# Table of Contents

# An Introduction to Modern CMake

People love to hate build systems. Just watch the talks from CppCon17 to see examples of developers making the state of build systems the brunt of jokes. This raises the question: Why? Certainly there are no shortage of problems when building. But I think that, in 2020, we have a very good solution to quite a few of those problems. It's CMake. Not CMake 2.8 though; that was released before C++11 even existed! Nor the horrible examples out there for CMake (even those posted on KitWare's own tutorials list). I'm talking about Modern CMake. CMake 3.1+, maybe even CMake 3.16+! It's clean, powerful, and elegant, so you can spend most of your time coding, not adding lines to an unreadable, unmaintainable Make (Or CMake 2) file. And CMake 3.11+ is supposed to be significantly faster, as well!

> This book is meant to be a living document. You can raise an issue or put in a merge request on GitLab. You can also download a copy as a PDF.

In short, here are the most likely questions in your mind if you are considering Modern CMake:

## Why do I need a good build system?

Do any of the following apply to you?

- You want to avoid hard-coding paths
- You need to build a package on more than one computer
- You want to use CI (continuous integration)
- You need to support different OSs (maybe even just flavors of Unix)
- You want to support multiple compilers
- You want to use an IDE, but maybe not all of the time
- You want to describe how your program is structured logically, not flags and commands
- You want to use a library
- You want to use tools, like Clang-Tidy, to help you code
- You want to use a debugger

If so, you'll benefit from a CMake-like build system.

## Why must the answer be CMake?

Build systems is a hot topic. Of course there are many options. But even a really good one, or one that re-uses a familiar syntax, can't come close to CMake. Why? Support. Every IDE supports CMake (or CMake supports that IDE). More packages use CMake than any other system. So, if you use a library that is designed to be included in your code, you have a choice: Make your own build system, or use one of of the provided ones, and that will almost always include CMake. And that will quickly be the common denominator if you include multiple projects. And, if you need a library that's preinstalled, the chances of it having a find CMake script or config CMake script are excellent.

## Why use a Modern CMake?

Around CMake 2.6-2.8, CMake started taking over. It was in most of the package managers for Linux OS's, and was being used in lots of packages.

Then Python 3 came out.

I know, this should have nothing whatsoever to do with CMake.

But it had a 3. And it followed 2. And it was a hard, ugly, transition that is still ongoing in some places, even today.

I believe that CMake 3 had the bad luck to follow Python 3.[1] Even though every version of CMake is insanely backward compatible, the 3 series was treated as if it was something new. And so, you'll find OS's like CentOS7 with GCC 4.8, with almost-complete C++14 support, and CMake 2.8, which came out before C++11.

You really should *at least* use a version of CMake that came out after your compiler, since it needs to know compiler flags, etc, for that version. And, since CMake will dumb itself down to the minimum required version in your CMake file, installing a new CMake, even system wide, is pretty safe. You should *at least* install it locally. It's easy (1-2 lines in many cases), and you'll find that 5 minutes of work will save you hundreds of lines and hours of `CMakeLists.txt` writing, and will be much easier to maintain in the long run.

This book tries to solve the problem of the poor examples and best practices that you'll find proliferating the web.

# Other sources

Other material from the original author of this book:

- CMake Workshop
- Interactive Modern CMake talk

There are some other places to find good information on the web. Here are some of them:

- The official help: Really amazing documentation. Nicely organized, great search, and you can toggle versions at the top. It just doesn't have a great "best practices tutorial", which is what this book tries to fill in.
- Effective Modern CMake: A great list of do's and don'ts.
- Embracing Modern CMake: A post with good description of the term
- It's time to do CMake Right: A nice set of best practices for Modern CMake projects.
- The Ultimate Guide to Modern CMake: A slightly dated post with similar intent.
- More Modern CMake: A great presentation from Meeting C++ 2018 that recommends CMake 3.12+. This talk makes calls CMake 3.0+ "Modern CMake" and CMake 3.12+ "More Modern CMake".
- Oh No! More Modern CMake: The sequel to More Modern CMake.
- toeb/moderncmake: A nice presentation and examples about CMake 3.5+, with intro to syntax through project organization

# Credits

Modern CMake was originally written by Henry Schreiner. Other contributors can be found listed on GitLab.

> [1]. CMake 3.0 also removed several long deprecated features from very old versions of CMake and make one very tiny backwards incompatible change to syntax related to square brackets, so this is not entirely fair; there might be some very, very old CMake files that would stop working with 3. I've never seen one, though. ↩

# Installing CMake

> Your CMake version should be newer than your compiler. It should be newer than the libraries you are using (especially Boost). New versions work better for everyone.

If you have a built in copy of CMake, it isn't special or customized for your system. You can easily install a new one instead, either on the system level or the user level. Feel free to instruct your users here if they complain about a CMake requirement being set too high. Especially if they want < 3.1 support. Maybe even if they want CMake < 3.16 support...

## Quick list (more info on each method below)

Ordered by author preference:

- All
  - Pip (official, sometimes delayed slightly)
  - Anaconda / Conda-Forge
- Windows
  - Chocolaty
  - Download binary (official)
- MacOS
  - Homebrew
  - MacPorts
  - Download binary (official)
- Linux
  - Snapcraft (official)
  - APT repository (Ubuntu/Debian only) (official)
  - Download binary (official)

# Official package

You can download CMake from KitWare. This is how you will probably get CMake if you are on Windows. It's not a bad way to get it on macOS either, but using `brew install cmake` is much nicer if you use Homebrew (and you should). You can also get it on most other package managers, such as Chocolaty for Windows or MacPorts for macOS.

On Linux, there are several options. Kitware provides a Debian/Ubunutu apt repository, as well as snap packages. There are universal Linux binaries provided, but you'll need to pick an install location. If you already use `~/.local` for user-space packages, the following single line command[1] will get CMake for you [2]:

```
~ $ wget -qO- "https://cmake.org/files/v3.16/cmake-3.16.3-Linux-x86_64.tar.gz" | tar --
strip-components=1 -xz -C ~/.local
```

If you just want a local folder with CMake only:

```
~ $ mkdir -p cmake-3.16 && wget -qO- "https://cmake.org/files/v3.16/cmake-3.16.3-Linux-
x86_64.tar.gz" | tar --strip-components=1 -xz -C cmake-3.16
~ $ export PATH=`pwd`/cmake-3.16/bin:$PATH
```

You'll obviously want to append to the PATH every time you start a new terminal, or add it to your `.bashrc` or to an LMod system.

And, if you want a system install, install to `/usr/local`; this is an excellent choice in a Docker container, for example on GitLab CI. Do not try it on a non-containerized system.

```
docker $ wget -qO- "https://cmake.org/files/v3.16/cmake-3.16.3-Linux-x86_64.tar.gz" | tar
--strip-components=1 -xz -C /usr/local
```

If you are on a system without wget, replace `wget -qO-` with `curl -s`.

You can also build CMake on any system, it's pretty easy, but binaries are faster.

# CMake Default Versions

Here are some common build environments and the CMake version you'll find on them. Feel free to install CMake yourself, it's 1-2 lines and there's nothing "special" about the built in version. It's also very backward compatible.

| Distribution | CMake version | Notes |
|---|---|---|
| RHEL/CentOS 7 | 2.8.11 | Don't use the default on this system. Grab a new copy or use the EPEL repo. |
| RHEL/CentOS 8 | 3.11.4 | Not too bad. |
| EPEL for RHEL/CentOS | 3.14.2 | Called `cmake3` |
| Ubuntu 14.04 LTS: Trusty | 2.8.12 | Don't use the default on this system. |
| Ubuntu 16.04 LTS: Xenial | 3.5.1 | |
| Ubuntu 18.04 LTS: Bionic | 3.10.2 | An LTS with a pretty decent minimum version! |
| Ubuntu 18.10: Cosmic | 3.12.1 | |
| Ubuntu 19.04: Disco | 3.13.4 | |
| Ubuntu 19.10: Eoan | 3.13.4 | Oddly identical to Disco. |
| AlpineLinux 3.11 | 3.15.5 | Useful in Docker |
| Python PyPI | 3.15.3 | Just `pip install cmake` on many systems. Add `--user` for local installs. (ManyLinux1 (old pip or OS) gets CMake 3.13.3) |
| Anaconda | 3.14.0 | For use with Conda |
| Conda-Forge | 3.16.3 | For use with Conda |
| Homebrew on macOS | 3.16.3 | On macOS with Homebrew, this is only a few minutes behind cmake.org. |
| MacPorts on macOS | 3.16.3 | Useful if you use the less popular MacPorts. |
| Chocolaty on Windows | 3.16.2 | Also up to date. The normal cmake.org installers are common on Windows, as well. |
| TravisCI Xenial | 3.12.4 | Mid November 2018 this image became ready for widescale use. |
| TravisCI Bionic | 3.12.4 | Same as Xenial at the moment. |
| Azure DevOps 18.04 | 3.12.4 | |

| | | |
|---|---|---|
| GitHub Actions 18.04 | 3.12.4 | Mostly in sync with Azure DevOps |

Also see pkgs.org/download/cmake.

