, 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("${app_name}_bundle_info_plist") {
      public_deps = [ ":${app_name}_generate_info_plist" ]
      sources = [ "$gen_path/Info.plist" ]
      outputs = [ "{{bundle_contents_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)
  }

  code_signing =
      defined(invoker.code_signing) && invoker.code_signing

  if (!is_ios || !code_signing) {
    bundle_data("${app_name}_bundle_executable") {
      public_deps = [ ":${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"

    if (is_ios) {
      bundle_root_dir = "$root_build_dir/$target_name"
      bundle_contents_dir = bundle_root_dir
      bundle_resources_dir = bundle_contents_dir
      bundle_executable_dir = bundle_contents_dir

      xcode_extra_attributes = {
        ONLY_ACTIVE_ARCH = "YES"
        DEBUG_INFORMATION_FORMAT = "dwarf"
      }
    } else {
      bundle_root_dir = "$root_build_dir/$target_name"
      bundle_contents_dir  = "$bundle_root_dir/Contents"
      bundle_resources_dir = "$bundle_contents_dir/Resources"
      bundle_executable_dir = "$bundle_contents_dir/MacOS"
    }
    deps = [ ":${app_name}_bundle_info_plist" ]
    if (is_ios && code_signing) {
      deps += [ ":${app_name}_generate_executable" ]
      code_signing_script = "//build/config/ios/codesign.py"
      code_signing_sources = [
        invoker.entitlements_path,
        "$target_gen_dir/$app_name",
      ]
      code_signing_outputs = [
        "$bundle_root_dir/$app_name",
        "$bundle_root_dir/_CodeSignature/CodeResources",
        "$bundle_root_dir/embedded.mobileprovision",
        "$target_gen_dir/$app_name.xcent",
      ]
      code_signing_args = [
        "-i=" + ios_code_signing_identity,
        "-b=" + rebase_path(
            "$target_gen_dir/$app_name", root_build_dir),
        "-e=" + rebase_path(
            invoker.entitlements_path, root_build_dir),
        "-e=" + rebase_path(
            "$target_gen_dir/$app_name.xcent", root_build_dir),
        rebase_path(bundle_root_dir, root_build_dir),
```

```
        ]
      } else {
        deps += [ ":${app_name}_bundle_executable" ]
      }
    }
  }
}
```

executable: Declare an executable target.

Language and compilation

```
The tools and commands used to create this target type will be
determined by the source files in its sources. Targets containing
multiple compiler-incompatible languages are not allowed (e.g. a
target containing both C and C++ sources is acceptable, but a
target containing C and Rust sources is not).
```

Variables

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, inputs, ldflags, lib_dirs,
       libs, precompiled_header, precompiled_source, rustflags,
       rustenv, swiftflags
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, friend, inputs, metadata,
         output_name, output_extension, public, sources, testonly,
         visibility
Rust variables: aliased_deps, crate_root, crate_name
```

generated_file: Declare a generated_file target.

```
Writes data value(s) to disk on resolution. This target type mirrors some
functionality of the write_file() function, but also provides the ability to
collect metadata from its dependencies on resolution rather than writing out
at parse time.

The `outputs` variable is required to be a list with a single element,
specifying the intended location of the output file.

The `output_conversion` variable specified the format to write the
value. See `gn help io_conversion`.

One of `contents` or `data_keys` must be specified; use of `contents` will
write the contents of that value to file, while use of `data_keys` will
trigger a metadata collection walk based on the dependencies of the target and
the optional values of the `rebase` and `walk_keys` variables. See
`gn help metadata`.

Collected metadata, if specified, will be returned in postorder of
dependencies. See the example for details.
```

Example (metadata collection)

```
Given the following targets defined in //base/BUILD.gn, where A depends on B
and B depends on C and D:

  group("a") {
    metadata = {
      doom_melon = [ "enable" ]
      my_files = [ "foo.cpp" ]

      # Note: this is functionally equivalent to not defining `my_barrier`
      # at all in this target's metadata.
      my_barrier = [ "" ]
    }

    deps = [ ":b" ]
  }

  group("b") {
    metadata = {
      my_files = [ "bar.cpp" ]
      my_barrier = [ ":c" ]
    }

    deps = [ ":c", ":d" ]
  }

  group("c") {
    metadata = {
      doom_melon = [ "disable" ]
      my_files = [ "baz.cpp" ]
    }
  }

  group("d") {
    metadata = {
      my_files = [ "missing.cpp" ]
    }
  }

If the following generated_file target is defined:

  generated_file("my_files_metadata") {
    outputs = [ "$root_build_dir/my_files.json" ]
    data_keys = [ "my_files" ]

    deps = [ "//base:a" ]
  }

The following will be written to "$root_build_dir/my_files.json" (less the
comments):
  [
    "baz.cpp",  // from //base:c via //base:b
    "missing.cpp"  // from //base:d via //base:b
    "bar.cpp",  // from //base:b via //base:a
    "foo.cpp",  // from //base:a
  ]
```

```
Alternatively, as an example of using walk_keys, if the following
generated_file target is defined:

generated_file("my_files_metadata") {
  outputs = [ "$root_build_dir/my_files.json" ]
  data_keys = [ "my_files" ]
  walk_keys = [ "my_barrier" ]

  deps = [ "//base:a" ]
}

The following will be written to "$root_build_dir/my_files.json" (again less
the comments):
  [
    "baz.cpp",  // from //base:c via //base:b
    "bar.cpp",  // from //base:b via //base:a
    "foo.cpp",  // from //base:a
  ]

If `rebase` is used in the following generated_file target:

generated_file("my_files_metadata") {
  outputs = [ "$root_build_dir/my_files.json" ]
  data_keys = [ "my_files" ]
  walk_keys = [ "my_barrier" ]
  rebase = root_build_dir

  deps = [ "//base:a" ]
}

The following will be written to "$root_build_dir/my_files.json" (again less
the comments) (assuming root_build_dir = "//out"):
  [
    "../base/baz.cpp",  // from //base:c via //base:b
    "../base/bar.cpp",  // from //base:b via //base:a
    "../base/foo.cpp",  // from //base:a
  ]
```

Variables

```
contents
data_keys
rebase
walk_keys
output_conversion
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
```

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.
```

Variables

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

Example

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

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.
```

Language and compilation

```
The tools and commands used to create this target type will be
determined by the source files in its sources. Targets containing
multiple compiler-incompatible languages are not allowed (e.g. a
target containing both C and C++ sources is acceptable, but a
target containing C and Rust sources is not).
```

Variables

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, inputs, ldflags, lib_dirs,
       libs, precompiled_header, precompiled_source, rustflags,
       rustenv, swiftflags
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, friend, inputs, metadata,
         output_name, output_extension, public, sources, testonly,
         visibility
Rust variables: aliased_deps, crate_root, crate_name, crate_type
```

rust_library: Declare a Rust library target.

```
A Rust library is an archive containing additional rust-c provided metadata.
These are the files produced by the rustc compiler with the `.rlib`
extension, and are the intermediate step for most Rust-based binaries.
```

Language and compilation

```
The tools and commands used to create this target type will be
determined by the source files in its sources. Targets containing
multiple compiler-incompatible languages are not allowed (e.g. a
target containing both C and C++ sources is acceptable, but a
target containing C and Rust sources is not).
```

Variables

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, inputs, ldflags, lib_dirs,
       libs, precompiled_header, precompiled_source, rustflags,
       rustenv, swiftflags
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, friend, inputs, metadata,
         output_name, output_extension, public, sources, testonly,
         visibility
Rust variables: aliased_deps, crate_root, crate_name
```

rust_proc_macro: Declare a Rust procedural macro target.

```
A Rust procedural macro allows creating syntax extensions as execution of a
function. They are compiled as dynamic libraries and used by the compiler at
runtime.

Their use is the same as of other Rust libraries, but their build has some
additional restrictions in terms of supported flags.
```

Language and compilation

```
The tools and commands used to create this target type will be
determined by the source files in its sources. Targets containing
multiple compiler-incompatible languages are not allowed (e.g. a
target containing both C and C++ sources is acceptable, but a
target containing C and Rust sources is not).
```

Variables

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, inputs, ldflags, lib_dirs,
       libs, precompiled_header, precompiled_source, rustflags,
       rustenv, swiftflags
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, friend, inputs, metadata,
         output_name, output_extension, public, sources, testonly,
         visibility
Rust variables: aliased_deps, crate_root, crate_name
```

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.
```

Language and compilation

```
The tools and commands used to create this target type will be
determined by the source files in its sources. Targets containing
multiple compiler-incompatible languages are not allowed (e.g. a
target containing both C and C++ sources is acceptable, but a
target containing C and Rust sources is not).
```

Variables

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, inputs, ldflags, lib_dirs,
       libs, precompiled_header, precompiled_source, rustflags,
       rustenv, swiftflags
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, friend, inputs, metadata,
         output_name, output_extension, public, sources, testonly,
         visibility
Rust variables: aliased_deps, crate_root, crate_name, crate_type
```

source_set: Declare a source set target.

