Using GN build

Artisanal metabuild

Brett Wilson

Project info

- **Mailing list**
  gn-dev@chromium.org (public)
- **Source code**
  gn.googlesource.com
- **All help in one file**
  docs/reference.md (concatenated from “gn help”)

History

- **Chrome inception: Visual Studio project files**
- **Chrome 2009: GYP**
  For Mac. Full fidelity with Visual Studio projects.
- **Chrome 2015: GN conversion starts**
  ~100× complexity. Everybody targets Ninja.

Important files

- **.gn**
  - Defines root of GN build tree.
  - See “gn help dotfile”

- **//build/config/BUILDCONFIG.gn**
  - Exact location defined by “.gn” file
  - Sets up global variables and default settings

Make an output directory once!

```
> gn gen out/Default
Done.

> touch base/BUILD.gn

> ninja -C out/Default base
[1/1] Regenerating ninja files
[101/323] CXX obj/base/icu_utf.o
...



> gn clean out/Default
```

Simple example

```
static_library(“base”) {
  sources = [
    “a.cc”,
    “b.cc”,
  ]
}
```

Dependencies

```
static_library(“base”) {
  sources = [
    “a.cc”,
    “b.cc”,
  ]

  deps = [
    “//fancypants”,
    “//foo/bar:baz”,
  ]
}
```

More about labels

Full label

`//chrome/browser:version`

→ *Looks for “version” in chrome/browser/BUILD.gn*

Implicit name

`//base`

→ *Shorthand for //base:base*

Useful when a folder has a “main thing”.

In current file

`:baz`

→ *Shorthand for “baz” in current file.*

Built-in target types

- **executable**, **shared_library**, **static_library**
- **loadable_module**: like a shared library but loaded at runtime
- **source_set**: compiles source files with no intermediate library

- **group**: a named group of targets (deps but no sources)
- **copy**
- **action, action_foreach**: run a script
- **bundle_data**, **create_bundle**: Mac & iOS

Common Chrome-defined ones

- **component**: shared library or source set depending on mode
- **test**
- **app**: executable or iOS application + bundle
- **android_apk**, **generate_jni**, etc.: Lots of Android ones!

Conditionals and expressions

```
component(“base”) {
  sources = [
    “a.cc”,
    “b.cc”,
  ]

  if (is_win || is_linux) {
    sources += [ “win_helper.cc” ]
  } else {
    sources -= [ “a.cc” ]
  }
}
```

Compiler configuration

```
executable(“doom_melon”) {
  sources = [ “doom_melon.cc” ]

  cflags = [ “-Wall” ]
  defines = [ “EVIL_BIT=1” ]
  include_dirs = [ “.” ]

  deps = [ “//base” ]
}
```

```
gn help
```

Configs group flags with a name.

- Additive
- Atomic

```
config(“myconfig”) {
  defines = [ “EVIL_BIT=1” ]
}

executable(“doom_melon”) {
  ...
  configs += [ “:myconfig” ]
}

test(“doom_melon_tests”) {
  ...
  configs += [ “:myconfig” ]
}
```

Apply settings to targets that depend on you.

```
config(“icu_dirs”) {
  include_dirs = [ “include” ]
}

shared_library(“icu”) {
  public_configs = [ “:icu_dirs” ]
}

executable(“doom_melon”) {
  deps = [
    # Apply ICU’s public_configs.
    “:icu”,
  ]
}
```

Forward public configs up the dependency chain.

```
shared_library(“i18n_utils”) {
  …
  public_deps = [
    “//third_party/icu”,
  ]
}

executable(“doom_melon”) {
  deps = [
    # Apply ICU’s public_configs.
    “:i18n_utils”,
  ]
}
```

Some things the code loads dynamically.

```
test(“doom_melon_tests”) {
  # This file is loaded @ runtime.
  data = [
    “melon_cache.txt”,
  ]
}


shared_library(“icu”) {
  # This target is loaded @ runtime.
  data_deps = [
    “:icu_data_tables”,
  ]
}
```

