GYP

User Documentation

Contents

Introduction

This document is intended to provide a user-level guide to GYP. The emphasis here is on how to use GYP to accomplish specific tasks, not on the complete technical language specification. (For that, see the LanguageSpecification.)

The document below starts with some overviews to provide context: an overview of the structure of a `.gyp` file itself, an overview of a typical executable-program target in a `.gyp` file, an an overview of a typical library target in a `.gyp` file.

After the overviews, there are examples of `gyp` patterns for different common use cases.

Skeleton of a typical Chromium .gyp file

Here is the skeleton of a typical `.gyp` file in the Chromium tree:

```
{
  'variables': {
    .
    .
    .
  },
  'includes': [
    '../build/common.gypi',
  ],
  'target_defaults': {
    .
    .
    .
  },
  'targets': [
    {
      'target_name': 'target_1',
        .
        .
        .
    },
    {
      'target_name': 'target_2',
        .
        .
        .
    },
  ],
  'conditions': [
    ['OS=="linux"', {
      'targets': [
        {
          'target_name': 'linux_target_3',
            .
            .
            .
        },
      ],
    }],
    ['OS=="win"', {
      'targets': [
        {
          'target_name': 'windows_target_4',
            .
            .
            .
        },
      ],
```

```
    }, { # OS != "win"
      'targets': [
        {
          'target_name': 'non_windows_target_5',
            .
            .
            .
        },
    }],
  ],
}
```

The entire file just contains a Python dictionary. (It's actually JSON, with two small Pythonic deviations: comments are introduced with `#`, and a `,` (comma)) is legal after the last element in a list or dictionary.)

The top-level pieces in the `.gyp` file are as follows:

`'variables'` : Definitions of variables that can be interpolated and used in various other parts of the file.

`'includes'` : A list of of other files that will be included in this file. By convention, included files have the suffix `.gypi` (gyp include).

`'target_defaults'` : Settings that will apply to *all* of the targets defined in this `.gyp` file.

`'targets'` : The list of targets for which this `.gyp` file can generate builds. Each target is a dictionary that contains settings describing all the information necessary to build the target.

`'conditions'` : A list of condition specifications that can modify the contents of the items in the global dictionary defined by this `.gyp` file based on the values of different variablwes. As implied by the above example, the most common use of a `conditions` section in the top-level dictionary is to add platform-specific targets to the `targets` list.

Skeleton of a typical executable target in a .gyp file

The most straightforward target is probably a simple executable program. Here is an example `executable` target that demonstrates the features that should cover most simple uses of gyp:

```
{
  'targets': [
    {
      'target_name': 'foo',
      'type': 'executable',
      'msvs_guid': '5ECEC9E5-8F23-47B6-93E0-C3B328B3BE65',
      'dependencies': [
        'xyzzy',
        '../bar/bar.gyp:bar',
      ],
      'defines': [
```

```
          'DEFINE_FOO',
          'DEFINE_A_VALUE=value',
        ],
        'include_dirs': [
          '..',
        ],
        'sources': [
          'file1.cc',
          'file2.cc',
        ],
        'conditions': [
          ['OS=="linux"', {
            'defines': [
              'LINUX_DEFINE',
            ],
            'include_dirs': [
              'include/linux',
            ],
          }],
          ['OS=="win"', {
            'defines': [
              'WINDOWS_SPECIFIC_DEFINE',
            ],
          }, { # OS != "win",
            'defines': [
              'NON_WINDOWS_DEFINE',
            ],
          }]
        ],
      },
    ],
  }
```

The top-level settings in the target include:

'target_name' : The name by which the target should be known, which should be unique across all .gyp files. This name will be used as the project name in the generated Visual Studio solution, as the target name in the generated XCode configuration, and as the alias for building this target from the command line of the generated SCons configuration.

'type' : Set to executable , logically enough.

'msvs_guid' : THIS IS ONLY TRANSITIONAL. This is a hard-coded GUID values that will be used in the generated Visual Studio solution file(s). This allows us to check in a chrome.sln file that interoperates with gyp-generated project files. Once everything in Chromium is being generated by gyp, it will no longer be important that the GUIDs stay constant across invocations, and we'll likely get rid of these settings,

'dependencies' : This lists other targets that this target depends on. The gyp-generated files will

guarantee that the other targets are built before this target. Any library targets in the `dependencies` list will be linked with this target. The various settings (`defines` , `include_dirs` , etc.) listed in the `direct_dependent_settings` sections of the targets in this list will be applied to how *this* target is built and linked. See the more complete discussion of `direct_dependent_settings` , below.