# Pip

This is also provided as an official package, maintained by the authors of CMake at KitWare. It's a rather new method, and might fail on some systems (Alpine isn't supported last I checked, but that has CMake 3.8), but works really well when it works (like on Travis CI). If you have pip (Python's package installer), you can do:

```
gitbook $ pip install cmake
```

And as long as a binary exists for your system, you'll be up-and-running almost immediately. If a binary doesn't exist, it will try to use KitWare's `scikit-build` package to build, which currently can't be listed as a dependency in the packaging system, and might even require (an older) copy of CMake to build. So only use this system if binaries exist, which is most of the time.

This has the benefit of respecting your current virtual environment, as well.

> Personally, on Linux, I put versions of CMake in folders, like `/opt/cmake312` or `~/opt/cmake312`, and then add them to [LMod]. See `envmodule_setup` for help setting up an LMod system on macOS or Linux. It takes a bit to learn, but is a great way to manage package and compiler versions.

> [1]. I assume this is obvious, but you are downloading and running code, which exposes you to a man in the middle attack. If you are in a critical environment, you should download the file and check the checksum. (And, no, simply doing this in two steps does not make you any safer, only a checksum is safer). ↩

> [2]. If you don't have a `.local` in your home directory, it's easy to start. Just make the folder, then add `export PATH="$HOME/.local/bin:$PATH"` to your `.bashrc` or `.bash_profile` or `.profile` file in your home directory. Now you can install any packages you build to `-DCMAKE_INSTALL_PREFIX=~/.local` instead of `/usr/local`! ↩

# Running CMake

Before writing CMake, let's make sure you know how to run it to make things. This is true for almost all CMake projects, which is almost everything.

## Building a project

Unless otherwise noted, you should always make a build directory and build from there. You can technically do an in-source build, but you'll have to be careful not to overwrite files or add them to git, so just don't.

Here's the Classic CMake Build Procedure (TM):

```
~/package $ mkdir build
~/package $ cd build
~/package/build $ cmake ..
~/package/build $ make
```

You can replace the make line with `cmake --build .` if you'd like, and it will call `make` or whatever build tool you are using. If you are using a newer version of CMake (which you usually should be, except for checking compatibility with older CMake), you can instead do this:

```
~/package $ cmake -S . -B build
~/package $ cmake --build build
```

Any *one* of these commands will install:

```
# From the build directory (pick one)
~/package/build $ make install
~/package/build $ cmake --build . --target install
~/package/build $ cmake --install . # CMake 3.15+ only

# From the source directory (pick one)
~/package $ cmake --build build --target install
~/package $ cmake --install build # CMake 3.15+ only
```

So set of methods should you use? As long as you *do not forget* to type the build directory as the argument, staying out of the build directory is shorter, and making source changes is easier from the source directory. You should try to get used to using `--build`, as that will free you from using only `make` to build. Note that working from the build directory is historically much more common, and some tools and commands (including CTest) still require running from the build directory.

Just to clarify, you can point CMake at either the source directory *from the build directory*, or at an *existing* build directory from anywhere.

If you use `cmake --build` instead of directly calling the underlying build system, you can use `-v` for verbose builds (CMake 3.14+), `-j N` for parallel builds on N cores (CMake 3.12+), and `--target` (any version of CMake) or `-t` (CMake 3.15+) to pick a target. Otherwise, these commands vary between build systems, such as `VERBOSE=1 make` and `ninja -v`.

## Picking a compiler

Selecting a compiler must be done on the first run in an empty directory. It's not CMake syntax per se, but you might not be familiar with it. To pick Clang:

```
~/package/build $ CC=clang CXX=clang++ cmake ..
```

That sets the environment variables in bash for CC and CXX, and CMake will respect those variables. This sets it just for that one line, but that's the only time you'll need those; afterwards CMake continues to use the paths it deduces from those values.

# Picking a generator

You can build with a variety of tools; `make` is usually the default. To see all the tools CMake knows about on your system, run

```
~/package/build $ cmake --help
```

And you can pick a tool with `-G"My Tool"` (quotes only needed if spaces are in the tool name). You should pick a tool on your first CMake call in a directory, just like the compiler. Feel free to have several build directories, like `build/` and `buildXcode`. You can set the environment variable `CMAKE_GENERATOR` to control the default generator (CMake 3.15+). Note that makefiles will only run in parallel if you explicilty pass a number of threads, such as `make -j2`, while Ninja will automatically run in parallel. You can directly pass a parallelization option such as `-j2` to the `cmake --build .` command in recent versions of CMake.

# Setting options

You set options in CMake with `-D`. You can see a list of options with `-L`, or a list with human-readable help with `-LH`. If you don't list the source/build directory, the listing will not rerun CMake (`cmake -L` instead of `cmake -L .`).

# Verbose and partial builds

Again, not really CMake, but if you are using a command line build tool like `make`, you can get verbose builds:

```
~/package/build $ VERBOSE=1 make
```

You can actually write `make VERBOSE=1`, and make will also do the right thing, though that's a feature of `make` and not the command line in general.

You can also build just a part of a build by specifying a target, such as the name of a library or executable you've defined in CMake, and make will just build that target.

# Options

CMake has support for cached options. A Variable in CMake can be marked as "cached", which means it will be written to the cache (a file called `CMakeCache.txt` in the build directory) when it is encountered. You can preset (or change) the value of a cached option on the command line with `-D`. When CMake looks for a cached variable, it will use the existing value and will not overwrite it.

## Standard options

These are common CMake options to most packages:

- `-DCMAKE_BUILD_TYPE=` Pick from Release, RelWithDebInfo, Debug, or sometimes more.
- `-DCMAKE_INSTALL_PREFIX=` The location to install to. System install on UNIX would often be `/usr/local` (the default), user directories are often `~/.local`, or you can pick a folder.

- `-DBUILD_SHARED_LIBS=` You can set this `ON` or `OFF` to control the default for shared libraries (the author can pick one vs. the other explicitly instead of using the default, though)
- `-DBUILD_TESTING=` This is a common name for enabling tests, not all packages use it, though, sometimes with good reason.

# Debugging your CMake files

We've already mentioned verbose output for the build, but you can also see verbose CMake configure output too. The `--trace` option will print every line of CMake that is run. Since this is very verbose, CMake 3.7 added `--trace-source="filename"`, which will print out every executed line of just the file you are interested in when it runs. If you select the name of the file you are interested in debugging (usually by selecting the parent directory when debugging a CMakeLists.txt, since all of those have the same name), you can just see the lines that run in that file. Very useful!

# Do's and Don'ts

## CMake Antipatterns

The next two lists are heavily based on the excellent gist Effective Modern CMake. That list is much longer and more detailed, feel free to read it as well.

- **Do not use global functions**: This includes `link_directories`, `include_libraries`, and similar.
- **Don't add unneeded PUBLIC requirements**: You should avoid forcing something on users that is not required (`-Wall`). Make these PRIVATE instead.
- **Don't GLOB files**: Make or another tool will not know if you add files without rerunning CMake. Note that CMake 3.12 adds a `CONFIGURE_DEPENDS` flag that makes this far better if you need to use it.
- **Link to built files directly**: Always link to targets if available.
- **Never skip PUBLIC/PRIVATE when linking**: This causes all future linking to be keyword-less.