I have no idea what is going on.

```
> gn desc out/Default //base
... <lots o' stuff> ...

> gn desc out/Default
        //tools/gn deps --tree
  //base:base
    //base:base_paths
    //base:base_static
    //base:build_date
    //base:copy_dbghelp.dll
    //base:debugging_flags
    //base/allocator:allocator
      //base/allocator:allocator_shim
        //base/allocator:prep_libc
    //base/third_party/dynamic_annotations:dynamic_annotations
    //base/trace_event/etw_manifest:chrome_events_win
    //build/config/sanitizers:deps
    //third_party/modp_b64:modp_b64
  //build/config/sanitizers:deps
  //tools/gn:gn_lib
    //base:base...
    //base/third_party/dynamic_annotations:dynamic_annotations
  //tools/gn:last_commit_position
```

Drowning in flags!

```
> gn desc out/Default
                //base cflags --blame

From //build/config/compiler:default_optimization
     (Added by //build/config/BUILDCONFIG.gn:456)
  /Od
  /Ob0
  /RTC1
From //build/config/compiler:default_symbols
     (Added by //build/config/BUILDCONFIG.gn:457)
  /Zi
From //build/config/compiler:runtime_library
     (Added by //build/config/BUILDCONFIG.gn:459)
  /MTd
From //build/config:precompiled_headers
     (Added by //base/BUILD.gn:968)
  /FIbuild/precompile.h
From //build/config/compiler:no_size_t_to_int_warning
     (Added by //base/BUILD.gn:1163)
  /wd4267
```

What targets exist?

```
> gn ls out/Default “//base/*”
//base:base
//base:base_i18n_perftests
//base:base_i18n_perftests_run
//base:base_paths
//base:base_perftests
//base:base_perftests_run
//base:base_static
//base:base_unittests
//base:base_unittests_bundle_data
//base:base_unittests_run
//base:build_date
//base:build_utf8_validator_tables
//base:check_example
//base:debugging_flags
//base:i18n
//base:message_loop_tests
//base/allocator:allocator
//base/allocator:features
//base/allocator:tcmalloc
```

How do I depend on that?

Why can’t I use a header from a dependency?

```
> gn path out/Default
        //content/browser //cc/base

//content/browser:browser --[private]-->
//cc:cc --[private]-->
//cc/base:base

Showing one of 118 unique non-data paths.
0 of them are public.
Use --all to print all paths.
```

What references something?

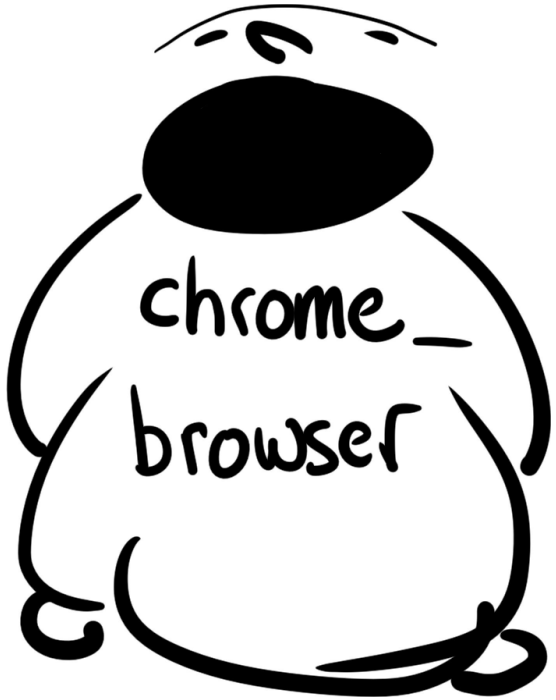
```
> gn refs out/Default //cc
//ash:ash
//ash/mus:lib
//blimp/client:blimp_client
...
```

```
> gn refs out/Default //cc --tree
//media/blink:blink
  //media/blink:media_blink_unittests
    //media/blink:media_blink_unittests_run
...
```

```
> gn refs out/Default
                    //base/macros.h
//base:base
```

Stepping back

How should you design your build?



Design your build like *code.*

- **Modular**
  GN 💖 small targets and lots of directories!

- **Clear relationship between modules**

Protect your code from your team.