`'defines'` : The C preprocessor definitions that will be passed in on compilation command lines (using `-D` or `/D` options).

`'include_dirs'` : The directories in which included header files live. These will be passed in on compilation command lines (using `-I` or `/I` options).

`'sources'` : The source files for this target.

`'conditions'` : A block of conditions that will be evaluated to update the different settings in the target dictionary.

Skeleton of a typical library target in a .gyp file

The vast majority of targets are libraries. Here is an example of a library target including the additional features that should cover most needs of libraries:

```
{
  'targets': [
    {
      'target_name': 'foo',
      'type': '<(library)'
      'msvs_guid': '5ECEC9E5-8F23-47B6-93E0-C3B328B3BE65',
      'dependencies': [
        'xyzzy',
        '../bar/bar.gyp:bar',
      ],
      'defines': [
        'DEFINE_FOO',
        'DEFINE_A_VALUE=value',
      ],
      'include_dirs': [
        '..',
      ],
      'direct_dependent_settings': {
        'defines': [
          'DEFINE_FOO',
          'DEFINE_ADDITIONAL',
        ],
        'linkflags': [
        ],
      },
      'export_dependent_settings': [
        '../bar/bar.gyp:bar',
      ],
```

```
        ],
        'sources': [
          'file1.cc',
          'file2.cc',
        ],
        'conditions': [
          ['OS=="linux"', {
            'defines': [
              'LINUX_DEFINE',
            ],
            'include_dirs': [
              'include/linux',
            ],
          ],
          ['OS=="win"', {
            'defines': [
              'WINDOWS_SPECIFIC_DEFINE',
            ],
          }, { # OS != "win",
            'defines': [
              'NON_WINDOWS_DEFINE',
            ],
          }]
        ],
    ],
  }
```

The possible entries in a library target are largely the same as those that can be specified for an executable target (defines , include_dirs , etc.). The differences include:

'type' : This should almost always be set to '<(library)', which allows the user to define at gyp time whether libraries are to be built static or shared. (On Linux, at least, linking with shared libraries saves significant link time.) If it's necessary to pin down the type of library to be built, the type can be set explicitly to static_library or shared_library .

'direct_dependent_settings' : This defines the settings that will be applied to other targets that *directly depend* on this target–that is, that list *this* target in their 'dependencies' setting. This is where you list the defines , include_dirs , cflags and linkflags that other targets that compile or link against this target need to build consistently.

'export_dependent_settings' : This lists the targets whose direct_dependent_settings should be "passed on" to other targets that use (depend on) this target. TODO: expand on this description.

Use Cases

These use cases are intended to cover the most common actions performed by developers using GYP.

Note that these examples are *not* fully-functioning, self-contained examples (or else they'd be way too long). Each example mostly contains just the keywords and settings relevant to the example, with perhaps a few extra keywords for context. The intent is to try to show the specific pieces you need to pay attention to when doing something. [NOTE: if practical use shows that these examples are confusing without additional context, please add what's necessary to clarify things.]

Add new source files

There are similar but slightly different patterns for adding a platform-independent source file vs. adding a source file that only builds on some of the supported platforms.

Add a source file that builds on all platforms

Simplest possible case: You are adding a file(s) that builds on all platforms.

Just add the file(s) to the `sources` list of the appropriate dictionary in the `targets` list:

```
{
  'targets': [
    {
      'target_name': 'my_target',
      'type': 'executable',
      'sources': [
        '../other/file_1.cc',
        'new_file.cc',
        'subdir/file3.cc',
      ],
    },
  ],
},
```

File path names are relative to the directory in which the `.gyp` file lives.

Keep the list sorted alphabetically (unless there's a really, really, *really* good reason not to).