## CMake Patterns

- **Treat CMake as code**: It is code. It should be as clean and readable as all other code.
- **Think in targets**: Your targets should represent concepts. Make an (IMPORTED) INTERFACE target for anything that should stay together and link to that.
- **Export your interface**: You should be able to run from build or install.
- **Write a Config.cmake file**: This is what a library author should do to support clients.
- **Make ALIAS targets to keep usage consistent**: Using `add_subdirectory` and `find_package` should provide the same targets and namespaces.
- **Combine common functionality into clearly documented functions or macros**: Functions are better usually.
- **Use lowercase function names**: CMake functions and macros can be called lower or upper case. Always user lower case. Upper case is for variables.
- **Use `cmake_policy` and/or range of versions**: Policies change for a reason. Only piecemeal set OLD policies if you have to.

# What's new in in CMake

This is an abbreviated version of the CMake changlog with just the highlights for authors. Names for each release are arbitrarily picked by the author.

## CMake 3.0 : Interface libraries

There were a ton of additions to this version of CMake, primarily to fill out the target interface. Some bits of needed functionality were missed and implemented in CMake 3.1 instead.

- New documentation
- INTERFACE libraries
- Project VERSION support
- Exporting build trees easily
- Bracket arguments and comments available (not widely used)
- Lots of improvements

## CMake 3.1 : C++11 and compile features

This is the first release of CMake to support C++11. Combined with fixes to the new features of CMake 3.0, this is currently a common minimum version of CMake for libraries that want to support old CMake builds.

- C++11 Support
- Compile features support
- Sources can be added later with `target_sources`
- Better support for generator expressions and INTERFACE targets

## CMake 3.2 : UTF8

This is a smaller release, with mostly small features and fixes. Internal changes, like better Windows and UTF8 support, were the focus.

- `continue()` inside loops
- File and directory locks added

## CMake 3.3 : if IN_LIST

This is notable for the useful `IN_LIST` option for if, but it also added better library search using `$PATH` (See CMake 3.6), dependencies for INTERFACE libraries, and several other useful improvements. The addition of a `COMPILE_LANGUAGE` generator expression would prove very useful in the future as more languages are added. Makefiles now produce better output in parallel.

- `IN_LIST` added to `if`
- `*_INCLUDE_WHAT_YOU_USE` property added
- `COMPILE_LANGUAGE` generator expression (limited support in some generators)

## CMake 3.4 : Swift & CCache

This release adds lots of useful tools, support for the Swift language, and the usual improvements. It also started supporting compiler launchers, like CCache.

- Added `Swift` language

- Added `BASE_DIR` to `get_filename_component`
- `if(TEST ...)` added
- `string(APPEND ...)` added
- `CMAKE_*_COMPILER_LAUNCHER` added for make and ninja
- `TARGET_MESSAGES` allow makefiles to print messages after target is completed
- Imported targets are beginning to show up in the official `Find*.cmake` files

## CMake 3.5 : ARM

This release expanded CMake to more platforms, and make warnings easier to control from the command line.

- Multiple input files supported for several of the `cmake -E` commands.
- `cmake_parse_arguments` now builtin
- Boost, GTest, and more now support imported targets
- ARMCC now supported, better support for iOS
- XCode backslash fix

## CMake 3.6 : Clang-Tidy

This release added Clang-Tidy support, along with more utilities and improvements. It also removed the search of `$PATH` on Unix systems due to problems, instead users should use `$CMAKE_PREFIX_PATH`.

- `EXCLUDE_FROM_ALL` for install
- `list(FILTER` added
- `CMAKE_*_STANDARD_INCLUDE_DIRECTORIES` and `CMAKE_*_STANDARD_LIBRARIES` added for toolchains
- Try-compile improvements
- `*_CLANG_TIDY` property added
- External projects can now be shallow clones, and other improvements

## CMake 3.7 : Android & CMake Server

You can now cross-compile to Android. Useful new if statement options really help clarify code. And the new server mode was supposed to improve integration with IDEs (but is being replaced by a different system in CMake 3.14+). Support for the VIM editor was also improved.

- `PARSE_ARGV` mode for `cmake_parse_arguments`
- Better 32-bit support on 64-bit machines
- Lots of useful new if comparisons, like `VERSION_GREATER_EQUAL` (really, why did it take this long?)
- `LINK_WHAT_YOU_USE` added
- Lots of custom properties related to files and directories
- CMake Server added
- Added `--trace-source="filename"` to monitor certain files only

## CMake 3.8 : C# & CUDA

This adds CUDA as a language, as well as `cxx_std_11` as a compiler meta-feature. The new generator expression could be really useful if you can require CMake 3.8+!

- Native support for C# as a language
- Native support for CUDA as a language
- Meta features cxx_std_11 (and 14, 17) added
- `try_compile` has better language support

- `BUILD_RPATH` property added
- `COMPILE_FLAGS` now supports generator expression
- `*_CPPLINT` added
- `$<IF:cond,true-value,false-value>` added (wow!)
- `source_group(TREE` added (finally allowing IDE's to reflect the project folder structure!)

## CMake 3.9 : IPO

Lots of fixes to CUDA support went into this release, including `PTX` support and MSVC generators. Interprocedural Optimizations are now supported properly. Even more modules provide imported targets, including MPI.

- CUDA supported for Windows
- Better object library support in several situations
- `DESCRIPTION` added to `project`
- `separate_arguments` gets `NATIVE_COMMAND`
- `INTERPROCEDURAL_OPTIMIZATION` enforced (and `CMAKE_*` initializer added, CheckIPOSupported added, Clang and GCC support)
- New `GoogleTest` module
- `FindDoxygen` drastically improved

## CMake 3.10 : CppCheck

CMake now is built with C++11 compilers. Lots of useful improvements help write cleaner code.

- Support for flang Fortran compiler
- Compiler launcher added to CUDA
- Indented `#cmakedefines` now supported for `configure_file`
- `include_guard()` added to ensure a file gets included only once
- `string(PREPEND` added
- `*_CPPCHECK` property added
- `LABELS` added to directories
- FindMPI vastly expanded
- FindOpenMP improved
- Dynamic test discovery for `GoogleTest`

## CMake 3.11 : Faster & IMPORTED INTERFACE

This release is supposed to be much faster. You can also finally directly add INTERFACE targets to IMPORTED libraries (the internal `Find*.cmake` scripts should become much cleaner eventually).

- Fortran supports compiler launchers
- Xcode and Visual Studio support `COMPILE_LANGUAGE` generator expressions finally
- You can now add INTERFACE targets directly to IMPORTED INTERFACE libraries (Wow!)
- Source file properties have been expanded
- `FetchContent` module now allows downloads to happen at configure time (Wow)

## CMake 3.12 : Version ranges and CONFIGURE_DEPENDS

Very powerful release, containing lots of smaller long-requested features. One of the smaller but immediately noticeable changes is the addition of version ranges; you can now set both the minimum and maximum known CMake version easily. You can also set `CONFIGURE_DEPENDS` on a `GLOB`ed set of files, and the build system will check those files and rerun if needed! You can use the general `PackageName_ROOT` for all `find_package` searches. Lots of additions to strings and lists, module updates, shiny new Python find module (2 and 3 versions too), and many more.

- Support for `cmake_minimum_required` ranges (backward compatible)
- Support for `-j,--parallel` in `--build` mode (passed on to build tool)
- Support for `SHELL:` strings in compile options (not deduplicated)
- New FindPython module
- `string(JOIN` and `list(JOIN`, and `list(TRANSFORM`
- `file(TOUCH` and `file(GLOB CONFIGURE_DEPENDS`
- C++20 support
- CUDA as a language improvements: CUDA 7 and 7.5 now supported
- Support for OpenMP on macOS (command line only)
- Several new properties and property initializers
- CPack finally reads `CMAKE_PROJECT_VERSION` variables

## CMake 3.13 : Linking control

You can now make symbolic links on Windows! Lots of new functions that fill out the popular requests for CMake, such as `add_link_options`, `target_link_directories`, and `target_link_options`. You can now do quite a bit more modification to targets outside of the source directory, for better file separation. And, `target_sources` *finally* handles relative paths properly (policy 76).