- **deps vs. public_deps** — control how you expose your dependencies
- **visibility** — limit what can depend on you
- **assert_no_deps** — “none of my dependencies should link Blink”
- **testonly** — can’t be linked into production code
- **List public headers in “public”** — other headers become “private”

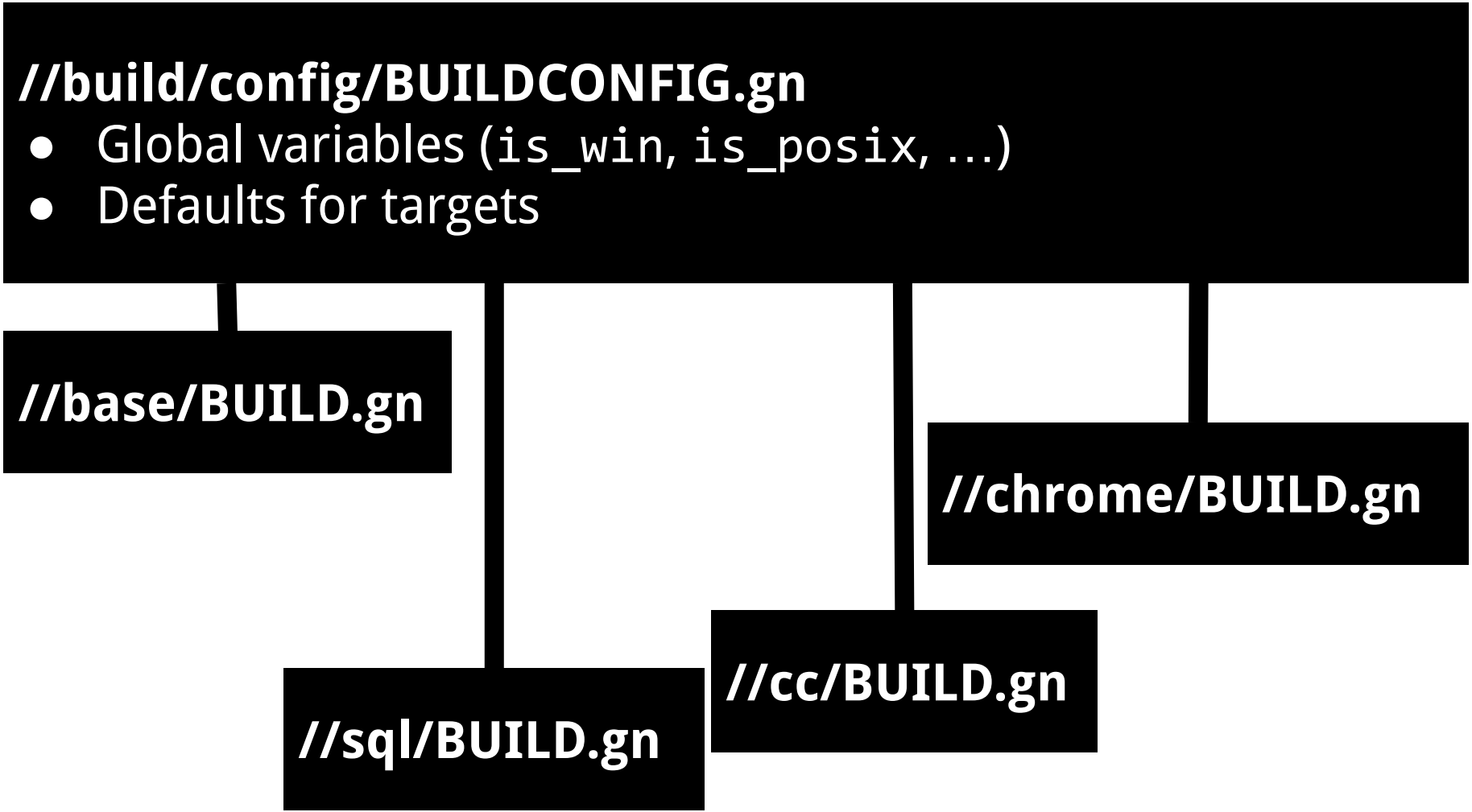
“gn check” validates includes.

```
> gn check out/Default

ERROR at //base/files/file_path.cc
#include "sql/statement.h"
          ^--------------
It is not in any dependency of
  //base:base
The include file is in the target(s):
  //sql:sql
which should somehow be reachable.
```

More advanced stuff

Build structure.