Add a platform-specific source file

Your platform-specific file is named `*_linux.{ext}`, `*_mac.{ext}`, `*_posix.{ext}` or `*_win.{ext}`

The simplest way to add a platform-specific source file, assuming you're adding a completely new file and get to name it, is to use one of the following standard suffixes:

- `_linux` (e.g. `foo_linux.cc`)
- `_mac` (e.g. `foo_mac.cc`)
- `_posix` (e.g. `foo_posix.cc`)
- `_win` (e.g. `foo_win.cc`)

Simply add the file to the `sources` list of the appropriate dict within the `targets` list, like you would any other source file.

```
{
  'targets': [
    {
      'target_name': 'foo',
      'type': 'executable',
      'sources': [
        'independent.cc',
        'specific_win.cc',
      ],
    },
  ],
},
```

The Chromium `.gyp` files all have appropriate `conditions` entries to filter out the files that aren't appropriate for the current platform. In the above example, the `specific_win.cc` file will be removed automatically from the source-list on non-Windows builds.

Your platform-specific file does not use an already-defined pattern

If your platform-specific file does not contain a `*_{linux,mac,posix,win}` substring (or some other pattern that's already in the `conditions` for the target), and you can't change the file name, there are two patterns that can be used.

Prefererred: Add the file to the `sources` list of the appropriate dictionary within the `targets` list. Add an appropriate `conditions` section to exclude the specific files name:

```
{
  'targets': [
    {
      'target_name': 'foo',
      'type': 'executable',
      'sources': [
        'linux_specific.cc',
      ],
      'conditions': [
        ['OS != "linux"', {
          'sources!': [
            # Linux-only; exclude on other platforms.
            'linux_specific.cc',
          ]
        }[,
      ],
    },
  ],
},
```

Despite the duplicate listing, the above is generally preferred because the `sources` list contains a useful global list of all sources on all platforms with consistent sorting on all platforms.

Non-preferred: In some situations, however, it might make sense to list a platform-specific file only in a `conditions` section that specifically *includes* it in the `sources` list:

```
{
  'targets': [
    {
      'target_name': 'foo',
      'type': 'executable',
      'sources': [],
      ['OS == "linux"', {
        'sources': [
          # Only add to sources list on Linux.
          'linux_specific.cc',
        ]
      }],
    },
  ],
},
```

The above two examples end up generating equivalent builds, with the small exception that the `sources` lists will list the files in different orders. (The first example defines explicitly where `linux_specific.cc` appears in the list–perhaps in in the middle–whereas the second example will always tack it on to the end of the list.)

Including or excluding files using patterns: There are more complicated ways to construct a `sources` list based on patterns. See `TODO` below.

Add a new executable

An executable program is probably the most straightforward type of target, since all it typically needs is a list of source files, some compiler/linker settings (probably varied by platform), and some library targets on which it depends and which must be used in the final link.

Add an executable that builds on all platforms

Add a dictionary defining the new executable target to the `targets` list in the appropriate `.gyp` file. Example:

```
{
  'targets': [
    {
      'target_name': 'new_unit_tests',
      'type': 'executable',
      'defines': [
        'FOO',
      ],
      'include_dirs': [
        '..',
```

```
        ],
        'dependencies': [
          'other_target_in_this_file',
          'other_gyp2:target_in_other_gyp2',
        ],
        'sources': [
          'new_additional_source.cc',
          'new_unit_tests.cc',
        ],
      },
    ],
  }
```

Add a platform-specific executable

Add a dictionary defining the new executable target to the `targets` list within an appropriate `conditions` block for the platform. The `conditions` block should be a sibling to the top-level `targets` list:

```
  {
    'targets': [
    ],
    'conditions': [
      ['OS=="win"', {
        'targets': [
          {
            'target_name': 'new_unit_tests',
            'type': 'executable',
            'defines': [
              'FOO',
            ],
            'include_dirs': [
              '..',
            ],
            'dependencies': [
              'other_target_in_this_file',
              'other_gyp2:target_in_other_gyp2',
            ],
            'sources': [
              'new_additional_source.cc',
              'new_unit_tests.cc',
            ],
          },
        ],
      }],
    ],
  }
```

Add settings to a target

There are several different types of settings that can be defined for any given target.

Add new preprocessor definitions (-D or /D flags)

New preprocessor definitions are added by the defines setting:

```
{
  'targets': [
    {
      'target_name': 'existing_target',
      'defines': [
        'FOO',
        'BAR=some_value',
      ],
    },
  ],
},
```

These may be specified directly in a target's settings, as in the above example, or in a conditions section.

Add a new include directory (-I or /I flags)

New include directories are added by the include_dirs setting:

```
{
  'targets': [
    {
      'target_name': 'existing_target',
      'include_dirs': [
        '..',
        'include',
      ],
    },
  ],
},
```

These may be specified directly in a target's settings, as in the above example, or in a conditions section.

