cmany Documentation

Release 0.1

Joao Paulo Magalhaes

Jun 06, 2020

Contents:

Easily batch-build cmake projects!

cmany is a command line tool to easily build variations of a CMake C/C++ project. It combines different compilers, cmake build types, bundles of compilation flags, processor architectures and operating systems. Each of these items can also have associated compilation flags.

For example, to configure and build a project combining clang++ and g++ with both Debug and Release:

```
$ cmany build -c clang++,g++ -t Debug,Release path/to/CMakeLists.txt
```

The command above will result in four different build trees, placed by default under a `build` subdirectory in the current working directory:

```
$ ls build/*
build/linux-x86_64-clang++3.9-Debug
build/linux-x86_64-clang++3.9-Release
build/linux-x86_64-gcc++6.1-Debug
build/linux-x86_64-gcc++6.1-Release
```

Each build tree is obtained by first configuring the project with the items in each combination, and then invoking `cmake --build` to build the project at once.

You can also use cmany just to simplify your cmake workflow! These two command sequences have the same effect (`b` is an alias to `build`):

typical cmake	cmany

`$ mkdir build`
`$ cd build`
`$ cmake ..`
`$ cmake --build .`
	`$ cmany b`

CHAPTER 1

Features

- Easily configures and builds many variations of your project with one simple command.
- Saves the tedious and error-prone work of dealing with many build trees by hand.
- Sensible defaults: `cmany build` will create and build a single project using CMake's defaults.
- Transparently pass flags (compiler flags, processor defines or cmake cache variables) to any or all of the builds.
- Useful for build comparison and benchmarking. You can easily setup bundles of flags, aka variants.
- Useful for validating and unit-testing your project with different compilers and flags.
- Useful for creating distributions of your project.
- Avoids a full rebuild when the build type is changed. Although this feature already exists in multi-configuration cmake generators like Visual Studio, it is missing from mono-configuration generators like Unix Makefiles.
- Runs arbitrary commands in every build tree or install tree.
- Full control over how the build items are combined.
- Emacs integration! https://github.com/biojppm/cmany.el

CHAPTER 2

Contents

			Linux + OS X:	Windows:

2.1 cmany

Easily batch-build cmake projects!

cmany is a command line tool to easily build variations of a CMake C/C++ project. It combines different compilers, cmake build types, compilation flags, processor architectures and operating systems.

For example, to configure and build a project combining clang++ and g++ with both Debug and Release:

```
$ cmany build -c clang++,g++ -t Debug,Release path/to/CMakeLists.txt
```

The command above will result in four different build trees, placed by default under a `build` directory placed in the current working directory:

```
$ ls build/*
build/linux-x86_64-clang++3.9-Debug
build/linux-x86_64-clang++3.9-Release
build/linux-x86_64-gcc++6.1-Debug
build/linux-x86_64-gcc++6.1-Release
```

Each build tree is obtained by first configuring CMake with the given parameters, and then invoking `cmake --build` to build the project at once.

You can also use cmany just to simplify your cmake workflow! These two command sequences have the same effect:

typical cmake	cmany

`$ mkdir build`
`$ cd build`
`$ cmake ..`
`$ cmake --build .`
	`$ cmany b`

2.1.1 Features

- Easily configure and build many variations of your project with one simple command.
- Saves the tedious work of dealing with many build trees by hand.
- Sensible defaults: `cmany build` will create and build a single project using CMake's defaults.
- Transparently pass flags (compiler flags, processor defines or cmake cache variables) to any or all of the builds.
- Useful for build comparison and benchmarking. You can easily setup bundles of flags, aka variants.
- Useful for validating and unit-testing your project with different compilers and flags.
- Useful for creating distributions of your project.
- Avoids a full rebuild when the build type is changed. Although this feature already exists in multi-configuration cmake generators like Visual Studio, it is missing from mono-configuration generators like Unix Makefiles.
- Run arbitrary commands in every build tree or install tree.

2.1.2 More info

- Project home
- Installing
- Getting started

2.1.3 Support

- Gitter room: https://gitter.im/cmany_/community.
- send bug reports to https://github.com/biojppm/cmany/issues.
- send pull requests to https://github.com/biojppm/cmany/pulls.

2.1.4 Current status

cmany is in alpha state, under current development.

Known issues

- cmany invokes the compilers given to it to find their name and version. So far, this successfully works with Visual Studio, gcc (also with arm-linux and mips-linux counterparts), clang, icc and zapcc. However, the current implementation for guessing the name and version is fragile and may fail in some compilers which were not tested. Please submit a bug or PR if you see such a failure.
- Though cmany works in OS X with gcc and clang, using Xcode has not been tested at all. Get in touch if you are interested in getting cmany to work with Xcode.
- Pure C projects (ie not C++) should work but have not yet been tested. Some bugs may be present.

2.1.5 License

cmany is permissively licensed under the MIT license.

2.2 Installing

2.2.1 Requirements

- Python 3.6+
- CMake 3.13+
- pip

2.2.2 Installing from PyPI

cmany is available through the PyPI repository. Installing couldn't be easier:

```
$ pip install cmany
```

2.2.3 Installing from source

Installing from source is easy with pip:

```
$ git clone https://github.com/biojppm/cmany
$ cd cmany
$ pip install .
```

If you want to develop cmany, use the `-e` option for pip so that any changes you make to cmany's sources are always reflected to the installed version:

```
$ pip install -e .
```

2.2.4 Uninstalling

To uninstall cmany, just use pip:

```
$ pip uninstall cmany
```

2.3 Quick tour

2.3.1 Getting help

To get a list of available commands and help topics:

```
$ cmany help
```

To get help on a particular command (eg, `build`) or topic (eg, `quick_tour`), any of the following can be used, and they are all equivalent:

```
$ cmany help build
$ cmany h build             # help has an alias: h
$ cmany build -h
$ cmany build --help
```

Note that this text is also available as a help topic. You can read it with the help command:

```
$ cmany h quick_tour
```

2.3.2 Build

Consider a directory initially with this layout:

```
$ ls -1
CMakeLists.txt
main.cpp
```

The following command invokes CMake to configure and build this project:

```
$ cmany build .
```

Like with CMake, this will look for the CMakeLists.txt file at the given path (`.`) and place the build tree at the current working directory. If the path is omitted, CMakeLists.txt is assumed to be on the current working dir. Also, the `build` command has an alias of `b`. So the following command is exactly the same as `cmany build .`:

```
$ cmany b
```

When no compiler is specified, cmany chooses CMake's default compiler. cmany's default build type is (explicitly set to) `Release`. As an example, using g++ 6.1 in Linux x86_64, the result of the command above will be this:

```
$ tree -fi -L 2
build/
build/linux-x86_64-g++6.1-Release/
CMakeLists.txt
main.cpp

$ ls build/*/CMakeCache.txt
build/linux-x86_64-g++6.1-Release/CMakeCache.txt
```

Note that unlike CMake, cmany will not place the resulting build tree directly at the current working directory: it will instead nest it under `./build`. Each build tree name is made unique by combining the names of the operating system, architecture, compiler+version and the CMake build type.

2.3.3 Configure

It was said above that the `cmany build` command will also configure. In fact, cmany will detect when a configure step is needed. But you can just use `cmany configure` if that's only what you want to do:

```
$ cmany configure
```

Same as above: `c` is an alias to `configure`:

```
$ cmany c
```

2.3.4 Install

The `cmany install` command does what it says, and will also `configure` and `build` if needed. `i` is an alias to `install`:

```
$ cmany i
```

The install root defaults to `./install`. So assuming the project creates a single executable named `hello`, the following will result:

```
$ ls -1
CMakeLists.txt
build/
install/
main.cpp

$ tree -fi install
install/
install/linux-x86_64-g++6.1-Release/
install/linux-x86_64-g++6.1-Release/bin/
install/linux-x86_64-g++6.1-Release/bin/hello
```

2.3.5 Choosing the build type

cmany uses Release as the default build type. To set a different build type use `--build-types/-t`. The following command chooses a build type of Debug instead of Release:

```
$ cmany b -t Debug
```

If the directory is initially empty, this will be the result:

```
$ ls -1 build/*
build/linux-x86_64-g++6.1-Debug/
```

Note that the build naming scheme will cause build trees with different build types to be placed in different directories. Apart from producing a better organization of your builds, this saves you a full project rebuild when the build type changes (and the cmake generator is not a multi-config generator like MSVC).

2.3.6 Choosing the compiler

cmany defaults to CMake's default compiler. To use different compilers, use `--compilers/-c`. Each argument given here must be an executable compiler found in your path, or an absolute path to the compiler.

The following command chooses clang++ instead of CMake's default compiler:

```
$ cmany b -c clang++
```

If the directory is initially empty, this will be the result:

```
$ ls -1 build/*
build/linux-x86_64-clang++3.9-Release/
```

Read more here.

2.3.7 Choosing build/install directories

By default, cmany creates the build trees nested under a directory `build` which is created as a sibling of the `CMakeLists.txt` project file. Similarly, the install trees are nested under the `install` directory. However, you don't have to use these defaults. The following command will use `foo` for building and `bar` for installing:

```
$ cmany i -c clang++,g++ --build-dir foo --install-dir bar

$ ls -1 foo/ bar/
bar/linux-x86_64-clang++3.9-Release/
bar/linux-x86_64-g++6.1-Release/
bar/linux-x86_64-icpc16.1-Release/
foo/linux-x86_64-clang++3.9-Release/
foo/linux-x86_64-g++6.1-Release/
foo/linux-x86_64-icpc16.1-Release/
```

Note that `foo` and `bar` will still be placed under the current working directory, since they are given as relative paths. cmany also accepts absolute paths here.