```
Only C-language source sets are supported at the moment.
```

C-language source_sets

```
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 source 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, inputs, ldflags, lib_dirs,
       libs, precompiled_header, precompiled_source, rustflags,
       rustenv, swiftflags
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, friend, inputs, metadata,
         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

```
complete_static_lib
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, inputs, ldflags, lib_dirs,
       libs, precompiled_header, precompiled_source, rustflags,
       rustenv, swiftflags
Deps: data_deps, deps, public_deps
Dependent configs: all_dependent_configs, public_configs
General: check_includes, configs, data, friend, inputs, metadata,
         output_name, output_extension, public, sources, testonly,
         visibility
Rust variables: aliased_deps, crate_root, crate_name

The tools and commands used to create this target type will be
determined by the source files in its sources. Targets containing
multiple compiler-incompatible languages are not allowed (e.g. a
target containing both C and C++ sources is acceptable, but a
target containing C and Rust sources is not).
```

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") {
  ...
}
```

Buildfile functions

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");
```

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".
```

More background

```
Configs solve a problem where the build system needs to have a higher-level
understanding of various compiler settings. For example, some compiler flags
have to appear in a certain order relative to each other, some settings like
defines and flags logically go together, and the build system needs to
de-duplicate flags even though raw command-line parameters can't always be
operated on in that way.

The config gives a name to a group of settings that can then be reasoned
about by GN. GN can know that configs with the same label are the same thing
so can be de-duplicated. It allows related settings to be grouped so they
are added or removed as a unit. And it allows targets to refer to settings
with conceptual names ("no_rtti", "enable_exceptions", etc.) rather than
having to hard-coding every compiler's flags each time they are referred to.
```

Variables valid in a config definition

```
Flags: cflags, cflags_c, cflags_cc, cflags_objc, cflags_objcc,
       asmflags, defines, include_dirs, inputs, ldflags, lib_dirs,
       libs, precompiled_header, precompiled_source, rustflags,
       rustenv, swiftflags
Nested configs: configs
General: visibility
```

Variables on a target used to apply configs

```
all_dependent_configs, configs, public_configs
```

Example

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

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

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_args() block executes. Any variable defined in the enclosing
    scope is available for reading, but any variable defined earlier in
    the current scope is not (since the overrides haven't been applied yet).
```

```
   2. At the end of executing the block, any variables set within that scope
      are saved, with the values specified in the block used as the "default
value"
      for that argument. Once saved, these variables are available for override
      via args.gn.

   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.

    - Because you cannot read the value of a variable defined in the same
      block, if you need to make the default value of one arg depend
      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=true"
  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").

The default script interpreter is Python ("python" on POSIX, "python.exe" or
"python.bat" on Windows). This can be configured by the script_executable
variable, see "gn help dotfile".
```

Arguments:

```
filename:
    File name of 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 io_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")
```

filter_exclude: Remove values that match a set of patterns.

```
filter_exclude(values, exclude_patterns)

The argument values must be a list of strings.

The argument exclude_patterns must be a list of file patterns (see
"gn help file_pattern"). Any elements in values matching at least one
of those patterns will be excluded.
```

Examples

```
values = [ "foo.cc", "foo.h", "foo.proto" ]
result = filter_exclude(values, [ "*.proto" ])
# result will be [ "foo.cc", "foo.h" ]
```

filter_include: Remove values that do not match a set of patterns.

```
filter_include(values, include_patterns)

The argument values must be a list of strings.

The argument include_patterns must be a list of file patterns (see
"gn help file_pattern"). Only elements from values matching at least
one of the pattern will be included.
```

Examples

```
values = [ "foo.cc", "foo.h", "foo.proto" ]
result = filter_include(values, [ "*.proto" ])
# result will be [ "foo.proto" ]
```

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 loop will iterate over a copy of <list>
so mutating it inside the loop will not affect iteration.

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).

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

```
# forward_variables_from(invoker, ["foo"])
# is equivalent to:
assert(!defined(foo))
if (defined(invoker.foo)) {
  foo = invoker.foo
}

# 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 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", "target_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, generated_file, 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, copy, and generated_file 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").

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 environment 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")
```

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")
```

not_needed: Mark variables from scope as not needed.

```
not_needed(variable_list_or_star, variable_to_ignore_list = [])
not_needed(from_scope, variable_list_or_star,
           variable_to_ignore_list = [])

Mark the variables in the current or given scope as not needed, which means
you will not get an error about unused variables for these. The
variable_to_ignore_list allows excluding variables from "all matches" if
variable_list_or_star is "*".
```

Example

```
not_needed("*", [ "config" ])
not_needed([ "data_deps", "deps" ])
not_needed(invoker, "*", [ "config" ])
not_needed(invoker, [ "data_deps", "deps" ])
```

pool: Defines a pool object.

```
Pool objects can be applied to a tool to limit the parallelism of the
build. This object has a single property "depth" corresponding to
the number of tasks that may run simultaneously.

As the file containing the pool definition may be executed in the
context of more than one toolchain it is recommended to specify an
explicit toolchain when defining and referencing a pool.

A pool named "console" defined in the root build file represents Ninja's
console pool. Targets using this pool will have access to the console's
stdin and stdout, and output will not be buffered. This special pool must
have a depth of 1. Pools not defined in the root must not be named "console".
The console pool can only be defined for the default toolchain.
Refer to the Ninja documentation on the console pool for more info.

A pool is referenced by its label just like a target.
```

Variables

```
depth*
* = required
```

Example

```
if (current_toolchain == default_toolchain) {
  pool("link_pool") {
    depth = 1
  }
}

toolchain("toolchain") {
  tool("link") {
    command = "..."
    pool = ":link_pool($default_toolchain)"
  }
}
```

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 io_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
it 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 "gn 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 only has an effect when run in the context of the
# default toolchain. Pick the right one according to the current CPU
# architecture.
if (target_cpu == "x64") {
  set_default_toolchain("//toolchains:64")
} else if (target_cpu == "x86") {
  set_default_toolchain("//toolchains:32")
}
```

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.
It can be called more than once and the most recent call in any scope will
apply, but there is no way to refer to the previous defaults and modify them
(each call to set_defaults must supply a complete list of all defaults it
wants). If you want to share defaults, store them in a separate variable.
```

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" ]
}
```

split_list: Splits a list into N different sub-lists.

```
result = split_list(input, n)

Given a list and a number N, splits the list into N sub-lists of
approximately equal size. The return value is a list of the sub-lists. The
result will always be a list of size N. If N is greater than the number of
elements in the input, it will be padded with empty lists.

The expected use is to divide source files into smaller uniform chunks.
```

Example

```
The code:
  mylist = [1, 2, 3, 4, 5, 6]
  print(split_list(mylist, 3))

Will print:
  [[1, 2], [3, 4], [5, 6]
```

string_join: Concatenates a list of strings with a separator.

```
result = string_join(separator, strings)

Concatenate a list of strings with intervening occurrences of separator.
```

Examples

```
  string_join("", ["a", "b", "c"])    --> "abc"
  string_join("|", ["a", "b", "c"])   --> "a|b|c"
  string_join(", ", ["a", "b", "c"])  --> "a, b, c"
  string_join("s", ["", ""])          --> "s"
```

string_replace: Replaces substring in the given string.

```
result = string_replace(str, old, new[, max])

Returns a copy of the string str in which the occurrences of old have been
replaced with new, optionally restricting the number of replacements. The
replacement is performed sequentially, so if new contains old, it won't be
replaced.
```

Example

```
The code:
  mystr = "Hello, world!"
  print(string_replace(mystr, "world", "GN"))

Will print:
  Hello, GN!
```

string_split: Split string into a list of strings.

```
result = string_split(str[, sep])

Split string into all substrings separated by separator and returns a list
of the substrings between those separators.

If the separator argument is omitted, the split is by any whitespace, and
any leading/trailing whitespace is ignored; similar to Python's str.split().
```

