GN Style Guide

Contents

Naming and ordering within the file

Location of build files

It usually makes sense to have more build files closer to the code than fewer ones at the toplevel (this is in contrast with what we did with GYP). This makes things easier to find and owners reviews easier since changes are more focused.

Targets

- Most BUILD files should have a target with the same name of the directory. This target should be the first target.
- Other targets should be in “some order that makes sense.” Usually more important targets will be first, and unit tests will follow the corresponding target. If there’s no clear ordering, consider alphabetical order.
- Test support libraries should be source sets named “test_support”. So “//ui/compositor:test_support”. Test support libraries should include as public deps the non-test-support version of the library so tests need only depend on the test_support target (rather than both).

Naming advice

- Targets and configs should be named using lowercase with underscores separating words, unless there is a strong reason to do otherwise.
- Source sets, groups, and static libraries do not need globally unique names. Prefer to give such targets short, non-redundant names without worrying about global uniqueness. For example, it looks much better to write a dependency as `"//mojo/public/bindings"` rather than `"//mojo/public/bindings:mojo_bindings"
- Shared libraries (and by extension, components) must have globally unique output names. Give such targets short non-unique names above, and then provide a globally unique `output_name` for that target.
- Executables and tests should be given a globally unique name. Technically only the output names must be unique, but since only the output names appear in the shell and on bots, it's much less confusing if the name matches the other places the executable appears.

Configs

- A config associated with a single target should be named the same as the target with `_config` following it.
- A config should appear immediately before the corresponding target that uses it.

Example

Example for the `src/foo/BUILD.gn` file:

```
# Copyright 2013 The Chromium Authors. All rights reserved.
# Use of this source code is governed by a BSD-style license that can be
# found in the LICENSE file.

# Config for foo is named foo_config and immediately precedes it in the file.
config("foo_config") {
}

# Target matching path name is the first target.
executable("foo") {
}

# Test for foo follows it.
test("foo_unittests") {
}

config("bar_config") {
}

source_set("bar") {
}
```

Ordering within a target

1. `output_name` / `visibility` / `testonly`
2. `sources`
3. `cflags`, `include_dirs`, `defines`, `configs` etc. in whatever order makes sense to you.
4. `public_deps`
5. `deps`

Conditions

Simple conditions affecting just one variable (e.g. adding a single source or adding a flag for one particular OS) can go beneath the variable they affect. More complicated conditions affecting more than one thing should go at the bottom.

Conditions should be written to minimize the number of conditional blocks.

Formatting and indenting

GN contains a built-in code formatter which defines the formatting style. Some additional notes:

- Variables are `lower_case_with_underscores`
- Comments should be complete sentences with periods at the end.
- Compiler flags and such should always be commented with what they do and why the flag is needed.

Sources

Prefer to list sources only once. It is OK to conditionally include sources rather than listing them all at the top and then conditionally excluding them when they don’t apply. Conditional inclusion is often clearer since a file is only listed once and it’s easier to reason about when reading.

```
sources = [
  "main.cc",
]
if (use_aura) {
  sources += [ "thing_aura.cc" ]
}
if (use_gtk) {
  sources += [ "thing_gtk.cc" ]
}
```

Deps

- Deps should be in alphabetical order.
- Deps within the current file should be written first and not qualified with the file name (just `:foo`).

- Other deps should always use fully-qualified path names unless relative ones are required for some reason.

```
deps = [
  ":a_thing",
  ":mystatic",
  "//foo/bar:other_thing",
  "//foo/baz:that_thing",
]
```

Import

Use fully-qualified paths for imports:

```
import("//foo/bar/baz.gni")  # Even if this file is in the foo/bar directory
```

Usage

Use `source_set` rather than `static_library` unless you have a reason to do otherwise. A static library is a standalone library which can be slow to generate. A source set just links all the object files from that target into the targets depending on it, which saves the “lib” step.

Build arguments

Scope

Build arguments should be scoped to a unit of behavior, e.g. enabling a feature. Typically an argument would be declared in an imported file to share it with the subset of the build that could make use of it.

Chrome has many legacy flags in `//build/config/features.gni`, `//build/config/ui.gni`. These locations are deprecated. Feature flags should go along with the code for the feature. Many browser-level features can go somewhere in `//chrome/` without lower-level code knowing about it. Some UI environment flags can go into `//ui/`, and many flags can also go with the corresponding code in `//components/`. You can write a `.gni` file in components and have build files in chrome or content import it if necessary.

The way to think about things in the `//build` directory is that this is DEPSed into various projects like V8 and WebRTC. Build flags specific to code outside of the build directory shouldn’t be in the build directory, and V8 shouldn’t get feature defines for Chrome features.

New feature defines should use the buildflag system. See `//build/buildflag_header.gni` which allows preprocessor defines to be modularized without many of the disadvantages that made us use global defines in the past.

Type

Arguments support all the GN language types.

In the vast majority of cases `boolean` is the preferred type, since most arguments are enabling or disabling features or includes.

`String` s are typically used for filepaths. They are also used for enumerated types, though `integer` s are sometimes used as well.

Naming conventions

While there are no hard and fast rules around argument naming there are many common conventions. If you ever want to see the current list of argument names and default values for your current checkout use `gn args out/Debug --list --short` .

`use_foo` - indicates dependencies or major codepaths to include (e.g. `use_open_ssl` , `use_ozone` , `use_cups`)

`enable_foo` - indicates feature or tools to be enabled (e.g. `enable_google_now` , `enable_nacl` , `enable_remoting` , `enable_pdf`)

`disable_foo` - *NOT* recommended, use `enable_foo` instead with swapped default value

`is_foo` - usually a global state descriptor (e.g. `is_chrome_branded` , `is_desktop_linux`); poor choice for non-globals

`foo_use_bar` - prefixes can be used to indicate a limited scope for an argument (e.g. `rtc_use_h264` , `v8_use_snapshot`)

Powered by Gitiles source log blame