- New `ctest --progress` option for live output
- `target_link_options` and `add_link_options` added
- `target_link_directories` added
- Symbolic link creation, `-E create_symlink`, supported on Windows
- IPO supported on Windows
- You can use `-S` and `-B` for source and build directories
- `target_link_libraries` and `install` work outside the current target directory
- `STATIC_LIBRARY_OPTIONS` property added
- `target_sources` is now relative to the current source directory (CMP0076)
- If you use Xcode, you now can experimentally set schema fields

## CMake 3.14 : File utilities (AKA CMake π)

This release has lots of small cleanups, including several utilities for files. Generator expressions work in a few more places, and list handling is better with empty variables. Quite a few more find packages produce targets. The new Visual Studio 16 2019 generator is a bit different than older versions. Windows XP and Vista support has been dropped.

- The cmake `--build` command gained `-v/--verbose`, to use verbose builds if your build tool supports it
- The FILE command gained `CREATE_LINK`, `READ_SYMLINK`, and `SIZE`
- `get_filename_component` gained `LAST_EXT` and `NAME_WLE` to access just the *last* extension on a file, which would get `.zip` on a file such as `version.1.2.zip` (very handy!)
- You can see if a variable is defined in the CACHE with `DEFINED CACHE{VAR}` in an `if` statement.
- `BUILD_RPATH_USE_ORIGIN` and CMake version were added to improve handling of RPath in the build directory.
- The CMake server mode is now being replaced with a file API, starting in this release. Will affect IDEs in the long run.

## CMake 3.15 : CLI upgrade

This release has many smaller polishing changes, include several of improvements to the CMake command line, such as control over the default generator through environment variables (so now it's easy to change the default generator to Ninja). Multiple targets are supported in `--build` mode, and `--install` mode added. CMake finally supports multiple levels of logging. Generator expressions gained a few handy tools. The still very new FindPython module continues to improve, and FindBoost is now more inline with Boost 1.70's new CONFIG module. `export(PACKAGE)` has drastically changed; it now no longer touches `$HOME/.cmake` by default (if CMake Minimum version is 3.15 or higher), and requires an extra step if a user wants to use it. This is generally less surprising.

- `CMAKE_GENERATOR` environment variable added to control default generator
- Multiple target support in build mode, `cmake . --build --target a b`
- Shortcut `-t` for `--target`
- Install support, `cmake . --install`, does not invoke the build system
- Support for `--loglevel` and `NOTICE`, `VERBOSE`, `DEBUG`, and `TRACE` for `message`
- The `list` command gained `PREPEND`, `POP_FRONT`, and `POP_BACK`
- `execute_process` gained `COMMAND_ECHO` option (`CMAKE_EXECUTE_PROCESS_COMMAND_ECHO`) allows you to automatically echo commands before running them
- Several Ninja improvements, include SWIFT language support
- Compiler and list improvements to generator expressions

## CMake 3.16 : Unity builds

A new unity build mode was added, allowing source files to be merged into a single build file. Support for precompiled headers (possibly preparing for C++20 modules, perhaps?) was added. Lots of other smaller fixes were implemented, especially to newer features, such as to FindPython, FindDoxygen, and others.

- Added support for Objective C and Objective C++ languages
- Support for precompiling headers, with `target_precompile_headers`
- Support for "Unity" or "Jumbo" builds (merging source files) with `CMAKE_UNITY_BUILD`
- CTest: Can now skip based on regex, expand lists
- Several new features to control RPath.
- Generator expressions work in more places, like build and install paths
- Find locations can now be explicitly controlled through new variables

## CMake 3.17 (in progress) : More CUDA

- `CUDA_RUNTIME_LIBRARY` can finally be set to Shared!
- FindCUDAToolkit finally added
- CUDA has meta features like `cuda_std_03`, etc.
- ExternalProject can now disable recursive checkouts
- FindPython better integration with Conda
- DEPRECATION can be applied to targets
- CMake gained a rm command
- Several new environment variables
- foreach can now do `ZIP_LISTS` (multiple lists at a time)

# Introduction to the basics

## Minimum Version

Here's the first line of every CMakeLists.txt, which is the required name of the file CMake looks for:

```
cmake_minimum_required(VERSION 3.1)
```

Let's mention a bit of CMake syntax. The command name `cmake_minimum_required` is case insensitive, so the common practice is to use lower case. [1] The `VERSION` is a special keyword for this function. And the value of the version follows the keyword. Like everywhere in this book, just click on the command name to see the official documentation, and use the dropdown to switch documentation between CMake versions.

This line is special! [2] The version of CMake will also dictate the policies, which define behavior changes. So, if you set `minimum_required` to `VERSION 2.8`, you'll get the wrong linking behavior on macOS, for example, even in the newest CMake versions. If you set it to 3.3 or less, you'll get the wrong hidden symbols behaviour, etc. A list of policies and versions is available at policies.

In CMake 3.12, this will support a range, such as `VERSION 3.1...3.12`; this means you support as low as 3.1 but have also tested it with the new policy settings up to 3.12. This is much nicer on users that need the better settings, and due to a trick in the syntax, it's backward compatible with older versions of CMake (though actually running CMake 3.2-3.11 will only set the 3.1 version of the policies in this example). New versions of policies tend to be most important for macOS and Windows users, who also usually have a very recent version of CMake.

This is what new projects should do:

```
cmake_minimum_required(VERSION 3.1...3.15)

if(${CMAKE_VERSION} VERSION_LESS 3.12)
    cmake_policy(VERSION ${CMAKE_MAJOR_VERSION}.${CMAKE_MINOR_VERSION})
endif()
```

If CMake version is less than 3.12, the if block will be true, and the policy will be set to the current CMake version. If CMake is 3.12 or higher, the if block will be false, but the new syntax in `cmake_minimum_required` will be respected and this will continue to work properly!

WARNING: MSVC's CMake server mode originally had a bug in reading this format, so if you need to support non-command line Windows builds for older MSVC versions, you will want to do this instead:

```
cmake_minimum_required(VERSION 3.1)

if(${CMAKE_VERSION} VERSION_LESS 3.15)
    cmake_policy(VERSION ${CMAKE_MAJOR_VERSION}.${CMAKE_MINOR_VERSION})
else()
    cmake_policy(VERSION 3.15)
endif()
```

> If you really need to set to a low value here, you can use `cmake_policy` to conditionally increase the policy level or set a specific policy. Please at least do this for your macOS users!

## Setting a project

Now, every top-level CMake file will have the next line:

```
project(MyProject VERSION 1.0
                  DESCRIPTION "Very nice project"
                  LANGUAGES CXX)
```

Now we see even more syntax. Strings are quoted, whitespace doesn't matter, and the name of the project is the first argument (positional). All the keyword arguments here are optional. The version sets a bunch of variables, like `MyProject_VERSION` and `PROJECT_VERSION`. The languages are `C`, `CXX`, `Fortran`, `ASM`, `CUDA` (CMake 3.8+), `CSharp` (3.8+), and `SWIFT` (CMake 3.15+ experimental). `C CXX` is the default. In CMake 3.9, `DESCRIPTION` was added to set a project description, as well. The documentation for `project` may be helpful.

> You can add comments with the `#` character. CMake does have an inline syntax for comments too, but it is rarely needed, as whitespace doesn't matter.

There's really nothing special about the project name. No targets are added at this point.

# Making an executable

Although libraries are much more interesting, and we'll spend most of our time with them, let's start with a simple executable.

```
add_executable(one two.cpp three.h)
```

There are several things to unpack here. `one` is both the name of the executable file generated, and the name of the CMake target created (you'll hear a lot more about targets soon, I promise). The source file list comes next, and you can list as many as you'd like. CMake is smart, and will only compile source file extensions. The headers will be, for most intents and purposes, ignored; the only reason to list them is to get them to show up in IDEs. Targets show up as folders in many IDEs. More about the general build system and targets is available at buildsystem.

# Making a library

Making a library is done with `add_library`, and is just about as simple:

```
add_library(one STATIC two.cpp three.h)
```

You get to pick a type of library, STATIC, SHARED, or MODULE. If you leave this choice off, the value of `BUILD_SHARED_LIBS` will be used to pick between STATIC and SHARED.