//build/config/BUILDCONFIG.gn
● Global variables (is_win, is_posix, ...)
● Defaults for targets
//base/BUILD.gn
//sql/BUILD.gn
//cc/BUILD.gn
//chrome/BUILD.gn

```
executable(“doom_melon”) {
  print(configs)
  ...
}
```

```
> gn gen out/Default

["//build/config:feature_flags",
"//build/config/compiler:compiler",
"//build/config/compiler:clang_stackrealign",
"//build/config/compiler:compiler_arm_fpu",
"//build/config/compiler:chromium_code",
"//build/config/compiler:default_include_dirs",
"//build/config/compiler:default_optimization",
"//build/config/compiler:default_symbols",
"//build/config/compiler:no_rtti",
"//build/config/compiler:runtime_library",
"//build/config/sanitizers:default_sanitizer_flags",
"//build/config/sanitizers:default_sanitizer_coverage_flags",
"//build/config/win:lean_and_mean",
"//build/config/win:nominmax",
"//build/config/win:unicode",
"//build/config/win:winver",
"//build/config:debug"]
```

A target can modify the configs to opt-out of defaults.

```
executable(“doom_melon”) {
  configs -= [
    "//build/config/compiler:chromium_code",
  ]
  configs += [
    "//build/config/compiler:no_chromium_code",
  ]
}
```

Documentation (!?!?!?)

Arg name

Default value

```
declare_args() {
  # Allow unlimited requests
  # to the Google speech API.
  bypass_speech_api_quota = false
}


executable("doom_melon") {
  if (bypass_speech_api_quota) {
    ...
  }
}
```

```
> gn args out/Default
```

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

```
> gn args --list out/Default
```

```
v8_use_snapshot  Default = true
    //v8/BUILD.gn:23
    Enable the snapshot feature, for fast context creation.
    http://v8project.blogspot.com/2015/09/custom-startup-snapshots.html

visual_studio_path  Default = ""
    //build/config/win/visual_studio_version.gni:9
    Path to Visual Studio. If empty, the default is used which is to use the
    automatic toolchain in depot_tools. If set, you must also set the
    visual_studio_version and wdk_path.

visual_studio_version  Default = ""
    //build/config/win/visual_studio_version.gni:13
    Version of Visual Studio pointed to by the visual_studio_path.
    Use "2013" for Visual Studio 2013, or "2013e" for the Express version.

wdk_path  Default = ""
    //build/config/win/visual_studio_version.gni:17
    Directory of the Windows driver kit. If visual_studio_path is empty, this
    will be auto-filled.

win_console_app  Default = false
    //build/config/win/console_app.gni:12
    If true, builds as a console app (rather than a windowed app), which allows
    logging to be printed to the user. This will cause a terminal window to pop
    up when the executable is not run from the command line, so should only be
    used for development. Only has an effect on Windows builds.

windows_sdk_path  Default = "C:\Program Files (x86)\Windows Kits\10"
    //build/config/win/visual_studio_version.gni:22
    Full path to the Windows SDK, not including a backslash at the end.
    This value is the default location, override if you have a different
    installation location.
```

Shared variables are put in a *.gni file and imported.

```
declare_args() {
  # Controls Chrome branding.
  is_chrome_branded = false
}

enable_crashing = is_win
```

```
import("//foo/build.gni")

executable("doom_melon") {
  if (is_chrome_branded) {
    ...
  }
  if (enable_crashing) {
    ...
  }
}
```

Advanced doodads.