Examples without a separator (split on whitespace):

```
string_split("")          --> []
string_split("a")         --> ["a"]
string_split(" aa  bb")   --> ["aa", "bb"]
```

Examples with a separator (split on separators):

```
string_split("", "|")           --> [""]
string_split("  a b  ", " ")    --> ["", "", "a", "b", "", ""]
string_split("aa+-bb+-c", "+-") --> ["aa", "bb", "c"]
```

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
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.
```

Overriding builtin targets

```
You can use template to redefine a built-in target in which case your template
takes a precedence over the built-in one. All uses of the target from within
the template definition will refer to the built-in target which makes it
possible to extend the behavior of the built-in target:

  template("shared_library") {
    shared_library(shlib) {
      forward_variables_from(invoker, "*")
      ...
    }
  }
```

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
      "cxx_module": C++ compiler used for Clang .modulemap files
      "objc": Objective C compiler
      "objcxx": Objective C++ compiler
      "rc": Resource compiler (Windows .rc files)
      "asm": Assembler
      "swift": Swift compiler driver

    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.
      "action": Defaults for actions

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

    Rust tools:
      "rust_bin": Tool for compiling Rust binaries
      "rust_cdylib": Tool for compiling C-compatible dynamic libraries.
      "rust_dylib": Tool for compiling Rust dynamic libraries.
      "rust_macro": Tool for compiling Rust procedural macros.
      "rust_rlib": Tool for compiling Rust libraries.
      "rust_staticlib": Tool for compiling Rust static libraries.
```

Tool variables

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

        The command to run.

    command_launcher  [string]
        Valid for: all tools except "action" (optional)

        The prefix with which to launch the command (e.g. the path to a Goma or
        CCache compiler launcher).

        Note that this prefix will not be included in the compilation database
or
        IDE files generated from the build.

    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}}"

  exe_output_extension [string, optional, rust tools only]
  rlib_output_extension [string, optional, rust tools only]
  dylib_output_extension [string, optional, rust tools only]
  cdylib_output_extension [string, optional, rust tools only]
  rust_proc_macro_output_extension [string, optional, rust tools only]
      Valid for: Rust tools

      These specify the default tool output for each of the crate types.
      The default is empty for executables, shared, and static libraries and
      ".rlib" for rlibs. Note that the Rust compiler complains with an error
```

```
    if external crates do not take the form `lib<name>.rlib` or
    `lib<name>.<shared_extension>`, where `<shared_extension>` is `.so`,
    `.dylib`, or `.dll` as appropriate for the platform.

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 to specify
    them.

    If you specified:
      lib_switch = "-l"
      lib_dir_switch = "-L"
    then the "{{libs}}" expansion for
      [ "freetype", "expat" ]
    would be
      "-lfreetype -lexpat".

framework_switch [string, optional, link tools only]
weak_framework_switch [string, optional, link tools only]
framework_dir_switch [string, optional, link tools only]
    Valid for: Linker tools

    These strings will be prepended to the frameworks and framework search
    path directories, respectively, because linkers differ on how to specify
    them.

    If you specified:
      framework_switch = "-framework "
      weak_framework_switch = "-weak_framework "
      framework_dir_switch = "-F"
    and:
      framework_dirs = [ "$root_out_dir" ]
      frameworks = [ "UIKit.framework", "Foo.framework" ]
      weak_frameworks = [ "MediaPlayer.framework" ]
    would be:
      "-F. -framework UIKit -framework Foo -weak_framework MediaPlayer"

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

    The string will be prependend to the path to the .swiftmodule files
    that are embedded in the linker output.

    If you specified:
      swiftmodule_swift = "-Wl,-add_ast_path,"
    then the "{{swiftmodules}}" expansion for
      [ "obj/foo/Foo.swiftmodule" ]
    would be
      "-Wl,-add_ast_path,obj/foo/Foo.swiftmodule"

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_outputs.

      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",
        ]

  partial_outputs  [list of strings with substitutions]
      Valid for: "swift" only

      An array of names for the partial outputs the tool produces. These
      are relative to the build output directory. The expansion will be
      evaluated for each file listed in the "sources" of the target.

      This is used to deal with whole module optimization, allowing to
      list one object file per source file when whole module optimization
      is disabled.

  pool [label, optional]
      Valid for: all tools (optional)

      Label of the pool to use for the tool. Pools are used to limit the
      number of tasks that can execute concurrently during the build.

      See also "gn help pool".

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

      These two 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.

      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 except "action" (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 except "action" (required when "rspfile" is used)

    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}} {{rlibs}}"
        }

  runtime_outputs  [string list with substitutions]
      Valid for: linker tools

      If specified, this list is the subset of the outputs that should be
      added to runtime deps (see "gn help runtime_deps"). By default (if
      runtime_outputs is empty or unspecified), it will be the link_output.

  rust_sysroot
      Valid for: Rust tools

      A path relative to root_out_dir. This is not used in the build
      process, but may be used when generating metadata for rust-analyzer.
      (See --export-rust-project). It enables such metadata to include
      information about the Rust standard library.
```

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.

  {{label_no_toolchain}}
      The label of the current target, never including the toolchain
      (otherwise, this is identical to {{label}}). This is used as the module
      name when using .modulemap files.

  {{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).

  {{module_deps}}
  {{module_deps_no_self}}
      Strings that correspond to the flags necessary to depend upon the Clang
      modules referenced by the current target. The "_no_self" version doesn't
      include the module for the current target, and can be used to compile
      the pcm itself.

  {{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_switch".

      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 dependencies 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"

  {{rlibs}}
```

```
      Any Rust .rlibs which need to be linked into a final C++ target.
      These should be treated as {{inputs}} except that sometimes
      they might have different linker directives applied.

      Example: "obj/foo/libfoo.rlib"

  {{frameworks}}
      Shared libraries packaged as framework bundle. This is principally
      used on Apple's platforms (macOS and iOS). All name must be ending
      with ".framework" suffix; the suffix will be stripped when expanding
      {{frameworks}} and each item will be preceded by "-framework" or
      "-weak_framework".

  {{swiftmodules}}
      Swift .swiftmodule files that needs to be embedded into the binary.
      This is necessary to correctly link with object generated by the
      Swift compiler (the .swiftmodule file cannot be embedded in object
      files directly). Those will be prefixed with "swiftmodule_switch"
      value.

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/macOS 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. The following
substitutions are available:

  {{inputs}}
      Expands to the list of .xcassets to use as input to compile the asset
      catalog.

  {{bundle_product_type}}
      Expands to the product_type of the bundle that will contain the
      compiled asset catalog. Usually corresponds to the product_type
      property of the corresponding create_bundle target.

  {{bundle_partial_info_plist}}
      Expands to the path to the partial Info.plist generated by the
      assets catalog compiler. Usually based on the target_name of
      the create_bundle target.

  {{xcasset_compiler_flags}}
      Expands to the list of flags specified in corresponding
      create_bundle target.

The Swift tool has multiple input and outputs. It must have exactly one
```

```
output of .swiftmodule type, but can have one or more object file outputs,
in addition to other type of outputs. The following expansions are available:

  {{module_name}}
      Expands to the string representing the module name of target under
      compilation (see "module_name" variable).

  {{module_dirs}}
      Expands to the list of -I<path> for the target Swift module search
      path computed from target dependencies.

  {{swiftflags}}
      Expands to the list of strings representing Swift compiler flags.

Rust 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:

  {{crate_name}}
      Expands to the string representing the crate name of target under
      compilation.

  {{crate_type}}
      Expands to the string representing the type of crate for the target
      under compilation.

  {{externs}}
      Expands to the list of --extern flags needed to include addition Rust
      libraries in this target. Includes any specified renamed dependencies.

  {{rustdeps}}
      Expands to the list of -Ldependency=<path> strings needed to compile
      this target.

  {{rustenv}}
      Expands to the list of environment variables.

  {{rustflags}}
      Expands to the list of strings representing Rust compiler flags.
```

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_switch = "-l"
  lib_dir_switch = "-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. The toolchain() function defines these commands.
```

Toolchain overview

```
You can have more than one toolchain in use at once in a build and a target
can exist simultaneously in multiple toolchains. A build file is executed
once for each toolchain it is referenced in so the GN code can vary all
parameters of each target (or which targets exist) on a per-toolchain basis.