As you'll see in the following sections, often you'll need to make a fictional target, that is, one where nothing needs to be compiled, for example, for a header-only library. That is called an INTERFACE library, and is another choice; the only difference is it cannot be followed by filenames.

You can also make an `ALIAS` library with an existing library, which simply gives you a new name for a target. The one benefit to this is that you can make libraries with `::` in the name (which you'll see later). [3]

# Targets are your friend

Now we've specified a target, how do we add information about it? For example, maybe it needs an include directory:

```
target_include_directories(one PUBLIC include)
```

`target_include_directories` adds an include directory to a target. `PUBLIC` doesn't mean much for an executable; for a library it lets CMake know that any targets that link to this target must also need that include directory. Other options are `PRIVATE` (only affect the current target, not dependencies), and `INTERFACE` (only needed for dependencies).

We can then chain targets:

```
add_library(another STATIC another.cpp another.h)
target_link_libraries(another PUBLIC one)
```

`target_link_libraries` is probably the most useful and confusing command in CMake. It takes a target ( `another` ) and adds a dependency if a target is given. If no target of that name ( `one` ) exists, then it adds a link to a library called `one` on your path (hence the name of the command). Or you can give it a full path to a library. Or a linker flag. Just to add a final bit of confusion, classic CMake allowed you to skip the keyword selection of `PUBLIC` , etc. If this was done on a target, you'll get an error if you try to mix styles further down the chain.

Focus on using targets everywhere, and keywords everywhere, and you'll be fine.

Targets can have include directories, linked libraries (or linked targets), compile options, compile definitions, compile features (see the C++11 chapter), and more. As you'll see in the two including projects chapters, you can often get targets (and always make targets) to represent all the libraries you use. Even things that are not true libraries, like OpenMP, can be represented with targets. This is why Modern CMake is great!

# Dive in

See if you can follow the following file. It makes a simple C++11 library and a program using it. No dependencies. I'll discuss more C++ standard options later, using the CMake 3.8 system for now.

```
cmake_minimum_required(VERSION 3.8)

project(Calculator LANGUAGES CXX)

add_library(calclib STATIC src/calclib.cpp include/calc/lib.hpp)
target_include_directories(calclib PUBLIC include)
target_compile_features(calclib PUBLIC cxx_std_11)

add_executable(calc apps/calc.cpp)
target_link_libraries(calc PUBLIC calclib)
```

> [1]. In this book, I'll mostly avoid showing you the wrong way to do things; you can find plenty of examples of that online. I'll mention alternatives occasionally, but these are not recommended unless they are absolutely necessary; often they are just there to help you read older CMake code. ↩

> [2]. You will sometimes see `FATAL_ERROR` here, that was needed to support nice failures when running this in CMake <2.6, which should not be a problem anymore. ↩

> [3]. The `::` syntax was originally intended for `INTERFACE IMPORTED` libraries, which were explicitly supposed to be libraries defined outside the current project. But, because of this, most of the `target_*` commands don't work on `IMPORTED` libraries, making them hard to set up yourself. So don't use the `IMPORTED` keyword for now, and use an `ALIAS` target instead; it will be fine until you start exporting targets. This limitation was fixed in CMake 3.11. ↩

# Variables and the Cache

## Local Variables

We will cover variables first. A local variable is set like this:

```
set(MY_VARIABLE "value")
```

The names of variables are usually all caps, and the value follows. You access a variable by using `${}`, such as `${MY_VARIABLE}`.[1] CMake has the concept of scope; you can access the value of the variable after you set it as long as you are in the same scope. If you leave a function or a file in a sub directory, the variable will no longer be defined. You can set a variable in the scope immediately above your current one with `PARENT_SCOPE` at the end.

Lists are simply a series of values when you set them:

```
set(MY_LIST "one" "two")
```

which internally become `;` separated values. So this is an identical statement:

```
set(MY_LIST "one;two")
```

The `list(` command has utilities for working with lists, and `separate_arguments` will turn a space separated string into a list (inplace). Note that an unquoted value in CMake is the same as a quoted one if there are no spaces in it; this allows you to skip the quotes most of the time when working with value that you know could not contain spaces.

When a variable is expanded using `${}` syntax, all the same rules about spaces apply. Be especially careful with paths; paths may contain a space at any time and should always be quoted when they are a variable (never write `${MY_PATH}`, always should be `"${MY_PATH}"`).

## Cache Variables

If you want to set a variable from the command line, CMake offers a variable cache. Some variables are already here, like `CMAKE_BUILD_TYPE`. The syntax for declaring a variable and setting it if it is not already set is:

```
set(MY_CACHE_VARIABLE "VALUE" CACHE STRING "Description")
```

This will **not** replace an existing value. This is so that you can set these on the command line and not have them overridden when the CMake file executes. If you want to use these variables as a make-shift global variable, then you can do:

```
set(MY_CACHE_VARIABLE "VALUE" CACHE STRING "" FORCE)
mark_as_advanced(MY_CACHE_VARIABLE)
```

The first line will cause the value to be set no matter what, and the second line will keep the variable from showing up in the list of variables if you run `cmake -L ..` or use a GUI. This is so common, you can also use the `INTERNAL` type to do the same thing (though technically it forces the STRING type, this won't affect any CMake code that depends on the variable):

```
set(MY_CACHE_VARIABLE "VALUE" CACHE INTERNAL "")
```

Since `BOOL` is such a common variable type, you can set it more succinctly with the shortcut:

```
option(MY_OPTION "This is settable from the command line" OFF)
```

For the `BOOL` datatype, there are several different wordings for `ON` and `OFF`.

See cmake-variables for a listing of known variables in CMake.

# Environment variables

You can also `set(ENV{variable_name} value)` and get `$ENV{variable_name}` environment variables, though it is generally a very good idea to avoid them.

# The Cache

The cache is actually just a text file, `CMakeCache.txt`, that gets created in the build directory when you run CMake. This is how CMake remembers anything you set, so you don't have to re-list your options every time you rerun CMake.

# Properties

The other way CMake stores information is in properties. This is like a variable, but it is attached to some other item, like a directory or a target. A global property can be a useful uncached global variable. Many target properties are initialized from a matching variable with `CMAKE_` at the front. So setting `CMAKE_CXX_STANDARD`, for example, will mean that all new targets created will have `CXX_STANDARD` set to that when they are created. There are two ways to set properties:

```
set_property(TARGET TargetName
             PROPERTY CXX_STANDARD 11)

set_target_properties(TargetName PROPERTIES
                      CXX_STANDARD 11)
```

The first form is more general, and can set multiple targets/files/tests at once, and has useful options. The second is a shortcut for setting several properties on one target. And you can get properties similarly:

```
get_property(ResultVariable TARGET TargetName PROPERTY CXX_STANDARD)
```

See cmake-properties for a listing of all known properties. You can also make your own in some cases.[2]

> [1]. `if` statements are a bit odd in that they can take the variable with or without the surrounding syntax; this is there for historical reasons: `if` predates the `${}` syntax. ↩

> [2]. Interface targets, for example, may have limits on custom properties that are allowed. ↩

# Programming in CMake

## Control flow

CMake has an `if` statement, though over the years it has become rather complex. There are a series of all caps keywords you can use inside an if statement, and you can often refer to variables by either directly by name or using the `${}` syntax (the if statement historically predates variable expansion). An example if statement:

```
if(variable)
    # If variable is `ON`, `YES`, `TRUE`, `Y`, or non zero number
else()
    # If variable is `0`, `OFF`, `NO`, `FALSE`, `N`, `IGNORE`, `NOTFOUND`, `""`, or ends in `-NOTFOUND`
endif()
# If variable does not expand to one of the above, CMake will expand it then try again
```

Since this can be a little confusing if you explicitly put a variable expansion, like `${variable}`, due to the potential expansion of an expansion, a policy (CMP0054) was added in CMake 3.1+ that keeps a quoted expansion from being expanded yet again. So, as long as the minimum version of CMake is 3.1+, you can do:

```
if("${variable}")
    # True if variable is not false-like
else()
    # Note that undefined variables would be `""` thus false
endif()
```

There are a variety of keywords as well, such as:

- Unary: `NOT`, `TARGET`, `EXISTS` (file), `DEFINED`, etc.
- Binary: `STREQUAL`, `AND`, `OR`, `MATCHES` (regular expression), `VERSION_LESS`, `VERSION_LESS_EQUAL` (CMake 3.7+), etc.
- Parentheses can be used to group

## Generator-expressions

Generator-expressions are really powerful, but a bit odd and specialized. Most CMake commands happen at configure time, include the if statements seen above. But what if you need logic to occur at build time or even install time? Generator expressions were added for this purpose.[1] They are evaluated in target properties.