2.3.8 Building many trees at once

The commands shown up to this point were only fancy wrappers for CMake. Since defaults were being used, or single arguments were given, the result for each command was a single build tree. But as its name attests to, cmany will build many trees at once by combining the build items. For example, to build both `Debug` and `Release` build types while using defaults for the remaining parameters, you can do the following (resulting in 2 build trees):

```
$ cmany b -t Debug,Release
$ ls -1 build/
build/linux-x86_64-g++6.1-Debug/
build/linux-x86_64-g++6.1-Release/
```

You can also do this for the compilers (2 build trees):

```
$ cmany b -c clang++,g++
$ ls -1 build/
build/linux-x86_64-clang++3.9-Release/
build/linux-x86_64-g++6.1-Release/
```

And you can also combine all of them (4 build trees):

```
$ cmany b -c clang++,g++ -t Debug,Release
$ ls -1 build/
build/linux-x86_64-clang++3.9-Debug/
build/linux-x86_64-clang++3.9-Release/
```

(continues on next page)

(continued from previous page)

```
build/linux-x86_64-g++6.1-Debug/
build/linux-x86_64-g++6.1-Release/
```

Another example – build using clang++,g++,icpc for Debug,Release,MinSizeRel build types (9 build trees):

```
$ cmany b -c clang++,g++,icpc -t Debug,Release,MinSizeRel
$ ls -1 build/
build/linux-x86_64-clang++3.9-Debug/
build/linux-x86_64-clang++3.9-MinSizeRel/
build/linux-x86_64-clang++3.9-Release/
build/linux-x86_64-g++6.1-Debug/
build/linux-x86_64-g++6.1-MinSizeRel/
build/linux-x86_64-g++6.1-Release/
build/linux-x86_64-icpc16.1-Debug/
build/linux-x86_64-icpc16.1-MinSizeRel/
build/linux-x86_64-icpc16.1-Release/
```

The items that can be combined by cmany are called **build items**. cmany *has the following classes of build items*:

- systems: `-s/--systems sys1[,sys2[,...]]`
- architectures (`-a/--architectures arch1[,arch2[,...]]`
- compilers (`-c/--compilers comp1[,comp2[,...]]`
- build types (`-t/--build-types btype1[,btype2[,...]]`
- variants (`-v/--variants var1[,var2[,...]]`

All of the arguments above accept a comma-separated list of items. Any omitted argument will default to a list of a single item based on the current system (for example, omitting `-s` in linux yields an implicit `-s linux` whereas in windows yields an implicit `-s windows`; or omitting `-a` in a 64 bit processor system yields an implicit `-a x86_64`).

cmany will generate builds by combining every build item with every other build item of different class. The resulting build has a name of the form `{system}-{architecture}-{compiler}-{build_type}[-{variant}]`.

A variant is a build item which brings with it a collection of flags. This allows for easy combination of these flags with other build items. But note that every build item can bring specific flags with it.

Since the number of build item combinations grows very quickly and not every combination will make sense, cmany has arguments to exclude certain combinations, either by the resulting build name (with a regex) or by the names of the items that a build would be composed of. Read more about it *here*.

2.3.9 Using flags

(*Full docs for flags here*).

You can set cmake cache variables using `--cmake-vars/-V`. For example, the following command will be the same as if `cmake -DCMAKE_VERBOSE_MAKEFILES=1 -DPROJECT_SOME_DEFINE=SOME_DEFINE=.` followed by `cmake --build` was used:

```
$ cmany b -V CMAKE_VERBOSE_MAKEFILES=1,PROJECT_SOME_DEFINE=SOME_DEFINE=
```

To add preprocessor macros, use the option `--defines/-D`:

```
$ cmany b -D MY_MACRO=1,FOO=bar,SOME_DEFINE
```

The command above has the same meaning as if `cmake -D CMAKE_CXX_FLAGS="-DMY_MACRO=1 -DFOO=bar -DSOME_DEFINE"` followed by `cmake --build` was used.

To add C++ compiler flags, use the command line option `--cxxflags/-X`. To prevent these flags being interpreted as cmany command options, use quotes or single quotes:

```
$ cmany b -X "--Wall","-O3"      # add -Wall -O3 to all builds
```

To add C compiler flags, use the option `--cflags/-C`. As with C++ flags, use quotes to escape:

```
$ cmany b -C "--Wall","-O3"
```

Using flags per build item is easy. Instead of specifying the flags at the command level as above, specify them at build item. For example:

```
$ cmany b --variant 'foo: --defines SOME_DEFINE=32 --cxxflags "-Os"' \
          --variant 'bar: --defines SOME_DEFINE=16 --cxxflags "-O2"'
```

To be clear, the `foo` variant will be compiled with the preprocessor symbol named `SOME_DEFINE` defined to 32, and will use the `-Os` C++ compiler flag. In turn, the `bar` variant will be compiled with the preprocessor symbol named `SOME_DEFINE` defined to 16, and will use the `-O2` C++ compiler flag. So instead of the build above, we now get:

```
$ ls -1 build
build/linux-x86_64-clang++3.9-Release-bar/
build/linux-x86_64-clang++3.9-Release-foo/
```

Note above the additional `-foo` and `-bar` suffixes to denote the originating variant.

You can make build items inherit the flags from other build items: add a `@` reference to the variant you want to inherit from. For example:

```
$ cmany b --variant 'foo: --defines SOME_DEFINE=32 --cxxflags "-Os"' \
          --variant 'bar: @foo --defines SOME_DEFINE=16 --cxxflags "-O2"'
```

This will result in the `bar` variant having its flags specifications as `--defines SOME_DEFINE=32 --cxxflags "-Os" --defines SOME_DEFINE=16 --cxxflags "-O2"`.

2.3.10 Cross-compiling

Cross compilation with cmake requires passing a toolchain file. cmany has the `--toolchain` option for this. This is more likely to be used as a flag of the system or architecture build type.

2.3.11 Building dependencies

cmany offers the argument `--deps path/to/extern/CMakeLists.txt` to enable building another CMake project which builds and installs the dependencies of the current project. When `--deps` is given, the external project is built for each configuration, and installed in the configuration's build directory. Use `--deps-prefix` to specify a different install directory for the external project. *Read more here*.

2.3.12 Argument reuse

Due to the way that compilation flags are accepted, the full form of a cmany command can become big. To save you from retyping the command, you can set the `CMANY_ARGS` environment variable to reuse arguments to cmany. As an experimental (and buggy) feature, you can also permanently store these options in a `cmany.yml` project file, which should be placed side by side with the project `CMakeLists.txt`. You can *find more about this here*.

2.3.13 Exporting build configurations

cmany has the command `export_vs`, which exports the build configurations to Visual Studio through the file `CMakeSettings.json` (placed side by side with the project `CMakeLists.txt`). Read this MS blog post to discover how to use the generated file.

For code-intelligence tools requiring knowledge of the compilation commands (for example, rtags, and many other tools used with emacs), cmany offers also the argument `-E/--export-compile`. This argument will instruct cmake to generate the file `compile_commands.json` (placed in each build tree).

2.4 Build items

(You can read this document with the command `cmany help build_items`).

cmany creates its builds by combining build items. There are the following classes of build items:

- systems: `-s/--systems sys1[,sys2[,...]]`
- architectures: `-a/--architectures arch1[,arch2[,...]]`
- compilers: `-c/--compilers comp1[,comp2[,...]]`
- build types: `-t/--build-types btype1[,btype2[,...]]`
- variants: `-v/--variants var1[,var2[,...]]`

All of the arguments above accept a comma-separated list of items (but note that no spaces should appear around the commmas). Repeated invokation of an argument has the same effect. For example, `-c g++,clang++` is the same as `-c g++ -c clang++`.

Any omitted argument will default to a list of a single item based on the current system (for example, omitting `-s` in linux yields an implicit `-s linux` whereas in windows yields an implicit `-s windows`; or omitting `-a` in a 64 bit processor system yields an implicit `-a x86_64`).

cmany will generate builds by combining every build item with every other build item of different class. The resulting builds have a name of the form `{system}-{architecture}-{compiler}-{build_type}[-{variant}]`.

Since the number of combinations grows very quickly and not every combination will make sense, cmany has arguments to exclude certain combinations, either by the resulting build name (with a regex) or by the names of the items that a build would be composed of. Read more about it *here*.

The following sections address each of these build items in more detail.

2.4.1 Per-item flags

To make a build item bring with it a set of properties (which can be compiler flags, macros, cmake flags, or combination exclusion arguments), use the following syntax: `'item_name: <flag_specs>...'` instead of just `item_name`. This will prompt cmany to use those flags whenever that build item is used in a build. For example, you can add a flag only to a certain build type:

```
$ cmany b --build-types Release,'Debug: --cxxflags "-Wall"'
```

Note the use of the quotes; this is necessary to have the arguments correctly parsed:

- The outer quotes (single quotes in this case) are necessary for causing the full specification of the `Debug` build type to be considered at once.

- The inner quotes around the `-Wall` flag in the `Debug` configuration are needed to prevent it from being parsed as an argument to cmany, which would cause an error.
- The order of the quotes is irrelevant to cmany. The Debug configuration could also have been specified as `"Debug: --cxxflags '-Wall'"`.
- You can also escape the quotes using the backslash character. For example, you can do `"Debug: --cxxflags \"-Wall\""` or `'Debug: --cxxflags \'-Wall\''`.

See the *cmany help on flags* to discover what flags can be used with build items. Note also that *exclusion flags* can also be used as per-item flags.