Templates & actions

Templates allow creating of new target types.

```
template(“grit”) {
  …
}

grit(“components_strings”) {
  source = “components.grd”
  outputs = [ … ]
}
```

Actions run Python scripts.

```
action(“myaction”) {
  script = “myscript.py”
```

Dependency management.

```
action(“myaction”) {
  script = “myscript.py”
  inputs = [ “myfile.txt” ]
  outputs = [
    …
  ]
```

This writes a file to the source tree!

```
action(“myaction”) {
  script = “myscript.py”
  inputs = [ “myfile.txt” ]
  outputs = [
    “generated.txt”,  # Error!
  ]
```

```
gn help
```

Put outputs in the target-specific out directory.

```
action(“myaction”) {
  script = “myscript.py”
  inputs = [ “myfile.txt” ]
  outputs = [
    target_out_dir + “/output.txt”,
  ]
  print(outputs)
```

```
> gn gen out/Default
[“//out/Default/obj/foo/output.txt”]
```

Use *$foo* or *${foo}* to expand variables in strings.

```
action("myaction") {
  script = "myscript.py"
  inputs = [ "myfile.txt" ]
  outputs = [
    "$target_out_dir/output.txt",
  ]
  print("out = $outputs")
```

```
> gn gen out/Default
out = ["//out/Default/obj/foo/output.txt"]
```

Args are what is passed to the script.

```
action(“myaction”) {
  script = “myscript.py”
  inputs = [ “myfile.txt” ]
  outputs = [
    “$target_out_dir/output.txt”,
  ]
  args = [
    “-i”, inputs[0], outputs[0],
  ]
}
```

```
>>> ERROR can’t open “myfile.txt”
or “//out/Default/obj/output.txt”
```

The script working directory is *root_build_dir*

```
action("myaction") {
  script = "myscript.py"
  inputs = [ "myfile.txt" ]
  outputs = [
    "$target_out_dir/output.txt",
  ]
  args = [
    "-i",
    rebase_path(inputs[0],
                root_build_dir)
    rebase_path(outputs[0],
                root_build_dir)
  ]
}
```

action_foreach runs a script over each source.

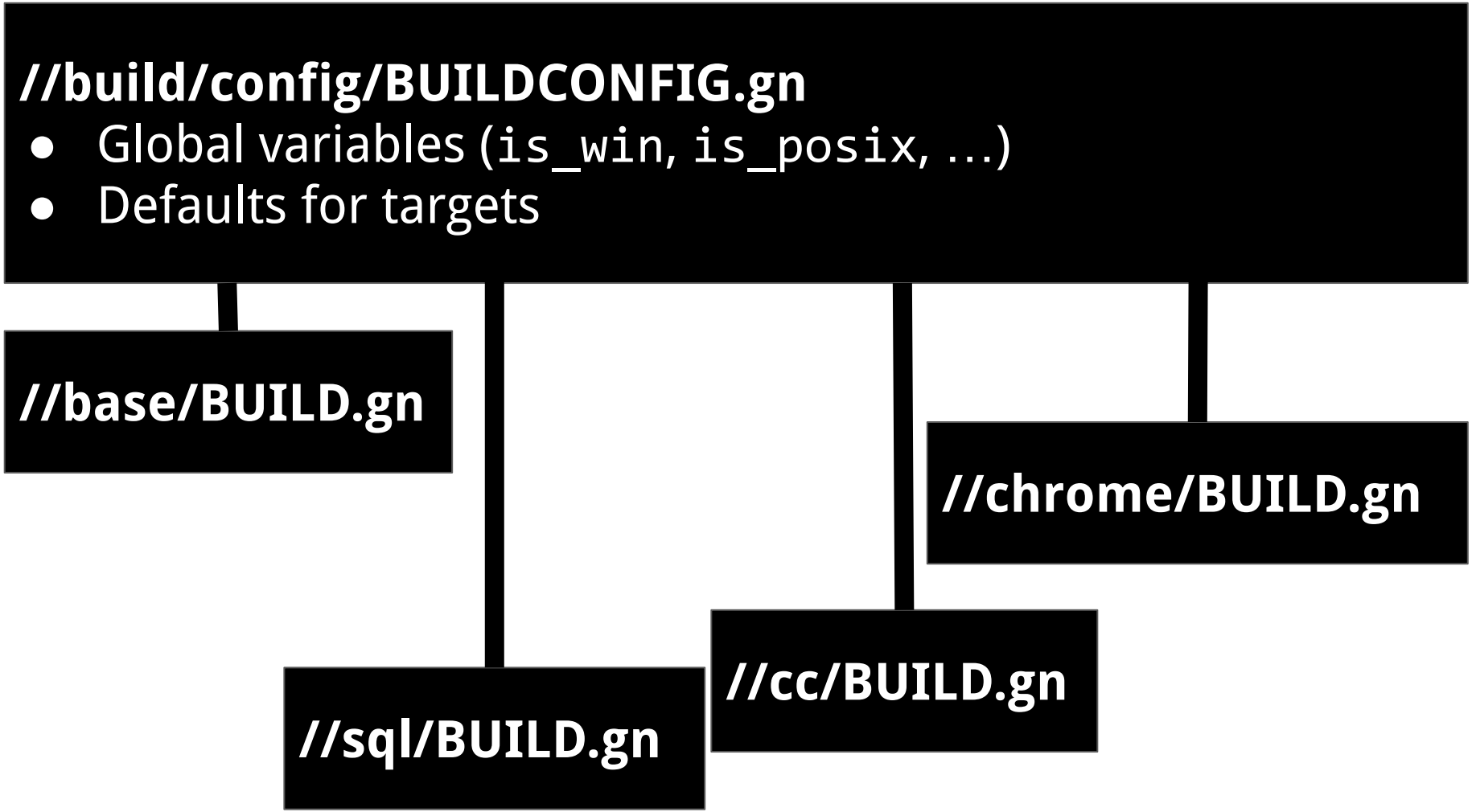
```
action_foreach(“process_idl”) {
  script = “idl_compiler.py”
  inputs = [ “static_input.txt” ]
  sources = [
    “a.idl”,
    “b.idl”,
  ]
```