The simplest generator expressions are informational expressions, and are of the form `$<KEYWORD>`; they evaluate to a piece of information relevant for the current configuration. The other form is `$<KEYWORD:value>`, where `KEYWORD` is a keyword that controls the evaluation, and value is the item to evaluate (an informational expression keyword is allowed here, too). If KEYWORD is a generator expression or variable that evaluates to 0 or 1, `value` is substituted if 1 and not if 0. You can nest generator expressions, and you can use variables to make reading nested variables bearable. Some expressions allow multiple values, separated by commas.[2]

If you want to put a compile flag only for the DEBUG configuration, for example, you could do:

```
target_compile_options(MyTarget PRIVATE "$<$<CONFIG:Debug>:--my-flag>")
```

This is a newer, better way to add things than using specialized `*_DEBUG` variables, and generalized to all the things generator expressions support. Note that you should never, never use the configure time value for the current configuration, because multi-configuration generators like IDEs do not have a "current" configuration at configure time, only at build time through generator expressions and custom `*_<CONFIG>` variables.

Other common uses for generator expressions:

- Limiting an item to a certain language only, such as CXX, to avoid it mixing with something like CUDA, or wrapping it so that it is different depending on target language.
- Accessing configuration dependent properties, like target file location.
- Giving a different location for build and install directories.

That last one is very common. You'll see something like this in almost every package that supports installing:

```
target_include_directories(
    MyTarget
  PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
)
```

# Macros and Functions

You can define your own CMake `function` or `macro` easily. The only difference between a function and a macro is scope; macros don't have one. So, if you set a variable in a function and want it to be visible outside, you'll need `PARENT_SCOPE`. Nesting functions therefore is a bit tricky, since you'll have to explicitly set the variables you want visible to the outside world to `PARENT_SCOPE` in each function. But, functions don't "leak" all their variables like macros do. For the following examples, I'll use functions.

An example of a simple function is as follows:

```
function(SIMPLE REQUIRED_ARG)
    message(STATUS "Simple arguments: ${REQUIRED_ARG}, followed by ${ARGV}")
    set(${REQUIRED_ARG} "From SIMPLE" PARENT_SCOPE)
endfunction()

simple(This)
message("Output: ${This}")
```

If you want positional arguments, they are listed explicitly, and all other arguments are collected in `ARGN` (`ARGV` holds all arguments, even the ones you list). You have to work around the fact that CMake does not have return values by setting variables. In the example above, you can explicitly give a variable name to set.

# Arguments

CMake has a named variable system that you've already seen in most of the build in CMake functions. You can use it with the `cmake_parse_arguments` function. If you want to support a version of CMake less than 3.5, you'll want to also include the CMakeParseArguments module, which is where it used to live before becoming a built in command. Here is an example of how to use it:

```
function(COMPLEX)
    cmake_parse_arguments(
        COMPLEX_PREFIX
        "SINGLE;ANOTHER"
        "ONE_VALUE;ALSO_ONE_VALUE"
        "MULTI_VALUES"
        ${ARGN}
    )
endfunction()

complex(SINGLE ONE_VALUE value MULTI_VALUES some other values)
```

Inside the function after this call, you'll find:

```
COMPLEX_PREFIX_SINGLE = TRUE
COMPLEX_PREFIX_ANOTHER = FALSE
COMPLEX_PREFIX_ONE_VALUE = "value"
COMPLEX_PREFIX_ALSO_ONE_VALUE = <UNDEFINED>
```

```
COMPLEX_PREFIX_MULTI_VALUES = "some;other;values"
```

If you look at the official page, you'll see a slightly different method using set to avoid explicitly writing the semicolons in the list; feel free to use the structure you like best. You can mix it with the positional arguments listed above; any remaining arguments (therefore optional positional arguments) are in `COMPLEX_PREFIX_UNPARSED_ARGUMENTS`.

> [1]. They act as if they are evaluated at build/install time, though actually they are evaluated for each build configuration. ↩

> [2]. The CMake docs splits expressions into Informational, Logical, and Output. ↩

# Communication with your code

## Configure File

CMake allows you to access CMake variables from your code using `configure_file`. This command copies a file (traditionally ending in `.in` from one place to another, substituting all CMake variables it finds. If you want to avoid replacing existing `${}` syntax in your input file, use the `@ONLY` keyword. There's also a `COPY_ONLY` keyword if you are just using this as a replacement for `file(COPY`.

This functionality is used quite frequently; for example, on `Version.h.in`:

### Version.h.in

```
#pragma once

#define MY_VERSION_MAJOR @PROJECT_VERSION_MAJOR@
#define MY_VERSION_MINOR @PROJECT_VERSION_MINOR@
#define MY_VERSION_PATCH @PROJECT_VERSION_PATCH@
#define MY_VERSION_TWEAK @PROJECT_VERSION_TWEAK@
#define MY_VERSION "@PROJECT_VERSION@"
```

### CMake lines:

```
configure_file (
    "${PROJECT_SOURCE_DIR}/include/My/Version.h.in"
    "${PROJECT_BINARY_DIR}/include/My/Version.h"
)
```

You should include the binary include directory as well when building your project. If you want to put any true/false variables in a header, CMake has C specific `#cmakedefine` and `#cmakedefine01` replacements to make appropriate define lines.

You can also (and often do) use this to produce `.cmake` files, such as the configure files (see installing).

## Reading files

The other direction can be done too; you can read in something (like a version) from your source files. If you have a header only library that you'd like to make available with or without CMake, for example, then this would be the best way to handle a version. This would look something like this:

```
# Assuming the canonical version is listed in a single line
# This would be in several parts if picking up from MAJOR, MINOR, etc.
set(VERSION_REGEX "#define MY_VERSION[ \t]+\"(.+)\"")

# Read in the line containing the version
file(STRINGS "${CMAKE_CURRENT_SOURCE_DIR}/include/My/Version.hpp"
    VERSION_STRING REGEX ${VERSION_REGEX})

# Pick out just the version
string(REGEX REPLACE ${VERSION_REGEX} "\\1" VERSION_STRING "${VERSION_STRING}")

# Automatically getting PROJECT_VERSION_MAJOR, My_VERSION_MAJOR, etc.
project(My LANGUAGES CXX VERSION ${VERSION_STRING})
```

Above, `file(STRINGS file_name variable_name REGEX regex)` picks lines that match a regex; and the same regex is used to then pick out the parentheses capture group with the version part. Replace is used with back substitution to output only that one group.

# How to structure your project

The following information is biased. But in a good way, I think. I'm going to tell you how to structure the directories in your project. This is based on convention, but will help you:

- Easily read other projects following the same patters,
- Avoid a pattern that causes conflicts,
- Keep from muddling and complicating your build.

First, this is what your files should look like when you start if your project is creatively called `project`, with a library called `lib`, and a executable called `app`:

```
- project
  - .gitignore
  - README.md
  - LICENCE.md
  - CMakeLists.txt
  - cmake
    - FindSomeLib.cmake
    - something_else.cmake
  - include
    - project
      - lib.hpp
  - src
    - CMakeLists.txt
    - lib.cpp
  - apps
    - CMakeLists.txt
    - app.cpp
  - tests
    - CMakeLists.txt
    - testlib.cpp
  - docs
    - CMakeLists.txt
  - extern
    - googletest
  - scripts
    - helper.py
```

The names are not absolute; you'll see contention about `test/` vs. `tests/`, and the application folder may be called something else (or not exist for a library-only project). You'll also sometime see a `python` folder for python bindings, or a `cmake` folder for helper CMake files, like `Find<library>.cmake` files. But the basics are there.