When creating the flags for the build, the order in the final compile command is the following:

1. command-level flags
2. system flags
3. architecture flags
4. compiler flags
5. build type flags
6. variant flags

So command level flags come first and variant flags come last in the compiler command line. Note that flags from later build items override those of earlier build items. For example, variant flags override flags from the build type; these in turn override flags specified with the compiler, and so on.

Inheriting per-item flags

To make a build item inherit all the flags in another build item, reference the base item name prefixed with `@`. The result is that the inheriting item will have all the flags of the base item, plus its own flags inserted at the point where the `@` reference occurs. *combination flags* are **not** inherited.

For example, note the `@foo` spec in the bar variant:

```
$ cmany b -v none \
          -v 'foo: --defines SOME_DEFINE=32 --cxxflags "-Os"' \
          -v 'bar: @foo --defines SOME_DEFINE=16 -cxxflags "-O2"'
```

With this command, bar will be now consist of `-D SOME_DEFINE=32,SOME_DEFINE=16 -X "-Os","-O2"'`.

Note: Order matters here: the place where the inheritance directive occurs is relevant. For example:

```
$ cmany b -v none \
         -v 'foo: --defines SOME_DEFINE=32 --cxxflags "-Os"' \
         -v 'bar: --defines SOME_DEFINE=16 -cxxflags "-O2" @foo'
```

will make bar consist instead of `-D SOME_DEFINE=16,SOME_DEFINE=32 -X "-O2","-Os"`. So here `SOME_DEFINE` will have with a value of 16, as opposed to 32 which will be the value in the previous example. This happens because cmany will insert foo's options right in the place where `@foo` appears.

2.4.2 Compilers

cmany defaults to CMake's default compiler. To use different compilers, use `--compilers/-c`. Each argument given here must be an executable compiler found in your path, or an absolute path to the compiler.

The following command chooses clang++ instead of CMake's default compiler:

```
$ cmany b -c clang++
```

If the directory is initially empty, this will be the result:

```
$ ls -1 build/*
build/linux-x86_64-clang++3.9-Release/
```

Note that cmany will query the compiler for a name and a version. This is for ensuring the use of different build trees for different versions of the same compiler. The logic for extracting the compiler name/version may fail in some cases, so open an issue if you see problems here.

Microsoft Visual Studio

Picking the Visual Studio version with CMake is harder than it should be. *cmany tries to make this easier* to do. For example, this will use Visual Studio 2015 **in the native architecture**:

```
$ cmany b -c vs2015
$ ls -1 build/*
build/windows-x86_64-vs2015-Release/
```

as opposed to the option required by CMake, which would be `-G "Visual Studio 15 2017 Win64"`). So if cmany is running in a 32 bit system, then the result of running the command above will be a 32 bit build instead:

```
$ cmany b -c vs2015
$ ls -1 build/*
build/windows-x86-vs2015-Release/
```

An explicit request for the target architecture may be made by appending a `_32` or `_64` suffix. For example, if Visual Studio 2017 in 32 bit mode is desired, then simply use `vs2017_32`:

```
$ cmany b -c vs2017_32
$ ls -1 build/*
build/windows-x86-vs2017-Release/
```

You can also choose the VS toolset to use in the compiler name. For example, compile with the `clang` frontend (equivalent in this case to cmake's `-T v141_clang_c2` option):

```
$ cmany b -c vs2017_clang
$ ls -1 build/*
build/windows-x86-vs2017_clang-Release/
```

cmany allows you to create any valid combination of the Visual Studio project versions (from vs2017 to vs2005), target architectures (32, 64, arm, ia64) and toolsets (from v141 to v80, with clang_c2 and xp variants). The general form for the cmany VS specification alias is:

```
<vs_project_version>[_<vs_platform_version>][_<vs_toolset_version>]
```

Note that the order must be exactly as given. Note also that the platform version or the toolset version can be omitted, in which case a sensible default will be used:

- if the platform is omitted, then the current platform will be used
- if the toolset is omitted, then the toolset of the given project version will be used.

Given the many VS versions, target architectures and toolsets, this creates hundreds of possible aliases, so read *the complete documentation for Visual Studio*.

2.4.3 Build types

cmany uses `Release` as the default build type. To set a different build type use `--build-types/-t`. The following command chooses a build type of Debug instead of Release:

```
$ cmany b -t Debug
```

If the directory is initially empty, this will be the result:

```
$ ls -1 build/*
build/linux-x86_64-g++6.1-Debug/
```

Note that the build naming scheme will cause build trees with different build types to be placed in different directories. Apart from producing a better organization of your builds, this saves you a full project rebuild when the build type changes (and the cmake generator is not a multi-config generator like MSVC).

2.4.4 Variants

cmany has **variants** for setting up *a bundle of flags* to be combined with all other build items.

A variant is a build item different from any other which uses a specific combination of flags via `--cmake-vars/-V`, `--defines/-D`, `--cxxflags/-X`, `--cflags/-C`. Like all other build items, it will be combined with other build items of different class. With the exception of the null variant, variants will usually have per-item flags.

The command option to setup a variant is `--variant/-v` and should be used as follows: `--variant 'variant_name: <variant_specs>'`. For example, assume a vanilla build:

```
$ cmany b
```

which will produce the following tree:

```
$ ls -1 build
build/linux-x86_64-clang3.9-Release/
```

If instead of this we want to produce two variants `foo` and `bar` with specific defines and compiler flags, the following command should be used:

```
$ cmany b --variant 'foo: --defines SOME_DEFINE=32 --cxxflags "-Os"' \
          --variant 'bar: --defines SOME_DEFINE=16 --cxxflags "-O2"'
```

or, in short form:

```
$ cmany b -v 'foo: -D SOME_DEFINE=32 -X "-Os"' \
          -v 'bar: -D SOME_DEFINE=16 -X "-O2"'
```

To be clear, the `foo` variant will be compiled with the preprocessor symbol named `SOME_DEFINE` defined to 32, and will use the `-Os` C++ compiler flag. In turn, the `bar` variant will be compiled with the preprocessor symbol named `SOME_DEFINE` defined to 16, and will use the `-O2` C++ compiler flag. So instead of the build above, we now get:

```
$ ls -1 build
build/linux-x86_64-clang3.9-Release-bar/
build/linux-x86_64-clang3.9-Release-foo/
```

Variants will be combined, just like compilers or build types. So applying the former two variants to the 9-build example above will result in 18 builds (3 compilers * 3 build types * 2 variants)

```
$ cmany b -c clang++,g++,icpc -t Debug,Release,MinSizeRel \
          --variant 'foo: -D SOME_DEFINE=32 -X "-Os"' \
          --variant 'bar: -D SOME_DEFINE=16 -X "-O2"'
$ ls -1 build/
build/linux-x86_64-clang3.9-Debug-bar/
build/linux-x86_64-clang3.9-Debug-foo/
build/linux-x86_64-clang3.9-MinSizeRel-bar/
build/linux-x86_64-clang3.9-MinSizeRel-foo/
build/linux-x86_64-clang3.9-Release-bar/
build/linux-x86_64-clang3.9-Release-foo/
build/linux-x86_64-gcc6.1-Debug-bar/
build/linux-x86_64-gcc6.1-Debug-foo/
build/linux-x86_64-gcc6.1-MinSizeRel-bar/
build/linux-x86_64-gcc6.1-MinSizeRel-foo/
build/linux-x86_64-gcc6.1-Release-bar/
build/linux-x86_64-gcc6.1-Release-foo/
build/linux-x86_64-icc16.1-Debug-bar/
build/linux-x86_64-icc16.1-Debug-foo/
build/linux-x86_64-icc16.1-MinSizeRel-bar/
build/linux-x86_64-icc16.1-MinSizeRel-foo/
build/linux-x86_64-icc16.1-Release-bar/
build/linux-x86_64-icc16.1-Release-foo/
```

Note that like any other build item `--variant/-v` also accepts comma-separated arguments:

```
$ cmany b -c clang++,g++,icpc -t Debug,Release,MinSizeRel \
          --variant 'foo: -D SOME_DEFINE=32 -X "-Os"','bar: -D SOME_DEFINE=16 -X "-O2"
↪'
```

Null variant

cmany will combine only the variants it is given. Notice above that the basic (variant-less) build `linux-x86_64-clang3.9-Debug` is not there. To retain the basic build without a variant suffix use the special name `none`:

```
$ cmany b -v none \
          -v 'foo: -D SOME_DEFINE=32 -X "-Os"' \
          -v 'bar: -D SOME_DEFINE=16 -X "-O2"'
$ ls -1 build
build/linux-x86_64-clang3.9-Release/
build/linux-x86_64-clang3.9-Release-bar/
build/linux-x86_64-clang3.9-Release-foo/
```

You can add flags to the `none` variant as well, and use inheritance at will:

```
$ cmany b -v 'none: -X "-Wall"' \
          -v 'foo: @none -D SOME_DEFINE=32 -X "-Os"' \
          -v 'bar: @none -D SOME_DEFINE=16 -X "-O2"'
```

2.4.5 Systems

The argument to specify system build items is `-s/--systems`. Systems are a special build item. For example, if you are on linux, omitting `-s` will default to `-s linux`, so using `-s` it is not really needed. But compiling for a different target system qualifies as cross compilation and requires a cmake toolchain file (for which cmany offers the `-T/--toolchain` flag).

Usually, a toolchain also fixes the compiler and maybe the processor architecture. So most of the time, whenever `-s/--systems` is used, it will bring with it a toolchain and will also require *exclusion arguments* to prevent combination of the target system with compilers or architectures different from those of the toolchain.