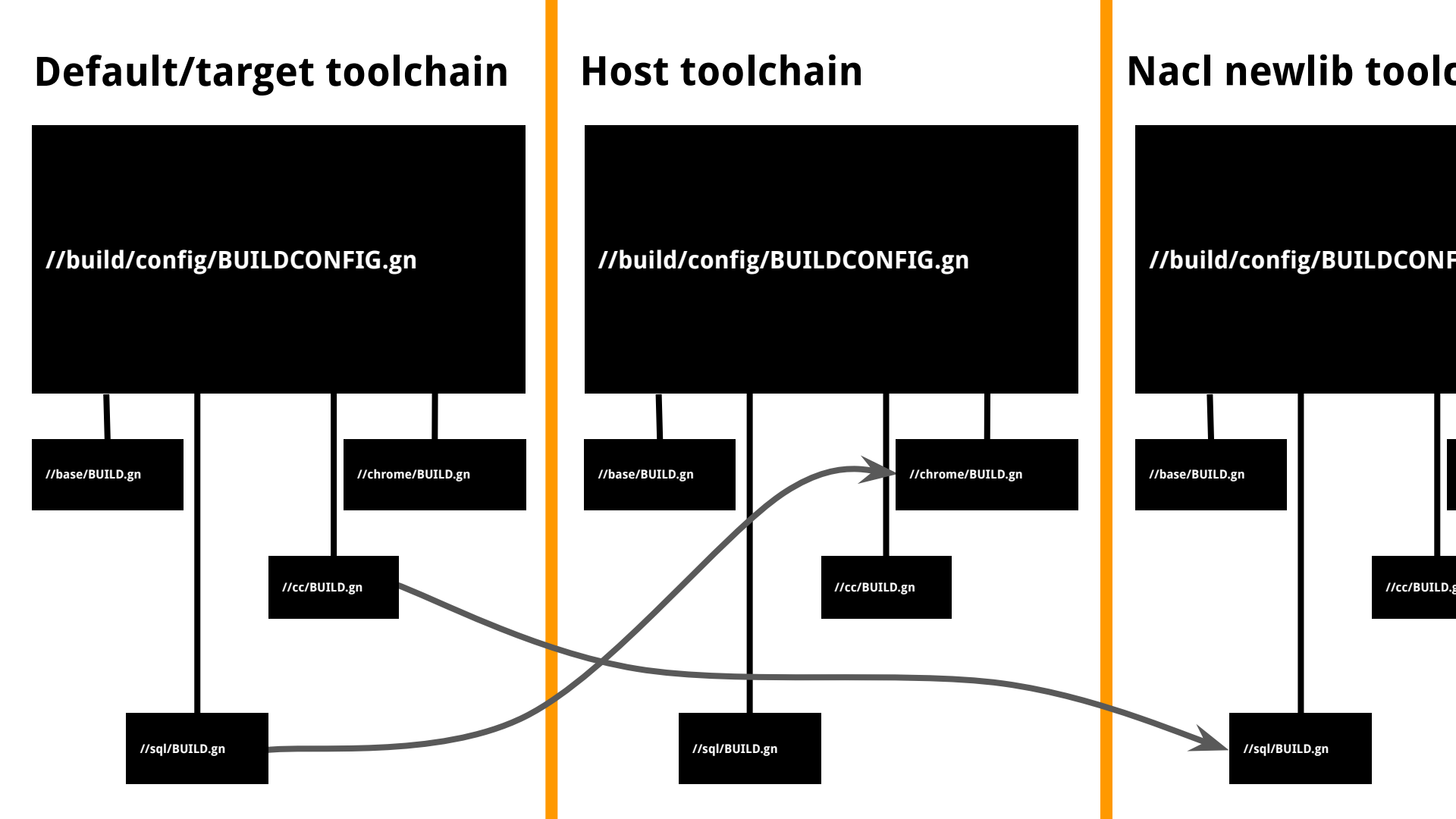
Magic substitutions for dealing with multiple sources.

```
action_foreach("process_idl") {
  script = "idl_compiler.py"
  inputs = [ "static_input.txt" ]
  sources = [
    "a.idl",
    "b.idl",
  ]
  outputs = [
    "$target_gen_dir/{{source_name_part}}.h"
  ]
  args = [
    "--input={{source}}"
  ]
}
```