When you have a simple build with only one toolchain, the build config file
is loaded only once at the beginning of the build. It must call
set_default_toolchain() (see "gn help set_default_toolchain") to tell GN the
label of the toolchain definition to use. The "toolchain_args" section of the
toolchain definition is ignored.

When a target has a dependency on a target using different toolchain (see "gn
help labels" for how to specify this), GN will start a build using that
secondary toolchain to resolve the target. GN will load the build config file
with the build arguments overridden as specified in the toolchain_args.
```

```
Because the default toolchain is already known, calls to
set_default_toolchain() are ignored.

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.
  3. Loads the destination build file in the context of the configuration file
     in the previous step.

The toolchain configuration is two-way. In the default toolchain (i.e. the
main build target) the configuration flows from the build config file to the
toolchain. The build config file looks at the state of the build (OS type,
CPU architecture, etc.) and decides which toolchain to use (via
set_default_toolchain()). In secondary toolchains, the configuration flows
from the toolchain to the build config file: the "toolchain_args" in the
toolchain definition specifies the arguments to re-invoke the build.
```

Functions and variables

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

toolchain_args [scope]
  Overrides for build arguments to pass to the toolchain when invoking it.
  This is a variable of type "scope" where the variable names correspond to
  variables in declare_args() blocks.

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

  Any default system arguments or arguments passed in via "gn args" 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.

propagates_configs [boolean, default=false]
  Determines whether public_configs and all_dependent_configs in this
  toolchain propagate to targets in other toolchains.

  When false (the default), this toolchain will not propagate any configs to
  targets in other toolchains that depend on it targets inside this
  toolchain. This matches the most common usage of toolchains where they
  represent different architectures or compilers and the settings that apply
  to one won't necessarily apply to others.

  When true, configs (public and all-dependent) will cross the boundary out
  of this toolchain as if the toolchain boundary wasn't there. This only
```

```
    affects one direction of dependencies: a toolchain can't control whether
    it accepts such configs, only whether it pushes them. The build is
    responsible for ensuring that any external targets depending on targets in
    this toolchain are compatible with the compiler flags, etc. that may be
    propagated.

  deps [string list]
    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.
```

Example of defining a toolchain

```
toolchain("32") {
  tool("cc") {
    command = "gcc {{source}}"
    ...
  }

  toolchain_args = {
    use_doom_melon = true  # Doom melon always required for 32-bit builds.
    current_cpu = "x86"
  }
}

toolchain("64") {
  tool("cc") {
    command = "gcc {{source}}"
    ...
  }

  toolchain_args = {
    # use_doom_melon is not overridden here, it will take the default.
    current_cpu = "x64"
  }
}
```

Example of cross-toolchain dependencies

```
If a 64-bit target wants to depend on a 32-bit binary, it would specify a
dependency using data_deps (data deps are like deps that are only needed at
runtime and aren't linked, since you can't link a 32-bit and a 64-bit
library).

  executable("my_program") {
    ...
    if (target_cpu == "x64") {
      # The 64-bit build needs this 32-bit helper.
      data_deps = [ ":helper(//toolchains:32)" ]
```

```
    }
  }

  if (target_cpu == "x86") {
    # Our helper library is only compiled in 32-bits.
    shared_library("helper") {
      ...
    }
  }
```

write_file: Write a file to disk.

```
write_file(filename, data, output_conversion = "")

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 preferable to use response
files for this purpose. See "gn help response_file_contents".
```

Arguments

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

data
    The list or string to write.

output_conversion
  Controls how the output is written. See `gn help io_conversion`.
```

Built-in predefined variables

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 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 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").
```

gn_version: [number] The version of gn.

```
Corresponds to the number printed by `gn --version`.
```

Example

```
assert(gn_version >= 1700, "need GN version 1700 for the frobulate feature")
```

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 overridden
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"
```

invoker: [string] The invoking scope inside a template.

```
Inside a template invocation, this variable refers to the scope of the
invoker of the template. Outside of template invocations, this variable is
undefined.

All of the variables defined inside the template invocation are accessible as
members of the "invoker" scope. This is the way that templates read values
set by the callers.

This is often used with "defined" to see if a value is set on the invoking
scope.

See "gn help template" for more examples.
```

Example

```
template("my_template") {
  print(invoker.sources)       # Prints [ "a.cc", "b.cc" ]
  print(defined(invoker.foo))  # Prints false.
  print(defined(invoker.bar))  # Prints true.
}

my_template("doom_melon") {
  sources = [ "a.cc", "b.cc" ]
  bar = 123
}
```

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, but not necessarily the current 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 does not change based on the current
toolchain. When writing rules, "current_cpu" should be used rather than
"target_cpu" most of the time.

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.
```

Possible values

```
- "x86"
- "x64"
- "arm"
- "arm64"
- "mipsel"
- "mips64el"
- "s390x"
- "ppc64"
- "riscv32"
- "riscv64"
- "e2k"
- "loong64"
```

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_name: [string] The name of the current target.

```
Inside a target or template invocation, this variable refers to the name
given to the target or template invocation. Outside of these, this variable
is undefined.

This is most often used in template definitions to name targets defined in
the template based on the name of the invocation. This is necessary both to
ensure generated targets have unique names and to generate a target with the
exact name of the invocation that other targets can depend on.

Be aware that this value will always reflect the innermost scope. So when
defining a target inside a template, target_name will refer to the target
rather than the template invocation. To get the name of the template
invocation in this case, you should save target_name to a temporary variable
outside of any target definitions.

See "gn help template" for more examples.
```

Example

```
executable("doom_melon") {
  print(target_name)    # Prints "doom_melon".
}

template("my_template") {
  print(target_name)    # Prints "space_ray" when invoked below.

  executable(target_name + "_impl") {
    print(target_name)  # Prints "space_ray_impl".
  }
}

my_template("space_ray") {
}
```

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.
```

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) ]
}
```

Variables you set in targets

aliased_deps: [scope] Set of crate-dependency pairs.

```
Valid for `rust_library` targets and `executable`, `static_library`, and
`shared_library` targets that contain Rust sources.

A scope, each key indicating the renamed crate and the corresponding value
specifying the label of the dependency producing the relevant binary.

All dependencies listed in this field *must* be listed as deps of the target.

  executable("foo") {
    sources = [ "main.rs" ]
    deps = [ "//bar" ]
  }

This target would compile the `foo` crate with the following `extern` flag:
`rustc ...command... --extern bar=<build_out_dir>/obj/bar`

  executable("foo") {
    sources = [ "main.rs" ]
    deps = [ ":bar" ]
    aliased_deps = {
      bar_renamed = ":bar"
    }
  }

With the addition of `aliased_deps`, above target would instead compile with:
`rustc ...command... --extern bar_renamed=<build_out_dir>/obj/bar`
```

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 they 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 occurrence 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 occurrence 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: (target variable) 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 occurrence 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.
  ]
}
```

bridge_header: [string] Path to C/Objective-C compatibility header.

```
Valid for binary targets that contain Swift sources.

Path to an header that includes C/Objective-C functions and types that
needs to be made available to the Swift module.
```

bundle_contents_dir: Expansion of {{bundle_contents_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_contents_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_deps_filter: [label list] A list of labels that are filtered out.

```
A list of target labels.

This list contains target label patterns that should be filtered out when
creating the bundle. Any target matching one of those label will be removed
from the dependencies of the create_bundle target.

This is mostly useful when creating application extension bundle as the
application extension has access to runtime resources from the application
bundle and thus do not require a second copy.

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

Example

```
create_bundle("today_extension") {
  deps = [
    "//base"
  ]
  bundle_root_dir = "$root_out_dir/today_extension.appex"
  bundle_deps_filter = [
    # The extension uses //base but does not use any function calling into
    # third_party/icu and thus does not need the icudtl.dat file.
    "//third_party/icu:icudata",
  ]
}
```

bundle_executable_dir

```
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_resources_dir

```
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_contents_dir}}/Info.plist" ]
}

create_bundle("doom_melon.app") {
  deps = [ ":info_plist" ]
  bundle_root_dir = "${root_build_dir}/doom_melon.app"
  bundle_contents_dir = "${bundle_root_dir}/Contents"
  bundle_resources_dir = "${bundle_contents_dir}/Resources"
  bundle_executable_dir = "${bundle_contents_dir}/MacOS"
}
```

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 and "swiftflags"
for swift 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 occurrence 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 and "swiftflags"
for swift 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 occurrence 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 and "swiftflags"
for swift 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 occurrence 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 and "swiftflags"
for swift 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 occurrence 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 and "swiftflags"
for swift 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 occurrence 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
  ...
}
```

code_signing_args: [string list] Arguments passed to code signing script.

```
For create_bundle targets, code_signing_args is the list of arguments to pass
to the code signing script. Typically you would use source expansion (see "gn
help source_expansion") to insert the source file names.

See also "gn help create_bundle".
```

code_signing_outputs: [file list] Output files for code signing step.

```
Outputs from the code signing step of a create_bundle target. Must refer to
files in the build directory.

See also "gn help create_bundle".
```

code_signing_script: [file name] Script for code signing."

```
An absolute or buildfile-relative file name of a Python script to run for a
create_bundle target to perform code signing step.

See also "gn help create_bundle".
```

code_signing_sources: [file list] Sources for code signing step.

```
A list of files used as input for code signing script step of a create_bundle
target. Non-absolute paths will be resolved relative to the current build
file.

See also "gn help create_bundle".
```

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 occurrence 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" ]
  }
}
```

contents: Contents to write to file.