For example, consider this linux ARM gnueabihf cmake toolchain:

```
# arm-linux-gnueabihf.cmake

set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_VERSION 1)

set(CMAKE_C_COMPILER   /usr/bin/arm-linux-gnueabihf-gcc)
set(CMAKE_CXX_COMPILER /usr/bin/arm-linux-gnueabihf-g++)
set(CMAKE_STRIP        /usr/bin/arm-linux-gnueabihf-strip)

set(CMAKE_FIND_ROOT_PATH  /usr/arm-linux-gnueabihf)

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

(If you are using Ubuntu you can install these tools with `sudo apt-get install binutils-arm-linux-gnueabihf g++-arm-linux-gnueabihf`). When you want to compile your project only for linux OS and (armv5,armv7) architectures, this would be the command line:

```
$ cmany b --systems 'linux: -T arm-linux-gnueabihf.cmake' \
          --architectures 'armv5: -X "-march=armv5"' \
          --architectures 'armv7: -X "-march=armv7"'
```

which will produce the following builds:

```
linux-armv5-arm-linux-gnueabihf-g++5.4-Release
linux-armv7-arm-linux-gnueabihf-g++5.4-Release
```

If you want to build at the same time for x86 and x86_64, this command could be used (note the use of `--exclude-builds`):

```
$ cmany b --systems linux \
          --architectures x86,x86_64 \
          --systems 'linux_arm: -T arm-linux-gnueabihf.cmake -xa x86,x86_64' \
          --architectures 'armv5: -X "-march=armv5"' \
          --architectures 'armv7: -X "-march=armv7"' \
          --exclude-builds 'linux_arm-.*x86','linux-.*arm'
```

which produces the following builds:

```
linux-x86-g++5.4-Release
linux-x86_64-g++5.4-Release
linux_arm-armv5-arm-linux-gnueabihf-g++5.4-Release
linux_arm-armv7-arm-linux-gnueabihf-g++5.4-Release
```

The arm builds for linux now use a system named linux_arm to prevent ambiguity with the native `linux` system. The `--exclude-builds` is used to prevent the arm toolchain from being combined with x86 or x86_64 architectures and also to prevent the arm architectures from being used without a toolchain.

2.4.6 Architectures

Architectures are specified with the argument `-a/--architectures`. When the architecture argument is omitted, cmany will silently default to `-a <current_architecture>`.

Like with `-s/--systems`, architectures have some special properties:

- When in a 64 bit system, asking for a `x86` architecture when the compiler is a gcc-like compiler (ie, gcc, clang, icc, zapcc or similar) will result in the the compiler flag `-m32` being added to `CMAKE_CXX_FLAGS` and `CMAKE_C_FLAGS`.
- Conversely, when in a 32 bit system, asking for a `x86_64` architecture with a gcc-like compiler will result in the flag `-m64` being added to `CMAKE_CXX_FLAGS` and `CMAKE_C_FLAGS`.
- When *using Visual Studio*, you can select the compiler and architecture at once. For example, using `-c vs2015_32,vs2015_64` will compile both for `x86` and `x86_64`, and is equivalent to `-c vs2015 -a x86,x86_64`.

Architectures other than x86/x86_64, as with `-s/--systems`, `-a/--architectures` will usually need cross-compilation and thus require toolchains. In fact the, command used in the example of the *Systems* section can be simplified if the toolchain is specified by architecture instead of the system, and the variants instead provide the `-m` compiler flags:

```
$ cmany b --architectures x86,x86_64,'arm: -T arm-linux-gnueabihf.cmake' \
          --variants 'none: -xa arm' \
          --variants 'armv5: -X "-march=armv5" -ia arm' \
          --variants 'armv7: -X "-march=armv7" -ia arm'
```

The `-xa` and the `-ia` arguments above are the short forms of respectively the *exclusion arguments* `--exclude-architectures` and `--include-architectures`. They are needed to prevent combination of the arm variants with the x86 architectures and vice-versa. The resulting builds are exactly the same as in that example, but are now named differently because of the variants:

```
linux-x86-g++5.4-Release
linux-x86_64-g++5.4-Release
linux-arm-arm-linux-gnueabihf-g++5.4-Release-armv5
linux-arm-arm-linux-gnueabihf-g++5.4-Release-armv7
```

2.5 Excluding builds

cmany accepts arguments for excluding certain combinations of build items. These arguments specify combination exclusion rules which apply either to a build's name or to the individual build items of which the build is composed.

When multiple combination arguments are given, they are processed in the order in which they are given. A build is then included if it successfully matches every argument.

Be aware that for added convenience, cmany offers the commands `show_build_names` and `show_builds`, which are useful for testing the input arguments.

2.5.1 Excluding builds by item name

To exclude a build based on its composing items, cmany offers the following arguments, which should be fairly self-explaining:

- **systems (`-s/--systems`):**
  - `-xs/--exclude-systems sys1[,sys2[,...]]`
  - `-is/--include-systems sys1[,sys2[,...]]`
- **architectures (`-a/--architectures`):**
  - `-xa/--exclude-architectures arch1[,arch2[,...]]`
  - `-ia/--include-architectures arch1[,arch2[,...]]`
- **compilers (`-c/--compilers`):**
  - `-xc/--exclude-compilers comp1[,comp2[,...]]`
  - `-ic/--include-compilers comp1[,comp2[,...]]`
- **build types (`-t/--build-types`):**
  - `-xt/--exclude-build-types btype1[,btype2[,...]]`
  - `-it/--include-build-types btype1[,btype2[,...]]`
- **variants (`-v/--variants`):**
  - `-xv/--exclude-variants var1[,var2[,...]]`
  - `-iv/--include-variants var1[,var2[,...]]`

Each argument receives a comma-separated list of item names. These names are matched literally to the name of any item of the same type, and the prospective build (which would combine those items) is included or excluding according to the rule.

Also note the mnemonic for the build item arguments: the letter you use to specify a build item is the letter after the short form of the include or exclude flag.

2.5.2 Excluding builds by build name

These are the arguments to prevent a build by matching against the build's final name:

- `-xb/--exclude-builds rule1[,rule2[,...]]`: excludes a build if its name matches **any** of the rules
- `-ib/--include-builds rule1[,rule2[,...]]`: includes a build only if its name matches **any** of the rules
- `-xba/--exclude-builds-all rule1[,rule2[,...]]`: excludes a build if its name matches **all** of the rules
- `-iba/--include-builds-all rule1[,rule2[,...]]`: includes a build only if its name matches **all** of the rules

As noted above, each argument accepts a comma-separated list of Python regular expressions that will be used as matching rules to each build name. A build is included only if its name successfully matches every combination argument. Note that the form of a build's name is `{system}-{architecture}-{compiler}-{build_type}[-{variant}]`. Note also the presence of the hyphen separating the build items; it can be used to distinguish between similarly named items such as `x86` and `x86_64`.

The rules do not need to be regular expressions: passing the full names of the builds to the argument works as expected.

2.5.3 Examples

As a first example, consider this command which addresses 12 builds by combining 2 architectures, 2 build types and 3 variants:

```
$ cmany show_build_names -a x86,x86_64 -t Debug,Release \
                         -v none,'foo: -D FOO','bar: -D BAR'
linux-x86-g++5.4-Debug
linux-x86-g++5.4-Debug-foo
linux-x86-g++5.4-Debug-bar
linux-x86-g++5.4-Release
linux-x86-g++5.4-Release-foo
linux-x86-g++5.4-Release-bar
linux-x86_64-g++5.4-Debug
linux-x86_64-g++5.4-Debug-foo
linux-x86_64-g++5.4-Debug-bar
linux-x86_64-g++5.4-Release
linux-x86_64-g++5.4-Release-foo
linux-x86_64-g++5.4-Release-bar
```

Now, if we want to exclude `foo` variants of the `x86` architecture, we can use:

```
$ cmany show_build_names -a x86,x86_64 -t Debug,Release \
                         -v none,'foo: -D FOO','bar: -D BAR' \
                         --exclude-builds '.*x86-.*foo'
linux-x86-g++5.4-Debug
linux-x86-g++5.4-Debug-bar     # NOTE: no x86 foo variant
linux-x86-g++5.4-Release
linux-x86-g++5.4-Release-bar   # NOTE: no x86 foo variant
linux-x86_64-g++5.4-Debug
linux-x86_64-g++5.4-Debug-foo
linux-x86_64-g++5.4-Debug-bar
linux-x86_64-g++5.4-Release
linux-x86_64-g++5.4-Release-foo
linux-x86_64-g++5.4-Release-bar
```

Note the hyphen appearing in the regular expression passed to `--exclude-builds '.*x86-.*foo'`. This prevents it from matching `x86_64`. As noted above, the name of the build is obtained by separating the build items of which it is composed with an hyphen. If this regular expression did not have the hyphen and was instead `.*x86.*foo`, then it would match both `x86` and `x86_64`, with the result that no builds would contain the `foo` variant.

For the previous example, it is actually easier to have the `foo` variant directly exclude the `x86` architecture. The result is exactly the same:

```
$ cmany show_builds -a x86,x86_64 -t Debug,Release \
                    -v none,'foo: -D FOO -xa x86','bar: -D BAR'
```

You could instead have the `x86` architecture exclude the `foo` variant, with the same result:

```
$ cmany show_builds -a 'x86: -xv foo',x86_64 -t Debug,Release \
                    -v none,'foo: -D FOO','bar: -D BAR' \
```

The logical opposite of `--exclude-builds` is naturally `--include-builds`:

```
$ cmany show_builds -a x86,x86_64 -t Debug,Release \
                    -v none,'foo: -D FOO','bar: -D BAR' \
                    --include-builds '.*x86-.*foo'
linux-x86-g++5.4-Debug-foo
linux-x86-g++5.4-Release-foo
```

This can also be done with the following command:

```
$ cmany show_builds -a x86,x86_64 -t Debug,Release \
                   -v none,'foo: -D FOO','bar: -D BAR' \
                   -ia x86 -iv foo
```

If you are wondering about the usefulness of the `-i*/--include` arguments, consider that the compile-edit loop is usually repeated many times. Being that the arguments to cmany usually come to a certain degree of complexity (something which Project mode also addresses), rewriting them every time is something we would like to avoid. So when you want to narrow down your previous command (or your project setup) just to a certain combination of builds, the `--include-*` arguments usually come in very handy.

Like all the arguments above, item combination arguments can be used both at the cmany command level or at each build item level.

You may have noticed that it does not make much sense to provide the `--include-*` arguments in a build item specification, as combination is implied for every build item. However, being able to use these at the scope of the command is certainly useful, either as a form of saving extensive editing when reusing complicated cmany commands (for example in shell sessions), or with Project mode.

2.6 Flags

cmany allows you to transparently pass flags to all or some of the builds. These flags can be compiler flags, preprocessor defines or CMake cache variables. Specifying flags is an important cmany usage pattern, which is used also when specifying *Per-item flags*.

Note: The examples below apply the flags across the board to all the individual builds produced with the cmany command. For example, this will add `-Wall` to all of the 9 resulting builds:

```
$ cmany b --compilers clang++,g++,icpc \
          --build-types Debug,Release,MinSizeRel \
          --cxxflags "-Wall"
```

If you want to add flags only to certain builds, it's cool! One of the main motivations of cmany is being able to easily do that, and it offers two different solutions. You can *use variants*. You can also set flags per *Per-item flags*, so that when one specific build item (such as a compiler or operating system) is used, then the flags are also used in the resulting build.

2.6.1 CMake cache variables

To set cmake cache variables use `--cmake-vars/-V`. For example:

```
$ cmany b -V CMAKE_VERBOSE_MAKEFILE=1,PROJECT_FOO=BAR
```

This is equivalent to the following sequence of commands:

```
$ cmake -D CMAKE_VERBOSE_MAKEFILE=1 -D PROJECT_FOO=BAR ../..
$ cmake --build .
```

Note the use of the comma to separate the variables. This is for consistency with the rest of the cmany options, namely (for example) those selecting compilers or build types. You can also use separate invocations:

```
$ cmany b -V CMAKE_VERBOSE_MAKEFILE=1 -V PROJECT_FOO=BAR
```

which will have the same result as above.

When specified as above, CMake cache variables will apply to all builds. If you want to add CMake cache variables only to certain builds by associating them with particular build items, you can use the pattern `'item_name: <flags...>'`. For example, this will enable the cmake variable `ENABLE_X64_ASM` only when `x86_64` is used:

```
$ cmany b -s x86,'x86_64: -V ENABLE_X64_ASM=1'
```

CMake's cache variables are typed, and this sometimes can affect subtly the outcome of a build. cmany will try to guess the type variable type (ie, whether it is a `BOOL`, `PATH`, `FILEPATH` or `STRING`), based on its name and value. You can override this behaviour by using the standard CMake syntax for specifying the variable type:

```
$ cmany b -V CMAKE_VERBOSE_MAKEFILE:BOOL=1,PROJECT_FOO:STRING=BAR
```

2.6.2 Preprocessor macros

To add preprocessor macros, use the option `--defines/-D`:

```
$ cmany b -D MY_MACRO=1,FOO=BAR
```

The command above has the same meaning of:

```
$ cmake -D CMAKE_CXX_FLAGS="-D MYMACRO=1 -D FOO=BAR" ../..
$ cmake --build .
```

Note also that `--defines/-D` can be used repeatedly, with the same result:

```
$ cmany b -D MY_MACRO=1 -D FOO=BAR
```

To add macros only to certain build items, use the `'item_name: <flags...>'` pattern. For example, this will add the macro `XDEBUG` only to the `Debug` build type:

```
$ cmany b -t Release,'Debug: -D XDEBUG'
```

2.6.3 C++ compiler flags

To add C++ compiler flags to a cmany build, use the option `--cxxflags/-X`. To prevent these flags being interpreted as cmany command options, use quotes or single quotes (or flag aliases, see below):

```
$ cmany b -X "-Wall","-O3"      # add -Wall -O3 to all builds
```

The command above has the same meaning of:

```
$ cmake -D CMAKE_CXX_FLAGS="-Wall -O3" ..
$ cmake --build .
```

Note that `--cxxflags/-X` can be used repeatedly, with the same result:

```
$ cmany b -X "-Wall" -X "-O3"
```

To add C++ compiler flags only to certain build items, use the `'item_name: <flags...>'` pattern. For example, this will add the `-ffast-math` flag only to the `Release` build type:

```
$ cmany b -t 'Release: -X "-ffast-math"',Debug
```

2.6.4 C compiler flags

To add C compiler flags, use the option `--cflags/-C`. As with C++ flags, to prevent interpretation by cmany, use single or double quotes to escape the compiler flags (or use flag aliases, see below):

```
$ cmany b -C "-Wall","-O3"
```

The command above has the same meaning of:

```
$ cmake -D CMAKE_C_FLAGS="-Wall -O3" ..
$ cmake --build .
```

Note that `--cflags/-C` can be chained, with the same result:

```
$ cmany b -C "-Wall" -C "-O3"
```

To add C compiler flags only to certain build items, use the `'item_name: <flags...>'` pattern. For example, this will add the `-ffast-math` flag only to the `Release` build type:

```
$ cmany b -t 'Release: -C -ffast-math',Debug
```

2.6.5 Linker flags

For now, cmany has no explicit support for linker flags. But you can set linker flags through the cmake cache variable mechanism:

```
$ cmany b -V '-DCMAKE_LINKER_FLAGS="....linker flags...."'
```

You may also have noticed that CMake cache variables will allow you to specify macros and compiler flags as well via `-DCMAKE_CXX_FLAGS=...`. Yes, that's right, you can also do that. But not only is it less verbose when passing macros and flags through `--defines/--cflags/--cxxflags`: there is a strong reason to prefer it this way: **flag aliases**, introduced below.

2.6.6 Flag aliases

For simplicity of use, cmany comes with a predefined set of flag aliases which you can use. A flag alias is a name which maps to specific flags for each compiler. For example, if you want to enable maximum warnings there is the `wall` alias (shown here in the yml markup which cmany uses to define it):

```
wall:
    desc: turn on all warnings
    gcc,clang,icc: -Wall
    vs: /Wall
```

or eg the `avx` alias if you want to enable AVX SIMD processing:

```
avx:
    desc: enable AVX instructions
    gcc,clang,icc: -mavx
    vs: /arch:avx
```

This allows use of the aliases instead of the flags directly, thus insulating you from differences between compilers. Using the aliases will also produce easier commands, because less quoting is needed to prevent the flags from being read by the shell. For example, the following command will translate to `g++ -mavx -Wall` with gcc, clang or icc, but with Visual Studio it will translate instead to `cl.exe /Wall /arch:avx`:

```
$ cmany b --cxxflags avx,wall
```

As a comparison, direct use of the flags would result in these commands:

```
$ cmany b --cxxflags '-Wall','-mavx'          # for gcc
$ cmany b --cxxflags '/Wall','/arch:avx'      # for VS
```

Note that flag aliases are translated only when they are given through `--cxxflags/-cflags`. Do not use aliases with `--cmake-vars CMAKE_CXX_FLAGS=...`, as cmany will not translate them there.

Built-in flag aliases

cmany provides some built-in flag aliases to simplify working with different compilers at the same time. Currently, you can see them in the file `conf/cmany.yml` (see the current version at github).

Defining more flag aliases

Being able to define your own flag aliases is in the roadmap. For now, you can submit PRs for adding aliases.

2.6.7 Toolchains

To use cmake toolchain use the option `-T/--toolchain`. Usually, this will be done inside a systems build item, `-s/--systems`; see the *Systems* section in the *build items document*.

2.6.8 Build exclusion arguments

Note that *arguments for excluding builds* can be used wherever flag arguments can be used. This makes it easier to declare incompatibility between build items. See *an example in this help page*.

2.7 Using cmany with Visual Studio

2.7.1 TL;DR

The general form for choosing the Visual Studio compiler version with cmany is:

```
<vs_version>[_<architecture>][_<vs_toolset>]
```

where:

- <vs_version> is one of vs2019, vs2017, vs2015, vs2013, vs2012, vs2010, vs2008 or vs2005
- <architecture> is one of 32, 64, arm, ia64. Defaults to the native architecture when omitted.
- **as for <vs_toolset>:**
  - when omitted, uses the default toolset of the chosen VS version
  - when either clang or xp is given, uses the default toolset of the chosen VS version for either clang or xp
  - **otherwise, one of:**
    * from vs2019: v142, v142_clang, v142_clang_c2, v142_xp
    * from vs2017: v141, v141_clang, v141_clang_c2, v141_xp
    * from vs2015: v140, v140_clang, v140_clang_c2, v140_xp
    * from vs2013: v120, v120_xp
    * from vs2012: v110, v110_xp
    * from vs2010: v100, v100_xp
    * from vs2008: v90, v90_xp
    * from vs2008: v80

Note that not every combination is valid:

- you cannot use toolsets newer than your chosen VS version
- arm is not available in VS versions older than VS2012
- ia64 is not available in VS versions later than VS2012
- clang is not available in versions older than VS2015.

Note: cmany defaults to using the native architecture when no target architecture is specified. For example, the command `cmany b -c vs2015` will produce a 64 bit build when it is run in a 64 bit system; but when it is run in a 32 bit system it will produce a 32 bit build.

This contrasts with cmake: `cmake -G "Visual Studio 15 2017" ../..` will produce a 32 bit build in any system.

2.7.2 VS alias examples

- vs2015: depending on the native architecture, same as `cmake -G "Visual Studio 14 2015"` **OR** `cmake -G "Visual Studio 14 2015 Win64"`
- vs2015_clang: ditto, but use the v140_clang_c2 toolset
- vs2015_arm: same as `cmake -G "Visual Studio 14 2015 ARM"`
- vs2015_32_clang: same as `cmake -G "Visual Studio 14 2015" -T v140_clang_c2`
- vs2015_64_clang: same as `cmake -G "Visual Studio 14 2015 Win64" -T v140_clang_c2`
- vs2017_64_v140: same as `cmake -G "Visual Studio 15 2017 Win64" -T v140`. This will generate a VS2017 solution for x86_64 which is compiled with the v140 toolset which is from VS2015.

2.7.3 Complete explanation

Visual Studio (VS) has an awkward numbering system: the year and the version number (it also has a different version number for the `cl.exe` compiler, but fortunately we have no need to deal with that here). Sadly, and confusingly, these have similar values, and cmake forces you to deal with this by having to explicitly state the full Visual Studio version and year.

cmany tries to help in this situation by making it easier to specify which Visual Studio IDE version and toolset version to use via a simple naming scheme. For example, this will create two VS projects **in the native architecture** (eg, x86_64), but with different versions:

```
$ cmany b -c vs2015,vs2017,vs2019
```

As usual, this would be the result the command:

```
$ ls -1 build/*
build/windows-x86_64-vs2015-Release/
build/windows-x86_64-vs2017-Release/
build/windows-x86_64-vs2019-Release/
```

For comparison, to achieve this with direct use of CMake, the equivalent (bash) commands would have to be:

```
$ mkdir build/windows-x86_64-vs2015-Release
$ mkdir build/windows-x86_64-vs2017-Release
$ mkdir build/windows-x86_64-vs2019-Release
$ cd build/windows-x86_64-vs2015-Release
$ cmake -G "Visual Studio 14 2015 Win64" ../..
$ cd -
$ cd build/windows-x86_64-vs2017-Release
$ cmake -G "Visual Studio 15 2017 Win64" ../..
$ cd -
$ cd build/windows-x86_64-vs2019-Release
$ cmake -G "Visual Studio 15 2019 Win64" ../..
$ cd -
```

A further point of confusion with Visual Studio is that a given VS solution version actually consists of two separate items: a) the compiler version (the toolset) and b) the IDE version. When you run eg `cmake -G "Visual Studio 14 2015"` you get a solution for editing your code with the 2015 IDE, *and* your code will be compiled with Visual Studio 2015's default toolset which is `v140`. But you do not have to use this toolset when using the VS 2015 IDE. For example, you can generate a 2015 solution which uses the 2013 toolset, `v120`. To do that in CMake, the command would be `cmake -G "Visual Studio 14 2015" -T v120`.

The issue of the Visual Studio toolsets was made less of a corner case with the recent addition (since VS 2015) of a clang frontend, which is convenient for multi-compiler validation of a project's code. So making it easier to specify non-standard VS toolsets was also from the outset a requirement for cmany.

2.7.4 Aliasing scheme

cmany allows you to create any valid combination of the Visual Studio project versions (from vs2017 to vs2005), target architectures (32, 64, arm, ia64) and toolsets (from v141 to v80, with clang_c2 and xp variants). The general form for the cmany VS alias is:

```
<vs_version>[_<vs_platform>][_<vs_toolset>]
```

Note that the platform or the toolset can be omitted, in which case a sensible default will be used:

- if the platform is omitted, then the current platform will be used

- if the toolset is omitted, then the toolset of the given project version will be used.

Note also that the order must be exactly as given: first the VS version, then the platform, then the toolset. For example, when the platform is omitted, then the alias should have the following form:

```
<vs_version>[_<vs_toolset>]
```

When the toolset is omitted, then the alias should have the form:

```
<vs_version>[_<vs_platform>]
```

If both the architecture and platform are omitted, then the alias becomes simply:

```
<vs_version>
```

Check the *VS alias examples* for seeing this scheme at work. The next subsections give a complete enumeration of the possible values for each item in the triplet.

Visual Studio versions

Here's a correspondence between the basic cmany names and the cmake specification. CMake simultaneously specifies the VS version and the target architecture.

cmany	cmake	target architecture
vs2019	Visual Studio 16 2019 ???	native, ie 32 or 64 bits
vs2019_32	Visual Studio 16 2019	x86
vs2019_64	Visual Studio 16 2019 Win64	x86_64
vs2019_arm	Visual Studio 16 2019 ARM	arm
vs2017	Visual Studio 15 2017 ???	native, ie 32 or 64 bits
vs2017_32	Visual Studio 15 2017	x86
vs2017_64	Visual Studio 15 2017 Win64	x86_64
vs2017_arm	Visual Studio 15 2017 ARM	arm
vs2015	Visual Studio 14 2015 ???	native, ie 32 or 64 bits
vs2015_32	Visual Studio 14 2015	x86
vs2015_64	Visual Studio 14 2015 Win64	x86_64
vs2015_arm	Visual Studio 14 2015 ARM	arm
vs2013	Visual Studio 12 2013 ???	native, ie 32 or 64 bits
vs2013_32	Visual Studio 12 2013	x86
vs2013_64	Visual Studio 12 2013 Win64	x86_64
vs2013_arm	Visual Studio 12 2013 ARM	arm
vs2012	Visual Studio 11 2012 ???	native, ie 32 or 64 bits
vs2012_32	Visual Studio 11 2012	x86
vs2012_64	Visual Studio 11 2012 Win64	x86_64
vs2012_arm	Visual Studio 11 2012 ARM	arm
vs2010	Visual Studio 10 2010 ???	native, ie 32 or 64 bits
vs2010_32	Visual Studio 10 2010	x86
vs2010_64	Visual Studio 10 2010 Win64	x86_64
vs2010_ia64	Visual Studio 10 2010 IA64	ia64
vs2008	Visual Studio 9 2008 ???	native, ie 32 or 64 bits
vs2008_32	Visual Studio 9 2008	x86
vs2008_64	Visual Studio 9 2008 Win64	x86_64
vs2008_ia64	Visual Studio 9 2008 IA64	ia64

Continued on next page

Table 1 – continued from previous page

cmany	cmake	target architecture
`vs2005`	`Visual Studio 8 2005 ???`	native, ie 32 or 64 bits
`vs2005_32`	`Visual Studio 8 2005`	x86
`vs2005_64`	`Visual Studio 8 2005 Win64`	x86_64

Target architecture

From the list above, it is easy to gather the list of valid architecture names in cmany's VS aliasing scheme:

- `32`
- `64`
- `arm`
- `ia64`

Visual Studio toolset

Here's the list of valid Visual Studio toolsets:

- `vs2019` compiler toolsets: `v142`, `v142_clang_c2`, `v142_xp`
- `vs2017` compiler toolsets: `v141`, `v141_clang_c2`, `v141_xp`
- `vs2015` compiler toolsets: `v140`, `v140_clang_c2`, `v140_xp`
- `vs2013` compiler toolsets: `v120`, `v120_xp`
- `vs2012` compiler toolsets: `v110`, `v110_xp`
- `vs2010` compiler toolsets: `v100`, `v100_xp`
- `vs2008` compiler toolsets: `v90`, `v90_xp`
- `vs2005` compiler toolsets: `v80`,

cmany allows several shorter forms for specifying some of these toolsets:

- the default toolset can be omitted. For example, `vs2017` is exactly the same as `vs2017_v141`, and `vs2013` is exactly the same as `vs2013_v120`
- the clang toolset can be shortened to `clang` instead of `clang_c2`. Also, omitting the version from a clang toolset will default to the current VS version's toolset. So for example, `vs2015_clang` or `vs2015_clang_c2` are the same as `vs2015_v140_clang_c2`.
- the xp toolset has the same ommission behaviour as clang. For example, `vs2015_xp` is the same as `vs2015_v140_xp`.

2.7.5 Alias list

It is easy to see that combining the VS solution version, target architecture and toolsets above creates hundreds of different possibilities. This section shows what each of them mean. (If you find any errors, please submit a bug or PR).

VS2019

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2019	16 2019	(native)	v142
vs2019_clang	16 2019	(native)	v142_clang_c2
vs2019_xp	16 2019	(native)	v142_xp
vs2019_v142	16 2019	(native)	v142
vs2019_v142_xp	16 2019	(native)	v142_xp
vs2019_v142_clang	16 2019	(native)	v142_clang_c2
vs2019_v141	16 2019	(native)	v141
vs2019_v141_xp	16 2019	(native)	v141_xp
vs2019_v141_clang	16 2019	(native)	v141_clang_c2
vs2019_v140	16 2019	(native)	v140
vs2019_v140_xp	16 2019	(native)	v140_xp
vs2019_v140_clang	16 2019	(native)	v140_clang_c2
vs2019_v120	16 2019	(native)	v120
vs2019_v120_xp	16 2019	(native)	v120_xp
vs2019_v110	16 2019	(native)	v110
vs2019_v110_xp	16 2019	(native)	v110_xp
vs2019_v100	16 2019	(native)	v100
vs2019_v100_xp	16 2019	(native)	v100_xp
vs2019_v90	16 2019	(native)	v90
vs2019_v90_xp	16 2019	(native)	v90_xp
vs2019_v80	16 2019	(native)	v80
vs2019_32	16 2019	x86	v142
vs2019_32_clang	16 2019	x86	v142_clang_c2
vs2019_32_xp	16 2019	x86	v142_xp
vs2019_32_v142	16 2019	x86	v142
vs2019_32_v142_xp	16 2019	x86	v142_xp
vs2019_32_v142_clang	16 2019	x86	v142_clang_c2
vs2019_32_v141	16 2019	x86	v141
vs2019_32_v141_xp	16 2019	x86	v141_xp
vs2019_32_v141_clang	16 2019	x86	v141_clang_c2
vs2019_32_v140	16 2019	x86	v140
vs2019_32_v140_xp	16 2019	x86	v140_xp
vs2019_32_v140_clang	16 2019	x86	v140_clang_c2
vs2019_32_v120	16 2019	x86	v120
vs2019_32_v120_xp	16 2019	x86	v120_xp
vs2019_32_v110	16 2019	x86	v110
vs2019_32_v110_xp	16 2019	x86	v110_xp
vs2019_32_v100	16 2019	x86	v100
vs2019_32_v100_xp	16 2019	x86	v100_xp
vs2019_32_v90	16 2019	x86	v90
vs2019_32_v90_xp	16 2019	x86	v90_xp
vs2019_32_v80	16 2019	x86	v80
vs2019_64	16 2019	x86_64	v142
vs2019_64_clang	16 2019	x86_64	v142_clang_c2
vs2019_64_xp	16 2019	x86_64	v142_xp
vs2019_64_v142	16 2019	x86_64	v142
vs2019_64_v142_xp	16 2019	x86_64	v142_xp
vs2019_64_v142_clang	16 2019	x86_64	v142_clang_c2
vs2019_64_v141	16 2019	x86_64	v141
vs2019_64_v141_xp	16 2019	x86_64	v141_xp

Continued on next page

Table 2 – continued from previous page

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2019_64_v141_clang	16 2019	x86_64	v141_clang_c2
vs2019_64_v140	16 2019	x86_64	v140
vs2019_64_v140_xp	16 2019	x86_64	v140_xp
vs2019_64_v140_clang	16 2019	x86_64	v140_clang_c2
vs2019_64_v120	16 2019	x86_64	v120
vs2019_64_v120_xp	16 2019	x86_64	v120_xp
vs2019_64_v110	16 2019	x86_64	v110
vs2019_64_v110_xp	16 2019	x86_64	v110_xp
vs2019_64_v100	16 2019	x86_64	v100
vs2019_64_v100_xp	16 2019	x86_64	v100_xp
vs2019_64_v90	16 2019	x86_64	v90
vs2019_64_v90_xp	16 2019	x86_64	v90_xp
vs2019_64_v80	16 2019	x86_64	v80
vs2019_arm	16 2019	arm	v142
vs2019_arm_clang	16 2019	arm	v142_clang_c2
vs2019_arm_v142	16 2019	arm	v142
vs2019_arm_v142_clang	16 2019	arm	v142_clang_c2
vs2019_arm_v141	16 2019	arm	v141
vs2019_arm_v141_clang	16 2019	arm	v141_clang_c2
vs2019_arm_v140	16 2019	arm	v140
vs2019_arm_v140_clang	16 2019	arm	v140_clang_c2
vs2019_arm_v120	16 2019	arm	v120
vs2019_arm_v110	16 2019	arm	v110
vs2019_arm_v100	16 2019	arm	v100

VS2017

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2017	15 2017	(native)	v141
vs2017_clang	15 2017	(native)	v141_clang_c2
vs2017_xp	15 2017	(native)	v141_xp
vs2017_v141	15 2017	(native)	v141
vs2017_v141_xp	15 2017	(native)	v141_xp
vs2017_v141_clang	15 2017	(native)	v141_clang_c2
vs2017_v140	15 2017	(native)	v140
vs2017_v140_xp	15 2017	(native)	v140_xp
vs2017_v140_clang	15 2017	(native)	v140_clang_c2
vs2017_v120	15 2017	(native)	v120
vs2017_v120_xp	15 2017	(native)	v120_xp
vs2017_v110	15 2017	(native)	v110
vs2017_v110_xp	15 2017	(native)	v110_xp
vs2017_v100	15 2017	(native)	v100
vs2017_v100_xp	15 2017	(native)	v100_xp
vs2017_v90	15 2017	(native)	v90
vs2017_v90_xp	15 2017	(native)	v90_xp
vs2017_v80	15 2017	(native)	v80
vs2017_32	15 2017	x86	v141
vs2017_32_clang	15 2017	x86	v141_clang_c2

Continued on next page

Table 3 – continued from previous page

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2017_32_xp	15 2017	x86	v141_xp
vs2017_32_v141	15 2017	x86	v141
vs2017_32_v141_xp	15 2017	x86	v141_xp
vs2017_32_v141_clang	15 2017	x86	v141_clang_c2
vs2017_32_v140	15 2017	x86	v140
vs2017_32_v140_xp	15 2017	x86	v140_xp
vs2017_32_v140_clang	15 2017	x86	v140_clang_c2
vs2017_32_v120	15 2017	x86	v120
vs2017_32_v120_xp	15 2017	x86	v120_xp
vs2017_32_v110	15 2017	x86	v110
vs2017_32_v110_xp	15 2017	x86	v110_xp
vs2017_32_v100	15 2017	x86	v100
vs2017_32_v100_xp	15 2017	x86	v100_xp
vs2017_32_v90	15 2017	x86	v90
vs2017_32_v90_xp	15 2017	x86	v90_xp
vs2017_32_v80	15 2017	x86	v80
vs2017_64	15 2017	x86_64	v141
vs2017_64_clang	15 2017	x86_64	v141_clang_c2
vs2017_64_xp	15 2017	x86_64	v141_xp
vs2017_64_v141	15 2017	x86_64	v141
vs2017_64_v141_xp	15 2017	x86_64	v141_xp
vs2017_64_v141_clang	15 2017	x86_64	v141_clang_c2
vs2017_64_v140	15 2017	x86_64	v140
vs2017_64_v140_xp	15 2017	x86_64	v140_xp
vs2017_64_v140_clang	15 2017	x86_64	v140_clang_c2
vs2017_64_v120	15 2017	x86_64	v120
vs2017_64_v120_xp	15 2017	x86_64	v120_xp
vs2017_64_v110	15 2017	x86_64	v110
vs2017_64_v110_xp	15 2017	x86_64	v110_xp
vs2017_64_v100	15 2017	x86_64	v100
vs2017_64_v100_xp	15 2017	x86_64	v100_xp
vs2017_64_v90	15 2017	x86_64	v90
vs2017_64_v90_xp	15 2017	x86_64	v90_xp
vs2017_64_v80	15 2017	x86_64	v80
vs2017_arm	15 2017	arm	v141
vs2017_arm_clang	15 2017	arm	v141_clang_c2
vs2017_arm_v141	15 2017	arm	v141
vs2017_arm_v141_clang	15 2017	arm	v141_clang_c2
vs2017_arm_v140	15 2017	arm	v140
vs2017_arm_v140_clang	15 2017	arm	v140_clang_c2
vs2017_arm_v120	15 2017	arm	v120
vs2017_arm_v110	15 2017	arm	v110
vs2017_arm_v100	15 2017	arm	v100

VS2015

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2015	14 2015	(native)	v140

Continued on next page

Table 4 – continued from previous page

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2015_clang	14 2015	(native)	v140_clang_c2
vs2015_xp	14 2015	(native)	v140_xp
vs2015_v140	14 2015	(native)	v140
vs2015_v140_xp	14 2015	(native)	v140_xp
vs2015_v140_clang	14 2015	(native)	v120
vs2015_v120	14 2015	(native)	v120_clang_c2
vs2015_v120_xp	14 2015	(native)	v120_xp
vs2015_v110	14 2015	(native)	v110
vs2015_v110_xp	14 2015	(native)	v110_xp
vs2015_v100	14 2015	(native)	v100
vs2015_v100_xp	14 2015	(native)	v100_xp
vs2015_v90	14 2015	(native)	v90
vs2015_v90_xp	14 2015	(native)	v90_xp
vs2015_v80	14 2015	(native)	v80
vs2015_32	14 2015	x86	v140
vs2015_32_clang	14 2015	x86	v140_clang_c2
vs2015_32_xp	14 2015	x86	v140_xp
vs2015_32_v140	14 2015	x86	v140
vs2015_32_v140_xp	14 2015	x86	v140_xp
vs2015_32_v140_clang	14 2015	x86	v140_clang_c2
vs2015_32_v120	14 2015	x86	v120
vs2015_32_v120_xp	14 2015	x86	v120_xp
vs2015_32_v110	14 2015	x86	v110
vs2015_32_v110_xp	14 2015	x86	v110_xp
vs2015_32_v100	14 2015	x86	v100
vs2015_32_v100_xp	14 2015	x86	v100_xp
vs2017_32_v90	14 2015	x86	v90
vs2017_32_v90_xp	14 2015	x86	v90_xp
vs2017_32_v80	14 2015	x86	v80
vs2015_64	14 2015	x86_64	v140
vs2015_64_clang	14 2015	x86_64	v140_clang_c2
vs2015_64_xp	14 2015	x86_64	v140_xp
vs2015_64_v140	14 2015	x86_64	v140
vs2015_64_v140_xp	14 2015	x86_64	v140_xp
vs2015_64_v140_clang	14 2015	x86_64	v140_clang_c2
vs2015_64_v120	14 2015	x86_64	v120
vs2015_64_v120_xp	14 2015	x86_64	v120_xp
vs2015_64_v110	14 2015	x86_64	v110
vs2015_64_v110_xp	14 2015	x86_64	v110_xp
vs2015_64_v100	14 2015	x86_64	v100
vs2015_64_v100_xp	14 2015	x86_64	v100_xp
vs2015_64_v90	14 2015	x86_64	v90
vs2015_64_v90_xp	14 2015	x86_64	v90_xp
vs2015_64_v80	14 2015	x86_64	v80
vs2015_arm	14 2015	arm	v140
vs2015_arm_clang	14 2015	arm	v140_clang_c2

VS2013

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2013	12 2013	(native)	v120
vs2013_xp	12 2013	(native)	v120_xp
vs2013_32	12 2013	x86	v120
vs2013_32_xp	12 2013	x86	v120_xp
vs2013_64	12 2013	x86_64	v120
vs2013_64_xp	12 2013	x86_64	v120_xp
vs2013_v110	12 2013	(native)	v110
vs2013_v110_xp	12 2013	(native)	v110_xp
vs2013_32_v110	12 2013	x86	v110
vs2013_32_v110_xp	12 2013	x86	v110_xp
vs2013_64_v110	12 2013	x86_64	v110
vs2013_64_v110_xp	12 2013	x86_64	v110_xp
vs2013_v100	12 2013	(native)	v100
vs2013_v100_xp	12 2013	(native)	v100_xp
vs2013_32_v100	12 2013	x86	v100
vs2013_32_v100_xp	12 2013	x86	v100_xp
vs2013_64_v100	12 2013	x86_64	v100
vs2013_64_v100_xp	12 2013	x86_64	v100_xp
vs2013_v90	12 2013	(native)	v90
vs2013_v90_xp	12 2013	(native)	v90_xp
vs2013_32_v90	12 2013	x86	v90
vs2013_32_v90_xp	12 2013	x86	v90_xp
vs2013_64_v90	12 2013	x86_64	v90
vs2013_64_v90_xp	12 2013	x86_64	v90_xp
vs2013_v80	12 2013	(native)	v80
vs2013_32_v80	12 2013	x86	v80
vs2013_64_v80	12 2013	x86_64	v80

VS2012

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2012	11 2012	(native)	v110
vs2012_xp	11 2012	(native)	v110_xp
vs2012_32	11 2012	x86	v110
vs2012_32_xp	11 2012	x86	v110_xp
vs2012_64	11 2012	x86_64	v110
vs2012_64_xp	11 2012	x86_64	v110_xp
vs2012_arm	11 2012	arm	v110
vs2012_arm_xp	11 2012	arm	v110_xp
vs2012_v110	11 2012	(native)	v110
vs2012_v110_xp	11 2012	(native)	v110_xp
vs2012_32_v110	11 2012	x86	v110
vs2012_32_v110_xp	11 2012	x86	v110_xp
vs2012_64_v110	11 2012	x86_64	v110
vs2012_64_v110_xp	11 2012	x86_64	v110_xp
vs2012_arm_v110	11 2012	arm	v110

Continued on next page

Table 5 – continued from previous page

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2012_arm_v110_xp	11 2012	arm	v110_xp
vs2012_v100	11 2012	(native)	v100
vs2012_v100_xp	11 2012	(native)	v100_xp
vs2012_32_v100	11 2012	x86	v100
vs2012_32_v100_xp	11 2012	x86	v100_xp
vs2012_64_v100	11 2012	x86_64	v100
vs2012_64_v100_xp	11 2012	x86_64	v100_xp
vs2012_arm_v100	11 2012	arm	v100
vs2012_arm_v100_xp	11 2012	arm	v100_xp
vs2012_v90	11 2012	(native)	v90
vs2012_v90_xp	11 2012	(native)	v90_xp
vs2012_32_v90	11 2012	x86	v90
vs2012_32_v90_xp	11 2012	x86	v90_xp
vs2012_64_v90	11 2012	x86_64	v90
vs2012_64_v90_xp	11 2012	x86_64	v90_xp
vs2012_arm_v90	11 2012	arm	v90
vs2012_arm_v90_xp	11 2012	arm	v90_xp
vs2012_v80	11 2012	(native)	v80
vs2012_32_v80	11 2012	x86	v80
vs2012_64_v80	11 2012	x86_64	v80
vs2012_arm_v80	11 2012	arm	v80

VS2010

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2010	10 2010	(native)	v100
vs2010_xp	10 2010	(native)	v100_xp
vs2010_32	10 2010	x86	v100
vs2010_32_xp	10 2010	x86	v100_xp
vs2010_64	10 2010	x86_64	v100
vs2010_64_xp	10 2010	x86_64	v100_xp
vs2010_ia64	10 2010	ia64	v100
vs2010_ia64_xp	10 2010	ia64	v100_xp
vs2010_v100	10 2010	(native)	v100
vs2010_v100_xp	10 2010	(native)	v100_xp
vs2010_32_v100	10 2010	x86	v100
vs2010_32_v100_xp	10 2010	x86	v100_xp
vs2010_64_v100	10 2010	x86_64	v100
vs2010_64_v100_xp	10 2010	x86_64	v100_xp
vs2010_ia64_v100	10 2010	ia64	v100
vs2010_ia64_v100_xp	10 2010	ia64	v100_xp
vs2010_v90	10 2010	(native)	v90
vs2010_v90_xp	10 2010	(native)	v90_xp
vs2010_32_v90	10 2010	x86	v90
vs2010_32_v90_xp	10 2010	x86	v90_xp
vs2010_64_v90	10 2010	x86_64	v90
vs2010_64_v90_xp	10 2010	x86_64	v90_xp
vs2010_ia64_v90	10 2010	ia64	v90
vs2010_ia64_v90_xp	10 2010	ia64	v90_xp
vs2010_v80	10 2010	(native)	v80
vs2010_32_v80	10 2010	x86	v80
vs2010_64_v80	10 2010	x86_64	v80

VS2008

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2008	9 2008	(native)	v90
vs2008_xp	9 2008	(native)	v90_xp
vs2008_32	9 2008	x86	v90
vs2008_32_xp	9 2008	x86	v90_xp
vs2008_64	9 2008	x86_64	v90
vs2008_64_xp	9 2008	x86_64	v90_xp
vs2008_ia64	9 2008	ia64	v90
vs2008_ia64_xp	9 2008	ia64	v90_xp
vs2008_v90	9 2008	(native)	v90
vs2008_v90_xp	9 2008	(native)	v90_xp
vs2008_32_v90	9 2008	x86	v90
vs2008_32_v90_xp	9 2008	x86	v90_xp
vs2008_64_v90	9 2008	x86_64	v90
vs2008_64_v90_xp	9 2008	x86_64	v90_xp
vs2008_ia64_v90	9 2008	ia64	v90
vs2008_ia64_v90_xp	9 2008	ia64	v90_xp
vs2008_v80	9 2008	(native)	v80
vs2008_32_v80	9 2008	x86	v80
vs2008_64_v80	9 2008	x86_64	v80
vs2008_ia64_v80	9 2008	ia64	v80

VS2005

cmany compiler alias	project VS version	Target arch.	VS Toolset
vs2005	8 2005	(native)	v80
vs2005_32	8 2005	x86	v80
vs2005_64	8 2005	x86_64	v80
vs2005_v80	8 2005	(native)	v80
vs2005_32_v80	8 2005	x86	v80
vs2005_64_v80	8 2005	x86_64	v80

2.8 Project dependencies

Certain projects need to integrate their dependencies fully into their build system, maybe because they are cross-platform or maybe because certain compiler flags and macros need to be used not just in the project's own source code, but also in the source code of the project's dependencies. Or maybe because it's just more practical to use CMake's external project facilities (which can serve as a sort-of dependency manager, as illustrated by projects such as Hunter) to get the dependencies' source code and compile them.

cmany offers the argument `--deps path/to/extern/CMakeLists.txt` to enable building another CMake project which builds and installs the dependencies of the current project. When `--deps` is given, the external project is built for each configuration, and installed in the configuration's build directory. Use `--deps-prefix` to specify a different install directory for the external project.

(To be continued)

2.9 Reusing arguments

In certain scenarios, the arguments to a `cmany` command can get a bit complicated. To aid in such scenarios, `cmany` offers two ways of storing these arguments, so that they are implicitly used in simple commands such as `cmany build`.

2.9.1 Session arguments

You can store arguments in the environment variable `CMANY_ARGS`. Then the resulting cmany command is taken as if it were given `cmany <subcommand> $CMANY_ARGS <command line arguments>`. In the example below, the configure, build and install commands will all use the five given compilers and two build types, resulting in 10 build trees:

```
$ export CMANY_ARGS="-c clang++-3.9,clang++-3.8,g++-6,g++-7,icpc \
                     -t Debug,Release"
$ cmany c   # 10 builds
$ cmany b   # same
$ cmany i   # same
```

The arguments stored in `CMANY_ARGS` can be combined with any other argument. For example, if you now want to build only the `Debug` types of the current value of `CMANY_ARGS`, just use the `--include-types/--it` *inclusion flag*:

```
$ cmany b -it Debug
```

or as another example, you can process only a single build tree via the `--include-builds/-ib`, say, `icpc` with `Release`:

```
$ cmany b -ib 'icpc.*Release'
```

Some arguments to cmany are meant to be used before the cmany subcommand. For those arguments, you should use the `CMANY_PFX_ARGS` environment variable instead of `CMANY_ARGS`. cmany will see commands given to it as `cmany $CMANY_PFX_ARGS <subcommand> $CMANY_ARGS <command line arguments>`.

Note: The values of the `CMANY_ARGS` and `CMANY_PFX_ARGS` environment variables are always used in every `cmany` invokation. To prevent cmany from using these values, you will have to unset the variables:

```
$ unset CMANY_ARGS CMANY_PFX_ARGS
$ cmany b   # uses defaults now
```

2.9.2 Project file

`cmany` also allows you to permanently store its arguments in a `cmany.yml` file which should be placed alongside the project `CMakeLists.txt`. This feature is under current development and is not ready.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search