Add new compiler flags

Specific compiler flags can be added with the cflags setting:

```
{
  'targets': [
    {
```

```
        'target_name': 'existing_target',
        'conditions': [
          ['OS=="win"', {
            'cflags': [
              '/WX',
            ],
          }, { # OS != "win"
            'cflags': [
              '-Werror',
            ],
          }],
        ],
      },
    ],
  },
```

Because these flags will be specific to the actual compiler involved, they will almost always be only set within a `conditions` section.

Add new linker flags

Setting linker flags is OS-specific. On linux and most non-mac posix systems, they can be added with the `ldflags` setting:

```
{
  'targets': [
    {
      'target_name': 'existing_target',
      'conditions': [
        ['OS=="linux"', {
          'ldflags': [
            '-pthread',
          ],
        }],
      ],
    },
  ],
},
```

Because these flags will be specific to the actual linker involved, they will almost always be only set within a `conditions` section.

On OS X, linker settings are set via `xcode_settings`, on Windows via `msvs_settings`.

Exclude settings on a platform

Any given settings keyword (`defines`, `include_dirs`, etc.) has a corresponding form with a trailing `!` (exclamation point) to remove values from a setting. One useful example of this is to remove the Linux `-Werror` flag from the global settings defined in `build/common.gypi`:

```
{
  'targets': [
    {
      'target_name': 'third_party_target',
      'conditions': [
        ['OS=="linux"', {
          'cflags!': [
            '-Werror',
          ],
        }],
      ],
    },
  ],
},
```

Cross-compiling

GYP has some (relatively limited) support for cross-compiling.

If the variable GYP_CROSSCOMPILE or one of the toolchain-related variables (like CC_host or CC_target) is set, GYP will think that you wish to do a cross-compile.

When cross-compiling, each target can be part of a “host” build, a “target” build, or both. By default, the target is assumed to be (only) part of the “target” build. The ‘toolsets’ property can be set on a target to change the default.

A target’s dependencies are assumed to match the build type (so, if A depends on B, by default that means that a target build of A depends on a target build of B). You can explicitly depend on targets across toolchains by specifying “#host” or “#target” in the dependencies list. If GYP is not doing a cross-compile, the “#host” and “#target” will be stripped as needed, so nothing breaks.

Add a new library

TODO: write intro

Add a library that builds on all platforms

Add the a dictionary defining the new library target to the targets list in the appropriate .gyp file. Example:

```
{
  'targets': [
    {
      'target_name': 'new_library',
      'type': '<(library)',
      'defines': [
        'FOO',
        'BAR=some_value',
```

```
        ],
        'include_dirs': [
          '..',
        ],
        'dependencies': [
          'other_target_in_this_file',
          'other_gyp2:target_in_other_gyp2',
        ],
        'direct_dependent_settings': {
          'include_dirs': '.',
        },
        'export_dependent_settings': [
          'other_target_in_this_file',
        ],
        'sources': [
          'new_additional_source.cc',
          'new_library.cc',
        ],
      },
    ],
  }
```

The use of the <(library) variable above should be the default type setting for most library targets, as it allows the developer to choose, at gyp time, whether to build with static or shared libraries. (Building with shared libraries saves a *lot* of link time on Linux.)

It may be necessary to build a specific library as a fixed type. Is so, the type field can be hard-wired appropriately. For a static library:

```
        'type': 'static_library',
```

For a shared library:

```
        'type': 'shared_library',
```

Add a platform-specific library

Add a dictionary defining the new library target to the targets list within a conditions block that's a sibling to the top-level targets list:

```
{
  'targets': [
  ],
  'conditions': [
    ['OS=="win"', {
      'targets': [
        {
          'target_name': 'new_library',
```

```
            'type': '<(library)',
            'defines': [
              'FOO',
              'BAR=some_value',
            ],
            'include_dirs': [
              '..',
            ],
            'dependencies': [
              'other_target_in_this_file',
              'other_gyp2:target_in_other_gyp2',
            ],
            'direct_dependent_settings': {
              'include_dirs': '.',
            },
            'export_dependent_settings': [
              'other_target_in_this_file',
            ],
            'sources': [
              'new_additional_source.cc',
              'new_library.cc',
            ],
          },
        ],
      }],
    ],
  }
```

Dependencies between targets

GYP provides useful primitives for establishing dependencies between targets, which need to be configured in the following situations.