```
The contents of the file for a generated_file target.
See "gn help generated_file".
```

crate_name: [string] The name for the compiled crate.

```
Valid for `rust_library` targets and `executable`, `static_library`,
`shared_library`, and `source_set` targets that contain Rust sources.

If crate_name is not set, then this rule will use the target name.
```

crate_root: [string] The root source file for a binary or library.

```
Valid for `rust_library` targets and `executable`, `static_library`,
`shared_library`, and `source_set` targets that contain Rust sources.

This file is usually the `main.rs` or `lib.rs` for binaries and libraries,
respectively.

If crate_root is not set, then this rule will look for a lib.rs file (or
main.rs for executable) or a single file in sources, if sources contains
only one file.
```

crate_type: [string] The type of linkage to use on a shared_library.

```
Valid for `rust_library` targets and `executable`, `static_library`,
`shared_library`, and `source_set` targets that contain Rust sources.

Options for this field are "cdylib", "staticlib", "proc-macro", and "dylib".
This field sets the `crate-type` attribute for the `rustc` tool on static
libraries, as well as the appropriate output extension in the
`rust_output_extension` attribute. Since outputs must be explicit, the `lib`
crate type (where the Rust compiler produces what it thinks is the
appropriate library type) is not supported.

It should be noted that the "dylib" crate type in Rust is unstable in the set
of symbols it exposes, and most usages today are potentially wrong and will
be broken in the future.

Static libraries, rust libraries, and executables have this field set
automatically.
```

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.

Note: On iOS and macOS, create_bundle targets will not be recursed into when
gathering data. See "gn help create_bundle" for details.

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.

Note: On iOS and macOS, create_bundle targets will not be recursed into when
gathering data_deps. See "gn help create_bundle" for details.

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

Example

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

data_keys: Keys from which to collect metadata.

```
These keys are used to identify metadata to collect. If a walked target
defines this key in its metadata, its value will be appended to the resulting
collection.

See "gn help generated_file".
```

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 occurrence 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.

A depfile should be used only when a target depends on files that are not
already specified by a target's inputs and sources. Likewise, depfiles should
specify only those dependencies not already included in sources or inputs.

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 paths must be relative to the root
build directory. Only one output may be listed and it must match the first
output of the action.

Although depfiles are created by an action, they should not be listed in the
action's "outputs" unless another target will use the file as an input.
```

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 dependent 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
dependents 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".
```

externs: [scope] Set of Rust crate-dependency pairs.

```
A list, each value being a scope indicating a pair of crate name and the path
to the Rust library.

These libraries will be passed as `--extern crate_name=path` to compiler
invocation containing the current target.
```

Examples

```
  executable("foo") {
    sources = [ "main.rs" ]
    externs = [{
      crate_name = "bar",
      path = "path/to/bar.rlib"
    }]
  }

This target would compile the `foo` crate with the following `extern` flag:
`--extern bar=path/to/bar.rlib`.
```

framework_dirs: [directory list] Additional framework search directories.

```
A list of source directories.

The directories in this list will be added to the framework search 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 occurrence 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

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

frameworks: [name list] Name of frameworks that must be linked.

```
A list of framework names.

The frameworks named in that list will be linked with any dynamic link
type 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 occurrence 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

```
frameworks = [ "Foundation.framework", "Foo.framework" ]
```

friend: Allow targets to include private headers.

```
A list of label patterns (see "gn help label_pattern") that allow dependent
targets to include private headers. Applies to all binary targets.

Normally if a target lists headers in the "public" list (see "gn help
public"), other headers are implicitly marked as private. Private headers
can not be included by other targets, even with a public dependency path.
The "gn check" function performs this validation.

A friend declaration allows one or more targets to include private headers.
This is useful for things like unit tests that are closely associated with a
target and require internal knowledge without opening up all headers to be
included by all dependents.

A friend target does not allow that target to include headers when no
dependency exists. A public dependency path must still exist between two
targets to include any headers from a destination target. The friend
annotation merely allows the use of headers that would otherwise be
prohibited because they are private.

The friend annotation is matched only against the target containing the file
with the include directive. Friend annotations are not propagated across
public or private dependencies. Friend annotations do not affect visibility.
```

Example

```
static_library("lib") {
  # This target can include our private headers.
  friend = [ ":unit_tests" ]

  public = [
    "public_api.h",  # Normal public API for dependent targets.
  ]
```

```
  # Private API and sources.
  sources = [
    "a_source_file.cc",

    # Normal targets that depend on this one won't be able to include this
    # because this target defines a list of "public" headers. Without the
    # "public" list, all headers are implicitly public.
    "private_api.h",
  ]
}

executable("unit_tests") {
  sources = [
    # This can include "private_api.h" from the :lib target because it
    # depends on that target and because of the friend annotation.
    "my_test.cc",
  ]

  deps = [
    ":lib",  # Required for the include to be allowed.
  ]
}
```

gen_deps: Declares targets that should generate when this one does.

```
A list of target labels.

Not all GN targets that get evaluated are actually turned into ninja targets
(see "gn help execution"). If this target is generated, then any targets in
the "gen_deps" list will also be generated, regardless of the usual critera.

Since "gen_deps" are not build time dependencies, there can be cycles between
"deps" and "gen_deps" or within "gen_deps" itself.
```

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 occurrence 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 or
linking the target. 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 implicit dependencies, meaning
that changes in any of the inputs will force all sources in the target to be
recompiled. If an input only applies to a subset of source files, you may
want to split those into a separate target to avoid unnecessary recompiles.
```

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 occurrence 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, they 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 occurrence 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, they 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_switch" 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_switch"
    of the tool.
```

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 occurrence 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" ]
```

metadata: Metadata of this target.

```
Metadata is a collection of keys and values relating to a particular target.
Values must be lists, allowing for sane and predictable collection behavior.
Generally, these keys will include three types of lists: lists of ordinary
strings, lists of filenames intended to be rebased according to their
particular source directory, and lists of target labels intended to be used
as barriers to the walk. Verification of these categories occurs at walk time,
not creation time (since it is not clear until the walk which values are
intended for which purpose).
```

Example

```
group("doom_melon") {
  metadata = {
    # These keys are not built in to GN but are interpreted when consuming
    # metadata.
    my_barrier = []
    my_files = [ "a.txt", "b.txt" ]
  }
}
```

module_name: [string] The name for the compiled module.

```
Valid for binary targets that contain Swift sources.

If module_name is not set, then this rule will use the target name.
```

output_conversion: Data format for generated_file targets.

```
Controls how the "contents" of a generated_file target is formatted.
See `gn help io_conversion`.
```

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 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".
```

partial_info_plist: [filename] Path plist from asset catalog compiler.

```
  Valid for create_bundle target, corresponds to the path for the partial
  Info.plist created by the asset catalog compiler that needs to be merged
  with the application Info.plist (usually done by the code signing script).

  The file will be generated regardless of whether the asset compiler has
  been invoked or not. See "gn help create_bundle".
```

pool: Label of the pool used by the action.

```
  A fully-qualified label representing the pool that will be used for the
  action. Pools are defined using the pool() {...} declaration.
```

Example

```
  action("action") {
    pool = "//build:custom_pool"
    ...
  }
```

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, by setting precompiled_header_type to either "gcc" or "msvc".

  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.

  If you use both C and C++ sources, the precompiled header and source file
  will be compiled once per language. You will want to make sure to wrap C++
  includes in __cplusplus #ifdefs so the file will compile in C mode.
```

GCC precompiled headers

```
When using GCC-style precompiled headers, "precompiled_source" contains the
path of a .h file that is precompiled and then included by all source files
in targets that set "precompiled_source".

The value of "precompiled_header" is not used with GCC-style precompiled
headers.
```

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.