Notice a few things already apparent; the `CMakeLists.txt` files are split up over all source directories, and are not in the include directories. This is because you should be able to copy the contents of the include directory to `/usr/include` or similar directly (except for configuration headers, which I go over in another chapter), and not have any extra files or cause any conflicts. That's also why there is a directory for your project inside the include directory. Use `add_subdirectory` to add a subdirectory containing a `CMakeLists.txt`.

You often want a `cmake` folder, with all of your helper modules. This is where your `Find*.cmake` files go. An set of some common helpers is at github.com/CLIUtils/cmake. To add this folder to your CMake path:

```
set(CMAKE_MODULE_PATH "${PROJECT_SOURCE_DIR}/cmake" ${CMAKE_MODULE_PATH})
```

Your `extern` folder should contain git submodules almost exclusively. That way, you can control the version of the dependencies explicitly, but still upgrade easily. See the Testing chapter for an example of adding a submodule.

You should have something like `/build*` in your `.gitignore`, so that users can make build directories in the source directory and use those to build. A few packages prohibit this, but it's much better than doing a true out-of-source build and having to type something different for each package you build.

If you want to avoid the build directory being in a valid source directory, add this near the top of your CMakeLists:

```
### Require out-of-source builds
file(TO_CMAKE_PATH "${PROJECT_BINARY_DIR}/CMakeLists.txt" LOC_PATH)
if(EXISTS "${LOC_PATH}")
    message(FATAL_ERROR "You cannot build in a source directory (or any directory with a CMakeLists.txt file). Please make a b
uild subdirectory. Feel free to remove CMakeCache.txt and CMakeFiles.")
endif()
```

See the extended code example here.

# Running other programs

## Running a command at configure time

Running a command at configure time is relatively easy. Use `execute_process` to run a process and access the results. It is generally a good idea to avoid hard coding a program path into your CMake; you can use `${CMAKE_COMMAND}`, `find_package(Git)`, or `find_program` to get access to a command to run. Use `RESULT_VARIABLE` to check the return code and `OUTPUT_VARIABLE` to get the output.

Here is an example that updates all git submodules:

```
find_package(Git QUIET)

if(GIT_FOUND AND EXISTS "${PROJECT_SOURCE_DIR}/.git")
    execute_process(COMMAND ${GIT_EXECUTABLE} submodule update --init --recursive
                    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
                    RESULT_VARIABLE GIT_SUBMOD_RESULT)
    if(NOT GIT_SUBMOD_RESULT EQUAL "0")
        message(FATAL_ERROR "git submodule update --init failed with ${GIT_SUBMOD_RESULT}, please checkout submodules")
    endif()
endif()
```

## Running a command at build time

Build time commands are a bit tricker. The main complication comes from the target system; when do you want your command to run? Does it produce an output that another target needs? With this in mind, here is an example that calls a Python script to generate a header file:

```
find_package(PythonInterp REQUIRED)
add_custom_command(OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp"
    COMMAND "${PYTHON_EXECUTABLE}" "${CMAKE_CURRENT_SOURCE_DIR}/scripts/GenerateHeader.py" --argument
    DEPENDS some_target)

add_custom_target(generate_header ALL
    DEPENDS "${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp")

install(FILES ${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp DESTINATION include)
```

Here, the generation happens after `some_target` is complete, and happens when you run make without a target (`ALL`). If you make this a dependency of another target with `add_dependencies`, you could avoid the `ALL` keyword. Or, you could require that a user explicitly builds the `generate_header` target when making.

## Included common utilities

A useful tool in writing CMake builds that work cross-platform is `cmake -E <mode>` (seen in CMake files as `${CMAKE_COMMAND} -E`). This mode allows CMake to do a variety of things without calling system tools explicitly, like `copy`, `make_directory`, and `remove`. It is mostly used for the build time commands. Note that the very useful `create_symlink` mode used to be Unix only, but was added for Windows in CMake 3.13. See the docs.

# A simple example

This is a simple yet complete example of a proper CMakeLists. For this program, we have one library (MyLibExample) with a header file and a source file, and one application, MyExample, with one source file.

```
# Almost all CMake files should start with this
# You should always specify a range with the newest
# and oldest tested versions of CMake. This will ensure
# you pick up the best policies.
cmake_minimum_required(VERSION 3.1...3.16)

# This is your project statement. You should always list languages;
# Listing the version is nice here since it sets lots of useful variables
project(ModernCMakeExample VERSION 1.0 LANGUAGES CXX)

# If you set any CMAKE_ variables, that can go here.
# (But usually don't do this, except maybe for C++ standard)

# Find packages go here.

# You should usually split this into folders, but this is a simple example

# This is a "default" library, and will match the *** variable setting.
# Other common choices are STATIC, SHARED, and MODULE
# Including header files here helps IDEs but is not required.
# Output libname matches target name, with the usual extensions on your system
add_library(MyLibExample simple_lib.cpp simple_lib.hpp)

# Link each target with other targets or add options, etc.

# Adding something we can run - Output name matches target name
add_executable(MyExample simple_example.cpp)

# Make sure you link your targets with this command. It can also link libraries and
# even flags, so linking a target that does not exist will not give a configure-time error.
target_link_libraries(MyExample PRIVATE MyLibExample)
```

The complete example is available in examples folder.

A larger, multi-file example is also available.

# Adding features

This section covers adding common features to your CMake project. You'll learn how to add a variety of options commonly needed in C++ projects, like C++11 support, as well as how to support IDEs and more.

## Default build type

CMake normally does a "non-release, non debug" empty build type; if you prefer to set the default build type yourself, you can follow this recipe for the default build type modified from the Kitware blog:

```
set(default_build_type "Release")
if(NOT CMAKE_BUILD_TYPE AND NOT CMAKE_CONFIGURATION_TYPES)
  message(STATUS "Setting build type to '${default_build_type}' as none was specified.")
  set(CMAKE_BUILD_TYPE "${default_build_type}" CACHE
      STRING "Choose the type of build." FORCE)
  # Set the possible values of build type for cmake-gui
  set_property(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS
    "Debug" "Release" "MinSizeRel" "RelWithDebInfo")
endif()
```

# C++11 and beyond

C++11 is supported by CMake. Really. Just not in CMake 2.8, because, guess what, C++11 didn't exist in 2009 when 2.0 was released. As long as you are using CMake 3.1 or newer, you should be fine, there are two different ways to enable support. And as you'll soon see, there's even better support in CMake 3.8+. I'll start with that, since this is Modern CMake.

## CMake 3.8+: Meta compiler features

As long as you can require that a user install CMake, you'll have access to the newest way to enable C++ standards. This is the most powerful way, with the nicest syntax and the best support for new standards, and the best target behavior for mixing standards and options. Assuming you have a target named `myTarget`, it looks like this:

```
target_compile_features(myTarget PUBLIC cxx_std_11)
set_target_properties(myTarget PROPERTIES CXX_EXTENSIONS OFF)
```

For the first line, we get to pick between `cxx_std_11`, `cxx_std_14`, and `cxx_std_17`. The second line is optional, but will avoid extensions being added; without it you'd get things like `-std=g++11` replacing `-std=c++11`. The first line even works on `INTERFACE` targets; only actual compiled targets can use the second line.

If a target further down the dependency chain specifies a higher C++ level, this interacts nicely. It's really just a more advanced version of the following method, so it interacts nicely with that, too.

## CMake 3.1+: Compiler features

You can ask for specific compiler features to be available. This was more granular than asking for a C++ version, though it's a bit tricky to pick out just the features a package is using unless you wrote the package and have a good memory. Since this ultimately checks against a list of options CMake knows your compiler supports and picks the highest flag indicated in that list, you don't have to specify all the options you need, just the rarest ones. The syntax is identical to the section above, you just have a list of options to pick instead of `cxx_std_*` options. See the whole list here.

If you have optional features, you can use the list `CMAKE_CXX_COMPILE_FEATURES` and use `if(... IN_LIST ...)` from CMake 3.3+ to see if that feature is supported, and add it conditionally. See the docs here for other use cases.

A related feature, `WriteCompilerDetectionHeader`, is worth checking out. It is a module that lets you make a file with macros allowing you to check and support optional features for specific compilers. Like any header generator, this will require that you build with CMake so that your header can be generated as part of the build process (only important if you care about supporting multiple build systems, or if you are making a no-build header-only library).