Linking with another library target

```
  {
    'targets': [
      {
        'target_name': 'foo',
        'dependencies': [
          'libbar',
        ],
      },
      {
        'target_name': 'libbar',
        'type': '<(library)',
        'sources': [
        ],
```

```
    },
  ],
}
```

Note that if the library target is in a different .gyp file, you have to specify the path to other .gyp file, relative to this .gyp file's directory:

```
{
  'targets': [
    {
      'target_name': 'foo',
      'dependencies': [
        '../bar/bar.gyp:libbar',
      ],
    },
  ],
}
```

Adding a library often involves updating multiple .gyp files, adding the target to the approprate .gyp file (possibly a newly-added .gyp file), and updating targets in the other .gyp files that depend on (link with) the new library.

Compiling with necessary flags for a library target dependency

We need to build a library (often a third-party library) with specific preprocessor definitions or command-line flags, and need to ensure that targets that depend on the library build with the same settings. This situation is handled by a direct_dependent_settings block:

```
{
  'targets': [
    {
      'target_name': 'foo',
      'type': 'executable',
      'dependencies': [
        'libbar',
      ],
    },
    {
      'target_name': 'libbar',
      'type': '<(library)',
      'defines': [
        'LOCAL_DEFINE_FOR_LIBBAR',
        'DEFINE_TO_USE_LIBBAR',
      ],
      'include_dirs': [
        '..',
        'include/libbar',
      ],
```

```
        'direct_dependent_settings': {
          'defines': [
            'DEFINE_TO_USE_LIBBAR',
          ],
          'include_dirs': [
            'include/libbar',
          ],
        },
      },
    ],
  }
```

In the above example, the sources of the `foo` executable will be compiled with the options `-DDEFINE_TO_USE_LIBBAR -Iinclude/libbar`, because of those settings' being listed in the `direct_dependent_settings` block.

Note that these settings will likely need to be replicated in the settings for the library target itsef, so that the library will build with the same options. This does not prevent the target from defining additional options for its "internal" use when compiling its own source files. (In the above example, these are the `LOCAL_DEFINE_FOR_LIBBAR` define, and the `..` entry in the `include_dirs` list.)

When a library depends on an additional library at final link time

```
  {
    'targets': [
      {
        'target_name': 'foo',
        'type': 'executable',
        'dependencies': [
          'libbar',
        ],
      },
      {
        'target_name': 'libbar',
        'type': '<(library)',
        'dependencies': [
          'libother'
        ],
        'export_dependent_settings': [
          'libother'
        ],
      },
      {
        'target_name': 'libother',
        'type': '<(library)',
        'direct_dependent_settings': {
          'defines': [
```

```
            'DEFINE_FOR_LIBOTHER',
          ],
          'include_dirs': [
            'include/libother',
          ],
        },
      },
    ],
  }
```

Support for Mac OS X bundles

gyp supports building bundles on OS X (.app, .framework, .bundle, etc). Here is an example of this:

```
    {
      'target_name': 'test_app',
      'product_name': 'Test App Gyp',
      'type': 'executable',
      'mac_bundle': 1,
      'sources': [
        'main.m',
        'TestAppAppDelegate.h',
        'TestAppAppDelegate.m',
      ],
      'mac_bundle_resources': [
        'TestApp/English.lproj/InfoPlist.strings',
        'TestApp/English.lproj/MainMenu.xib',
      ],
      'link_settings': {
        'libraries': [
          '$(SDKROOT)/System/Library/Frameworks/Cocoa.framework',
        ],
      },
      'xcode_settings': {
        'INFOPLIST_FILE': 'TestApp/TestApp-Info.plist',
      },
    },
```

The `mac_bundle` key tells gyp that this target should be a bundle. `executable` targets get extension `.app` by default, `shared_library` targets get `.framework` – but you can change the bundle extensions by setting `product_extension` if you want. Files listed in `mac_bundle_resources` will be copied to the bundle's `Resource` folder of the bundle. You can also set `process_outputs_as_mac_bundle_resources` to 1 in actions and rules to let the output of actions and rules be added to that folder (similar to `process_outputs_as_sources`). If `product_name` is not set, the bundle will be named after `target_name` as usual.

Move files (refactoring)

TODO

Custom build steps

TODO

Adding an explicit build step to generate specific files

TODO

Adding a rule to handle files with a new suffix

TODO

Build flavors

TODO