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_header_type: [string] "gcc" or "msvc".

```
See "gn help precompiled_header".
```

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 unless that target is marked as a friend (see "gn help
friend").

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.

It is common for test targets to need to include private headers for their
associated code. In this case, list the test target in the "friend" list of
the target that owns the private header to allow the inclusion. See
"gn help friend" for more.

When a binary target has no explicit or implicit public headers (a "public"
list is defined but is empty), GN assumes that the target can not propagate
any compile-time dependencies up the dependency tree. In this case, the build
can be parallelized more efficiently.
Say there are dependencies:
  A (shared library) -> B (shared library) -> C (action).
Normally C must complete before any source files in A can compile (because
there might be generated includes). But when B explicitly declares no public
headers, C can execute in parallel with A's compile steps. C must still be
complete before any dependents link.
```

Examples

```
These exact files are public:
  public = [ "foo.h", "bar.h" ]

No files are public (no targets may include headers from this one):
  # This allows starting compilation in dependent targets earlier.
  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.
Generally, public configs are used to apply defines and include directories
necessary to compile this target's header files.

See also "gn help all_dependent_configs".
```

Propagation of public configs

```
Public configs are applied to all targets that depend directly on this one.
These dependent targets can further push this target's public configs
higher in the dependency tree by depending on it via public_deps (see "gn
help public_deps").

  static_library("toplevel") {
    # This target will get "my_config" applied to it. However, since this
    # target uses "deps" and not "public_deps", targets that depend on this
    # one won't get it.
    deps = [ ":intermediate" ]
  }

  static_library("intermediate") {
    # Depending on "lower" in any way will apply "my_config" to this target.
    # Additionall, since this target depends on "lower" via public_deps,
    # targets that depend on this one will also get "my_config".
    public_deps = [ ":lower" ]
  }

  static_library("lower") {
    # This will get applied to all targets that depend on this one.
    public_configs = [ ":my_config" ]
  }

Public config propagation 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 they can not be removed. As a result, this capability should generally
only be used to add defines and include directories rather than setting
complicated flags that some targets may not want.

Public configs may or may not be propagated across toolchain boundaries
depending on the value of the propagates_configs flag (see "gn help
```

```
toolchain") on the toolchain of the target declaring the public_config.
```

Avoiding applying public configs to this target

```
If you want the config to apply to targets that depend on this one, but NOT
this one, define an extra layer of indirection using a group:

  # External targets depend on this group.
  group("my_target") {
    # Config to apply to all targets that depend on this one.
    public_configs = [ ":external_settings" ]
    deps = [ ":internal_target" ]
  }

  # Internal target to actually compile the sources.
  static_library("internal_target") {
    # Force all external targets to depend on the group instead of directly
    # on this so the "external_settings" config will get applied.
    visibility = [ ":my_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 occurrence 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.
```

```
See also "gn help public_configs".
```

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" ]
}
```

rebase: Rebase collected metadata as files.

```
A boolean that triggers a rebase of collected metadata strings based on their
declared file. Defaults to false.

Metadata generally declares files as strings relative to the local build file.
However, this data is often used in other contexts, and so setting this flag
will force the metadata collection to be rebased according to the local build
file's location and thus allow the filename to be used anywhere.

Setting this flag will raise an error if any target's specified metadata is
not a string value.

See also "gn help generated_file".
```

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 existence 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
preceding "/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).

For Rust targets that do not specify a crate_root, then the crate_root will
```

```
look for a lib.rs file (or main.rs for executable) or a single file in
sources, if sources contains only one file.
```

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.
```

swiftflags: Flags passed to the swift compiler.

```
A list of strings.

"swiftflags" are passed to any invocation of a tool that takes an .swift
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 occurrence 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.
```

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/*" ]
```

walk_keys: Key(s) for managing the metadata collection walk.

```
  Defaults to [""].

  These keys are used to control the next step in a collection walk, acting as
  barriers. If a specified key is defined in a target's metadata, the walk will
  use the targets listed in that value to determine which targets are walked.

  If no walk_keys are specified for a generated_file target (i.e. "[""]"), the
  walk will touch all deps and data_deps of the specified target recursively.

  See "gn help generated_file".
```

weak_frameworks: [name list] Name of frameworks that must be weak linked.

```
  A list of framework names.

  The frameworks named in that list will be weak linked with any dynamic link
  type target. Weak linking instructs the dynamic loader to attempt to load
  the framework, but if it is not able to do so, it leaves any imported symbols
  unresolved. This is typically used when a framework is present in a new
  version of an SDK but not on older versions of the OS that the software runs
  on.
```

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 occurrence 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

```
  weak_frameworks = [ "OnlyOnNewerOSes.framework" ]
```

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").
```

xcasset_compiler_flags: Flags passed to xcassets compiler.

```
A list of strings.

Valid for create_bundle target. Those flags are directly passed to
xcassets compiler, corresponding to {{xcasset_compiler_flags}} substitution
in compile_xcassets tool.
```

xcode_extra_attributes: [scope] Extra attributes for Xcode projects.

```
The value defined in this scope will be copied to the EXTRA_ATTRIBUTES
property of the generated Xcode project. They are only meaningful when
generating with --ide=xcode.

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

xcode_test_application_name: Name for Xcode test target.

```
Each unit and ui test target must have a test application target, and this
value is used to specify the relationship. Only meaningful to Xcode (used as
part of the Xcode project generation).

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

Example

```
create_bundle("chrome_xctest") {
  test_application_name = "chrome"
  ...
}
```

Other help topics

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

Next, project-specific overrides are applied. These are specified inside
the default_args variable of //.gn. See "gn help dotfile" for more.

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" 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

```
arg_file_template [optional]
    Path to a file containing the text that should be used as the default
    args.gn content when you run `gn args`.

buildconfig [required]
    Path to 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 neither check_targets or
    no_check_targets (see below) is specified, all targets will be checked.
    It is an error to specify both check_targets and no_check_targets. If it
    is the empty list, no targets will be checked. To bypass this list,
    request an explicit check of targets, like "//*".

    The format of this list is identical to that of "visibility" so see "gn
    help visibility" for examples.

no_check_targets [optional]
    A list of labels and label patterns that should *not* be checked when
    running "gn check" or "gn gen --check". All other targets will be checked.
    If neither check_targets (see above) or no_check_targets is specified, all
    targets will be checked. It is an error to specify both check_targets and
    no_check_targets.

    The format of this list is identical to that of "visibility" so see "gn
    help visibility" for examples.
```

```
check_system_includes [optional]
    Boolean to control whether system style includes are checked by default
    when running "gn check" or "gn gen --check".  System style includes are
    includes that use angle brackets <> instead of double quotes "". If this
    setting is omitted or set to false, these includes will be ignored by
    default. They can be checked explicitly by running
    "gn check --check-system" or "gn gen --check=system"

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. Note that build_file_extension
    applies to the default case as well.

    The command-line switch --root-target will override this value (see "gn
    help --root-target").

script_executable [optional]
    Path to specific Python executable or other interpreter to use in
    action targets and exec_script calls. By default GN searches the
    PATH for Python to execute these scripts.

    If set to the empty string, the path specified in action targets
    and exec_script calls will be executed directly.

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.

default_args [optional]
```

```
    Scope containing the default overrides for declared arguments. These
    overrides take precedence over the default values specified in the
    declare_args() block, but can be overridden using --args or the
    args.gn file.

    This is intended to be used when subprojects declare arguments with
    default values that need to be changed for whatever reason.

build_file_extension [optional]
    If set to a non-empty string, this is added to the name of all build files
    to load.
    GN will look for build files named "BUILD.$build_file_extension.gn".
    This is intended to be used during migrations or other situations where
    there are two independent GN builds in the same directories.

ninja_required_version [optional]
    When set specifies the minimum required version of Ninja. The default
    required version is 1.7.2. Specifying a higher version might enable the
    use of some of newer features that can make the build more efficient.
```

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/"

default_args = {
  # Default to release builds for this project.
  is_debug = false
  is_component_build = false
}
```

Build graph and execution overview

Overall build flow

```
1. Look for ".gn" file (see "gn help dotfile") in the current directory and
   walk up the directory tree until one is found. Set this directory to be
   the "source root" and interpret this file to find the name of the build
   config file.

