# GN Quick Start guide

## Contents

## Running GN

You just run `gn` from the command line. There is a script in depot_tools (which is presumably on your path) with this name. The script will find the binary in the source tree containing the current directory and run it.

## Setting up a build

In GYP, the system would generate `Debug` and `Release` build directories for you and configure them accordingly. GN doesn't do this. Instead, you set up whatever build directory you want with whatever configuration you want. The Ninja files will be automatically regenerated if they're out of date when you build in that directory.

To make a build directory:

```
gn gen out/my_build
```

## Passing build arguments

Set build arguments on your build directory by running:

```
gn args out/my_build
```

This will bring up an editor. Type build args into that file like this:

```
is_component_build = true
is_debug = false
```

You can see the list of available arguments and their default values by typing

```
gn args --list out/my_build
```

on the command line. See "Taking build arguments" below for information on how to use these in your code. (Note that you have to specify the build directory for this command because the available arguments can change according to what's set.

Chrome developers can also read the Chrome-specific build configuration instructions for more information.

## Cross-compiling to a target OS or architecture

Run `gn args out/Default` (substituting your build directory as needed) and add one or more of the following lines for common cross-compiling options.

```
target_os = "chromeos"
target_os = "android"

target_cpu = "arm"
target_cpu = "x86"
target_cpu = "x64"
```

See GNCrossCompiles for more info.

## Configuring goma

Run `gn args out/Default` (substituting your build directory as needed). Add:

```
use_goma = true
goma_dir = "~/foo/bar/goma"
```

If your goma is in the default location ( `~/goma` ) then you can omit the `goma_dir` line.

## Configuring component mode

This is a build arg like the goma flags. run `gn args out/Default` and add:

```
is_component_build = true
```

## Step-by-step

### Adding a build file

Create a `tools/gn/tutorial/BUILD.gn` file and enter the following:

```
executable("hello_world") {
  sources = [
    "hello_world.cc",
  ]
}
```

There should already be a `hello_world.cc` file in that directory, containing what you expect. That's it! Now we just need to tell the build about this file. Open the `BUILD.gn` file in the root directory and add the label of this target to the dependencies of one of the root groups (a "group" target is a meta-target that is just a collection of other targets):

```
group("root") {
  deps = [
    ...
    "//url",
    "//tools/gn/tutorial:hello_world",
  ]
}
```

You can see the label of your target is "//" (indicating the source root), followed by the directory name, a colon, and the target name.

### Testing your addition

From the command line in the source root directory:

```
gn gen out/Default
ninja -C out/Default hello_world
out/Default/hello_world
```

GN encourages target names for static libraries that aren't globally unique. To build one of these, you can pass the label with no leading "//" to ninja:

```
ninja -C out/Default tools/gn/tutorial:hello_world
```

## Declaring dependencies

Let’s make a static library that has a function to say hello to random people. There is a source file `hello.cc` in that directory which has a function to do this. Open the `tools/gn/tutorial/BUILD.gn` file and add the static library to the bottom of the existing file:

```
static_library("hello") {
  sources = [
    "hello.cc",
  ]
}
```

Now let’s add an executable that depends on this library:

```
executable("say_hello") {
  sources = [
    "say_hello.cc",
  ]
  deps = [
    ":hello",
  ]
}
```

This executable includes one source file and depends on the previous static library. The static library is referenced by its label in the `deps`. You could have used the full label `//tools/gn/tutorial:hello` but if you’re referencing a target in the same build file, you can use the shortcut `:hello`.

## Test the static library version

From the command line in the source root directory:

```
ninja -C out/Default say_hello
out/Default/say_hello
```

Note that you **didn’t** need to re-run GN. GN will automatically rebuild the ninja files when any build file has changed. You know this happens when ninja prints `[1/1] Regenerating ninja files` at the beginning of execution.

## Compiler settings

Our hello library has a new feature, the ability to say hello to two people at once. This feature is controlled by defining `TWO_PEOPLE`. We can add defines like so:

```
static_library("hello") {
  sources = [
    "hello.cc",
  ]
  defines = [
    "TWO_PEOPLE",
  ]
}
```

## Putting settings in a config

However, users of the library also need to know about this define, and putting it in the static library target defines it only for the files there. If somebody else includes `hello.h`, they won't see the new definition. To see the new definition, everybody will have to define `TWO_PEOPLE`.

GN has a concept called a "config" which encapsulates settings. Let's create one that defines our preprocessor define:

```
config("hello_config") {
  defines = [
    "TWO_PEOPLE",
  ]
}
```

To apply these settings to your target, you only need to add the config's label to the list of configs in the target:

```
static_library("hello") {
  ...
  configs += [
    ":hello_config",
  ]
}
```

Note that you need "+=" here instead of "=" since the build configuration has a default set of configs applied to each target that set up the default build stuff. You want to add to this list rather than overwrite it. To see the default configs, you can use the `print` function in the build file or the `desc` command-line subcommand (see below for examples of both).

## Dependent configs

This nicely encapsulates our settings, but still requires everybody that uses our library to set the config on themselves. It would be nice if everybody that depends on our `hello` library can get this automatically. Change your library definition to:

```
static_library("hello") {
  sources = [
    "hello.cc",
  ]
  all_dependent_configs = [
    ":hello_config"
  ]
}
```

This applies the `hello_config` to the `hello` target itself, plus all targets that transitively depend on the current one. Now everybody that depends on us will get our settings. You can also set `public_configs` which applies only to targets that directly depend on your target (not transitively).

Now if you compile and run, you'll see the new version with two people:

```
> ninja -C out/Default say_hello
ninja: Entering directory 'out/Default'
[1/1] Regenerating ninja files
[4/4] LINK say_hello
> out/Default/say_hello
Hello, Bill and Joy.
```

## Add a new build argument

You declare which arguments you accept and specify default values via `declare_args` .

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

See `gn help buildargs` for an overview of how this works. See `gn help declare_args` for specifics on declaring them.

It is an error to declare a given argument more than once in a given scope, so care should be used in scoping and naming arguments.

## Don't know what's going on?

You can run GN in verbose mode to see lots of messages about what it's doing. Use `-v` for this.

### Print debugging

There is a `print` command which just writes to stdout:

```
static_library("hello") {
  ...
  print(configs)
}
```

This will print all of the configs applying to your target (including the default ones).

## The “desc” command

You can run `gn desc <build_dir> <targetname>` to get information about a given target:

```
gn desc out/Default //tools/gn/tutorial:say_hello
```

will print out lots of exciting information. You can also print just one section. Lets say you wanted to know where your `TWO_PEOPLE` define came from on the `say_hello` target:

```
> gn desc out/Default //tools/gn/tutorial:say_hello defines --blame
...lots of other stuff omitted...
  From //tools/gn/tutorial:hello_config
       (Added by //tools/gn/tutorial/BUILD.gn:12)
    TWO_PEOPLE
```

You can see that `TWO_PEOPLE` was defined by a config, and you can also see the which line caused that config to be applied to your target (in this case, the `all_dependent_configs` line).

Another particularly interesting variation:

```
gn desc out/Default //base:base_i18n deps --tree
```

See `gn help desc` for more.

## Performance

You can see what took a long time by running it with the –time command line flag. This will output a summary of timings for various things.

You can also make a trace of how the build files were executed:

```
gn --tracelog=mylog.trace
```

and you can load the resulting file in Chrome’s `about:tracing` page to look at everything.

Powered by Gitiles source log blame