Toolchains

Imagine your build as an *n*-dimensional hypercube...


//build/config/BUILDCONFIG.gn
● Global variables (is_win, is_posix, ...)
● Defaults for targets
//base/BUILD.gn
//sql/BUILD.gn
//cc/BUILD.gn
//chrome/BUILD.gn


Default/target toolchain
//build/config/BUILDCONFIG.gn
//base/BUILD.gn
//chrome/BUILD.gn
//cc/BUILD.gn
//sql/BUILD.gn
Host toolchain
//build/config/BUILDCONFIG.gn
//base/BUILD.gn
//chrome/BUILD.gn
//cc/BUILD.gn
//sql/BUILD.gn
Nacl newlib toolc
//build/config/BUILDCONF
//base/BUILD.gn
//cc/BUILD.g
//sql/BUILD.gn

What's a toolchain?

- **Identified by a label**
- **Defines a set of compiler and linker rules.**
- **Goes with a set of variables (OS, CPU, etc.)**

Cross-toolchain dependencies.

```
executable(“chrome”) {
  …
  data_deps = [
    “//nacl:irt(//build/toolchain/nacl:newlib)”
  ]
}

action(“compile_some_protos”) {
  …
  deps = [
    “:proto_compiler($host_toolchain)”
  ]
}
```

Comparing toolchains.

```
if (current_toolchain ==
    host_toolchain) {
  executable(“proto_compiler”) {
    …
  }
}
```

Other things that exist

- **Generate projects for popular IDEs**
  → see “gn help gen”

fin.

Bonus advanced content

Magic target_name variable expands to “components_strings” in this example.

Access the variables from the caller via “invoker.”

```
template(“grit”) {
  action(target_name) {
    script = “//tools/grit.py”
    sources = [ invoker.source ]
    ...
  }
}
```

```
grit(“components_strings”) {
  source = “components.grd”
  outputs = [ ... ]
}
```

exec_script:
The universal escape hatch.

```
gypi_values = exec_script(
  “//build/gypi_to_gn.py”,
  [ rebase_path(“chrome_browser.gypi”) ],
  “scope”,
  [ “chrome_browser.gypi” ])
```