2. Execute the build config file identified by .gn to set up the global
   variables and default toolchain name. Any arguments, variables, defaults,
   etc. set up in this file will be visible to all files in the build.

3. Load the //BUILD.gn (in the source root directory).

4. Recursively evaluate rules and load BUILD.gn in other directories as
   necessary to resolve dependencies. If a BUILD file isn't found in the
```

```
   specified location, GN will look in the corresponding location inside
   the secondary_source defined in the dotfile (see "gn help dotfile").

5. When a target's dependencies are resolved, write out the `.ninja`
   file to disk.

6. When all targets are resolved, write out the root build.ninja file.

Note that the BUILD.gn file name may be modulated by .gn arguments such as
build_file_extension.
```

Executing target definitions and templates

```
Build files are loaded in parallel. This means it is impossible to
interrogate a target from GN code for any information not derivable from its
label (see "gn help label"). The exception is the get_target_outputs()
function which requires the target being interrogated to have been defined
previously in the same file.

Targets are declared by their type and given a name:

  static_library("my_static_library") {
    ... target parameter definitions ...
  }

There is also a generic "target" function for programmatically defined types
(see "gn help target"). You can define new types using templates (see "gn
help template"). A template defines some custom code that expands to one or
more other targets.

Before executing the code inside the target's { }, the target defaults are
applied (see "gn help set_defaults"). It will inject implicit variable
definitions that can be overridden by the target code as necessary. Typically
this mechanism is used to inject a default set of configs that define the
global compiler and linker flags.
```

Which targets are built

```
All targets encountered in the default toolchain (see "gn help toolchain")
will have build rules generated for them, even if no other targets reference
them. Their dependencies must resolve and they will be added to the implicit
"all" rule (see "gn help ninja_rules").

Targets in non-default toolchains will only be generated when they are
required (directly or transitively) to build a target in the default
toolchain.

Some targets might be associated but without a formal build dependency (for
example, related tools or optional variants). A target that is marked as
"generated" can propagate its generated state to an associated target using
"gen_deps". This will make the referenced dependency have Ninja rules
generated in the same cases the source target has but without a build-time
dependency and even in non-default toolchains.

See also "gn help ninja_rules".
```

Dependencies

```
The only difference between "public_deps" and "deps" except for pushing
configs around the build tree and allowing includes for the purposes of "gn
check".

A target's "data_deps" are guaranteed to be built whenever the target is
built, but the ordering is not defined. The meaning of this is dependencies
required at runtime. Currently data deps will be complete before the target
is linked, but this is not semantically guaranteed and this is undesirable
from a build performance perspective. Since we hope to change this in the
future, do not rely on this behavior.
```

Language and grammar for GN build files

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".

An expansion will evaluate the variable following the '$' and insert a
stringified version of it into the result. For example, to concat two path
components with a slash separating them:
  "$var_one/$var_two"
Use the "${var_one}" format to be explicitly deliniate the variable for
otherwise-ambiguous cases.
```

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 .
    LValue        = identifier | ArrayAccess | ScopeAccess .
    Assignment    = LValue AssignOp Expr .
    Call          = identifier "(" [ ExprList ] ")" [ Block ] .
    Condition     = "if" "(" Expr ")" Block
                    [ "else" ( Condition | Block ) ] .
    Block         = "{" StatementList "}" .
    StatementList = { Statement } .

    ArrayAccess = identifier "[" Expr "]" .
    ScopeAccess = identifier "." identifier .
```

```
    Expr       = UnaryExpr | Expr BinaryOp Expr .
    UnaryExpr  = PrimaryExpr | UnaryOp UnaryExpr .
    PrimaryExpr = identifier | integer | string | Call
                | ArrayAccess | ScopeAccess | Block
                | "(" Expr ")"
                | "[" [ ExprList [ "," ] ] "]" .
    ExprList   = Expr { "," Expr } .

    AssignOp = "=" | "+=" | "-=" .
    UnaryOp  = "!" .
    BinaryOp = "+" | "-"                  // highest priority
             | "<" | "<=" | ">" | ">="
             | "==" | "!="
             | "&&"
             | "||" .                     // lowest priority

All binary operators are left-associative.
```

Types

```
The GN language is dynamically typed. The following types are used:

 - Boolean: Uses the keywords "true" and "false". There is no implicit
   conversion between booleans and integers.

 - Integers: All numbers in GN are signed 64-bit integers.

 - Strings: Strings are 8-bit with no enforced encoding. When a string is
   used to interact with other systems with particular encodings (like the
   Windows and Mac filesystems) it is assumed to be UTF-8. See "String
   literals" above for more.

 - Lists: Lists are arbitrary-length ordered lists of values. See "Lists"
   below for more.

 - Scopes: Scopes are like dictionaries that use variable names for keys. See
   "Scopes" below for more.
```

Lists

```
Lists are created with [] and using commas to separate items:

     mylist = [ 0, 1, 2, "some string" ]

A comma after the last item is optional. Lists are dereferenced using 0-based
indexing:

     mylist[0] += 1
     var = mylist[2]

Lists can be concatenated using the '+' and '+=' operators. Bare values can
not be concatenated with lists, to add a single item, it must be put into a
list of length one.

Items can be removed from lists using the '-' and '-=' operators. This will
remove all occurrences of every item in the right-hand list from the
```

```
left-hand list. It is an error to remove an item not in the list. This is to
prevent common typos and to detect dead code that is removing things that no
longer apply.

It is an error to use '=' to replace a nonempty list with another nonempty
list. This is to prevent accidentally overwriting data when in most cases
'+=' was intended. To overwrite a list on purpose, first assign it to the
empty list:

  mylist = []
  mylist = otherlist
```

Scopes

```
All execution happens in the context of a scope which holds the current state
(like variables). With the exception of loops and conditions, '{' introduces
a new scope that has a parent reference to the old scope.

Variable reads recursively search all nested scopes until the variable is
found or there are no more scopes. Variable writes always go into the current
scope. This means that after the closing '}' (again excepting loops and
conditions), all local variables will be restored to the previous values.
This also means that "foo = foo" can do useful work by copying a variable
into the current scope that was defined in a containing scope.

Scopes can also be assigned to variables. Such scopes can be created by
functions like exec_script, when invoking a template (the template code
refers to the variables set by the invoking code by the implicitly-created
"invoker" scope), or explicitly like:

  empty_scope = {}
  myvalues = {
    foo = 21
    bar = "something"
  }

Inside such a scope definition can be any GN code including conditionals and
function calls. After the close of the scope, it will contain all variables
explicitly set by the code contained inside it. After this, the values can be
read, modified, or added to:

  myvalues.foo += 2
  empty_scope.new_thing = [ 1, 2, 3 ]

Scope equality is defined as single-level scopes identical within the current
scope. That is, all values in the first scope must be present and identical
within the second, and vice versa. Note that this means inherited scopes are
always unequal by definition.
```

Input and output conversion

```
Input and output conversions are arguments to file and process functions
that specify how to convert data to or from external formats. The possible
values for parameters specifying conversions are:

"" (the default)
```

```
      input: Discard the result and return None.

      output: If value is a list, then "list lines"; otherwise "value".

  "list lines"
      input:
        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.

      output:
        Renders the value contents as a list, with a string for each line. The
        newlines will not be present in the result. The last line will end in
        with a newline.

  "scope"
      input:
        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

      output:
        Renders the value contents as a GN code block, reversing the input
        result above.

  "string"
      input: Return the file contents into a single string.

      output:
        Render the value contents into a single string. The output is:
        a string renders with quotes, e.g. "str"
        an integer renders as a stringified integer, e.g. "6"
        a boolean renders as the associated string, e.g. "true"
        a list renders as a representation of its contents, e.g. "[\"str\", 6]"
        a scope renders as a GN code block of its values. If the Value was:
            Value val;
            val.a = [ "hello.cc", "world.cc" ];
            val.b = 26
          the resulting output would be:
            "{
                a = [ \"hello.cc\", \"world.cc\" ]
                b = 26
            }"

  "value"
      input:
        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.

    output:
      Render the value contents as a literal rvalue. Strings render with
      escaped quotes.

"json"
    input: Parse the input as a JSON and convert it to equivalent GN rvalue.

    output: Convert the Value to equivalent JSON value.

    The data type mapping is:
      a string in JSON maps to string in GN
      an integer in JSON maps to integer in GN
      a float in JSON is unsupported and will result in an error
      an object in JSON maps to scope in GN
      an array in JSON maps to list in GN
      a boolean in JSON maps to boolean in GN
      a null in JSON is unsupported and will result in an error

    Nota that the input dictionary keys have to be valid GN identifiers
    otherwise they will produce an error.

"trim ..." (input only)
    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.
```

File 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".
```

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 parenthesis at the end of the label pattern. In this case, the
toolchain must be fully qualified (no wildcards are supported in the
toolchain name).

  "//foo:bar(//build/toolchain: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.
```

About labels

```
Everything that can participate in the dependency graph (targets, configs,
and toolchains) are identified by labels. A common label looks like:

  //base/test:test_support
```

```
  This consists of a source-root-absolute path, a colon, and a name. This means
  to look for the thing named "test_support" in "base/test/BUILD.gn".

  You can also specify system absolute paths if necessary. Typically such
  paths would be specified via a build arg so the developer can specify where
  the component is on their system.

    /usr/local/foo:bar    (Posix)
    /C:/Program Files/MyLibs:bar   (Windows)
```

Toolchains

```
  A canonical label includes the label of the toolchain being used. Normally,
  the toolchain label is implicitly inherited from the current execution
  context, but you can override this to specify cross-toolchain dependencies:

    //base/test:test_support(//build/toolchain/win:msvc)

  Here GN will look for the toolchain definition called "msvc" in the file
  "//build/toolchain/win" to know how to compile this target.
```

Relative labels

```
  If you want to refer to something in the same buildfile, you can omit
  the path name and just start with a colon. This format is recommended for
  all same-file references.

    :base

  Labels can be specified as being relative to the current directory.
  Stylistically, we prefer to use absolute paths for all non-file-local
  references unless a build file needs to be run in different contexts (like a
  project needs to be both standalone and pulled into other projects in
  difference places in the directory hierarchy).

    source/plugin:myplugin
    ../net:url_request
```

Implicit names

```
  If a name is unspecified, it will inherit the directory name. Stylistically,
  we prefer to omit the colon and name when possible:

    //net  ->  //net:net
    //tools/gn  ->  //tools/gn:gn
```

Metadata Collection

```
  Metadata is information attached to targets throughout the dependency tree. GN
  allows for the collection of this data into files written during the
generation
  step, enabling users to expose and aggregate this data based on the dependency
  tree.
```

generated_file targets

```
Similar to the write_file() function, the generated_file target type
creates a file in the specified location with the specified content. The
primary difference between write_file() and this target type is that the
write_file function does the file write at parse time, while the
generated_file target type writes at target resolution time. See
"gn help generated_file" for more detail.

When written at target resolution time, generated_file enables GN to
collect and write aggregated metadata from dependents.

A generated_file target can declare either 'contents' to write statically
known contents to a file or 'data_keys' to aggregate metadata and write the
result to a file. It can also specify 'walk_keys' (to restrict the metadata
collection), 'output_conversion', and 'rebase'.
```

Collection and Aggregation

```
Targets can declare a 'metadata' variable containing a scope, and this
metadata may be collected and written out to a file specified by
generated_file aggregation targets. The 'metadata' scope must contain
only list values since the aggregation step collects a list of these values.

During the target resolution, generated_file targets will walk their
dependencies recursively, collecting metadata based on the specified
'data_keys'. 'data_keys' is specified as a list of strings, used by the walk
to identify which variables in dependencies' 'metadata' scopes to collect.

The walk begins with the listed dependencies of the 'generated_file' target.
The 'metadata' scope for each dependency is inspected for matching elements
of the 'generated_file' target's 'data_keys' list.  If a match is found, the
data from the dependent's matching key list is appended to the aggregate walk
list. Note that this means that if more than one walk key is specified, the
data in all of them will be aggregated into one list. From there, the walk
will then recurse into the dependencies of each target it encounters,
collecting the specified metadata for each.

For example:

  group("a") {
    metadata = {
      doom_melon = [ "enable" ]
      my_files = [ "foo.cpp" ]
      my_extra_files = [ "bar.cpp" ]
    }

    deps = [ ":b" ]
  }

  group("b") {
    metadata = {
      my_files = [ "baz.cpp" ]
    }
  }
```

```
  generated_file("metadata") {
    outputs = [ "$root_build_dir/my_files.json" ]
    data_keys = [ "my_files", "my_extra_files" ]

    deps = [ ":a" ]
  }

The above will produce the following file data:

  foo.cpp
  bar.cpp
  baz.cpp

The dependency walk can be limited by using the 'walk_keys'. This is a list of
labels that should be included in the walk. All labels specified here should
also be in one of the deps lists. These keys act as barriers, where the walk
will only recurse into the targets listed. An empty list in all specified
barriers will end that portion of the walk.

  group("a") {
    metadata = {
      my_files = [ "foo.cpp" ]
      my_files_barrier = [ ":b" ]
    }

    deps = [ ":b", ":c" ]
  }

  group("b") {
    metadata = {
      my_files = [ "bar.cpp" ]
    }
  }

  group("c") {
    metadata = {
      my_files = [ "doom_melon.cpp" ]
    }
  }

  generated_file("metadata") {
    outputs = [ "$root_build_dir/my_files.json" ]
    data_keys = [ "my_files" ]
    walk_keys = [ "my_files_barrier" ]

    deps = [ ":a" ]
  }

The above will produce the following file data (note that `doom_melon.cpp` is
not included):

  foo.cpp
  bar.cpp

A common example of this sort of barrier is in builds that have host tools
built as part of the tree, but do not want the metadata from those host tools
to be collected with the target-side code.
```

Common Uses

```
Metadata can be used to collect information about the different targets in the
build, and so a common use is to provide post-build tooling with a set of data
necessary to do aggregation tasks. For example, if each test target specifies
the output location of its binary to run in a metadata field, that can be
collected into a single file listing the locations of all tests in the
dependency tree. A local build tool (or continuous integration infrastructure)
can then use that file to know which tests exist, and where, and run them
accordingly.

Another use is in image creation, where a post-build image tool needs to know
various pieces of information about the components it should include in order
to put together the correct image.
```

Ninja build rules

The "all" and "default" rules

```
All generated targets (see "gn help execution") will be added to an implicit
build rule called "all" so "ninja all" will always compile everything. The
default rule will be used by Ninja if no specific target is specified (just
typing "ninja"). If there is a target named "default" in the root build file,
it will be the default build rule, otherwise the implicit "all" rule will be
used.
```

Phony rules

```
GN generates Ninja "phony" rules for targets in the default toolchain.  The
phony rules can collide with each other and with the names of generated files
so are generated with the following priority:

  1. Actual files generated by the build always take precedence.

  2. Targets in the toplevel //BUILD.gn file.

  3. Targets in toplevel directories matching the names of the directories.
     So "ninja foo" can be used to compile "//foo:foo". This only applies to
     the first level of directories since usually these are the most
     important (so this won't apply to "//foo/bar:bar").

  4. The short names of executables if there is only one executable with that
     short name. Use "ninja doom_melon" to compile the
     "//tools/fruit:doom_melon" executable.

  5. The short names of all targets if there is only one target with that
     short name.

  6. Full label name with no leading slashes. So you can use
     "ninja tools/fruit:doom_melon" to build "//tools/fruit:doom_melon".

  7. Labels with an implicit name part (when the short names match the
     directory). So you can use "ninja foo/bar" to compile "//foo/bar:bar".

These "phony" rules are provided only for running Ninja since this matches
```

```
people's historical expectations for building. For consistency with the rest
of the program, GN introspection commands accept explicit labels.

To explicitly compile a target in a non-default toolchain, you must give
Ninja the exact name of the output file relative to the build directory.
```

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 its 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

```
Linker tools can specify which of their outputs should be considered when
computing the runtime deps by setting runtime_outputs. If this is unset on
the tool, the default will be the first output only.
```

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"

{{source_target_relative}}
    The path to the source file relative to the target's directory. This will
    generally be used for replicating the source directory layout in the
    output directory. This can only be used in actions and bundle_data
    targets. It is an error to use in process_file_template where there is no
    "target".
      "//foo/bar/baz.txt" => "baz.txt"
```

(*) 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.
  *   --ninja-executable: Set the Ninja executable.
  *   --nocolor: Force non-colored output.
  *   -q: Quiet mode. Don't print output on success.
  *   --root: Explicitly specify source root.
  *   --root-target: Override the root target.
  *   --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.
```