## CMake 3.1+: Global and property settings

There is another way that C++ standards are supported; a specific set of three properties (both global and target level). The global properties are:

```
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

The first line sets a C++ standard level, and the second tells CMake to use it, and the final line is optional and ensures `-std=c++11` vs. something like `-std=g++11`. This method isn't bad for a final package, but shouldn't be used by a library. You can also set these values on a target:

```
set_target_properties(myTarget PROPERTIES
    CXX_STANDARD 11
    CXX_STANDARD_REQUIRED YES
    CXX_EXTENSIONS NO
)
```

Which is better, but still doesn't have the sort of explicit control that compiler features have for populating `PRIVATE` and `INTERFACE` properties.

You can find more information about the final two methods on Craig Scott's useful blog post.

> Don't set manual flags yourself. You'll then become responsible for mainting correct flags for every release of every compiler, error messages for unsupported compilers won't be useful, and some IDEs might not pick up the manual flags.

# Adding Features

There are lots of compiler and linker settings. When you need to add something special, you could check first to see if CMake supports it; if it does, you can avoid explicitly tying yourself to a compiler version. And, better yet, you explain what you mean in your CMakeLists, rather than spewing flags.

The first and most common feature was C++ standards support, which got it's own chapter.

# Position independent code

This is best known as the `-fPIC` flag. Much of the time, you don't need to do anything. CMake will include the flag for `SHARED` or `MODULE` libraries. If you do explicitly need it:

```
set(CMAKE_POSITION_INDEPENDENT_CODE ON)
```

will do it globally, or:

```
set_target_properties(lib1 PROPERTIES POSITION_INDEPENDENT_CODE ON)
```

to explicitly turn it `ON` (or `OFF`) for a target.

# Little libraries

If you need to link to the `dl` library, with `-ldl` on Linux, just use the built-in CMake variable `${CMAKE_DL_LIBS}` in a `target_link_libraries` command. No module or `find_package` needed. (This adds whatever is needed to get `dlopen` and `dlclose`)

Unfortunately, the math library is not so lucky. If you need to explicitly link to it, you can always do `target_link_libraries(MyTarget PUBLIC m)`, but it might be better to use CMake's generic `find_library`:

```
find_library(MATH_LIBRARY m)
if(MATH_LIBRARY)
    target_link_libraries(MyTarget PUBLIC ${MATH_LIBRARY})
endif()
```

You can pretty easily find `Find*.cmake`'s for this and other libraries that you need with a quick search; most major packages have a helper library of CMake modules. See the chapter on existing package inclusion for more.

# Interprocedural optimization

`INTERPROCEDURAL_OPTIMIZATION`, best known as *link time optimization* and the `-flto` flag, is available on very recent versions of CMake. You can turn this on with `CMAKE_INTERPROCEDURAL_OPTIMIZATION` (CMake 3.9+ only) or the `INTERPROCEDURAL_OPTIMIZATION` property on targets. Support for GCC and Clang was added in CMake 3.8. If you set `cmake_minimum_required(VERSION 3.9)` or better (see CMP0069), setting this to `ON` on a target is an error if the compiler doesn't support it. You can use check_ipo_supported(), from the built-in CheckIPOSupported module, to see if support is available before hand. An example of 3.9 style usage:

```
include(CheckIPOSupported)
check_ipo_supported(RESULT result)
if(result)
  set_target_properties(foo PROPERTIES INTERPROCEDURAL_OPTIMIZATION TRUE)
endif()
```

# CCache and Utilities

Over the versions, common utilities that help you write good code have had support added to CMake. This is usually in the form of a property and matching `CMAKE_*` initialization variable. The feature is not meant to be tied to one special program, but rather any program that is somewhat similar in behavior.

All of these take `;` separated values (a standard list in CMake) that describe the program and options that you should run on the source files of this target.

# CCache

Set the `CMAKE_<LANG>_COMPILER_LAUNCHER` variable or the `<LANG>_COMPILER_LAUNCHER` property on a target to use something like CCache to "wrap" the compilation of the target. Support for CCache has been expanding in the latest versions of CMake. In practice, this tends to look like this:

```
find_program(CCACHE_PROGRAM ccache)
if(CCACHE_PROGRAM)
    set(CMAKE_CXX_COMPILER_LAUNCHER "${CCACHE_PROGRAM}")
    set(CMAKE_CUDA_COMPILER_LAUNCHER "${CCACHE_PROGRAM}") # CMake 3.9+
endif()
```

# Utilities

Set the following properties or `CMAKE_*` initializer variables to the command line for the tools. Most of them are limited to C or CXX with make or ninja generators.

- `<LANG>_CLANG_TIDY` : CMake 3.6+
- `<LANG>_CPPCHECK`
- `<LANG>_CPPLINT`
- `<LANG>_INCLUDE_WHAT_YOU_USE`

# Clang tidy

Here is a simple example of using Clang-Tidy:

```
if(CMAKE_VERSION VERSION_GREATER 3.6)
    # Add clang-tidy if available
    option(CLANG_TIDY_FIX "Perform fixes for Clang-Tidy" OFF)
    find_program(
        CLANG_TIDY_EXE
        NAMES "clang-tidy"
        DOC "Path to clang-tidy executable"
    )

    if(CLANG_TIDY_EXE)
        if(CLANG_TIDY_FIX)
            set(CMAKE_CXX_CLANG_TIDY "${CLANG_TIDY_EXE}" "-fix")
        else()
            set(CMAKE_CXX_CLANG_TIDY "${CLANG_TIDY_EXE}")
        endif()
    endif()
endif()
```

The `-fix` part is optional, and will modify your source files to try to fix the tidy warning issued. If you are working in a git repository, this is fairly safe as you can see what has changed. However, make sure you **do not run your makefile/ninja build in parallel**! This will not work very well at all if it tries to fix the same header twice.

If you want to explicitly use the target form to ensure you only call this on your local targets, you can set a variable (I usually chose `DO_CLANG_TIDY`) instead of the `CMAKE_CXX_CLANG_TIDY` variable, then add it to your target properties as you create them.

# Include what you use

This is an example for using include what you use. First, you'll need to have the tool, such as in a docker container:

```
gitbook $ docker run --rm -it tuxity/include-what-you-use:clang_4.0
```

Then, you can pass this into your build without modifying the source:

```
build # cmake .. -DCMAKE_CXX_INCLUDE_WHAT_YOU_USE=include-what-you-use
```

Finally, you can collect the output and apply the fixes:

```
build # make 2> iwyu.out
build # fix_includes.py < iwyu.out
```

# Link what you use

There is a boolean target property, `LINK_WHAT_YOU_USE`, that will check for extraneous files when linking.

# Clang-format

Clang-format doesn't really have an integration with CMake, unfortunately. You could make a custom target (See this post, or you can run it manually. An interesting project that I have not really tried is here; it adds a format target and even makes sure that you can't commit unformatted files.

The following two line would do that in a git repository in bash (assuming you have a `.clang-format` file):

```
gitbook $ git ls-files -- '*.cpp' '*.h' | xargs clang-format -i -style=file
gitbook $ git diff --exit-code --color
```

# Useful Modules

There are a ton of useful modules in CMake's modules collection; but some of them are more useful than others. Here are a few highlights.

## CMakeDependentOption

This adds a command `cmake_dependent_option` that sets an option based on another set of variables being true. It looks like this:

```
include(CMakeDependentOption)
cmake_dependent_option(BUILD_TESTS "Build your tests" ON "VAL1;VAL2" OFF)
```

which is just a shortcut for this:

```
if(VAL1 AND VAL2)
    set(BUILD_TESTS_DEFAULT ON)
else()
    set(BUILD_TESTS_DEFAULT OFF)
endif()

option(BUILD_TESTS "Build your tests" ${BUILD_TESTS_DEFAULT})

if(NOT BUILD_TESTS_DEFAULT)
    mark_as_advanced(BUILD_TESTS)
endif()
```

Note that `BUILD_TESTING` is a better way to check for testing being enabled if you use `include(CTest)`, since it is defined for you. This is just an example of `CMakeDependentOption`.

## CMakePrintHelpers

This module has a couple of handy output functions. `cmake_print_properties` lets you easily print properties. And `cmake_print_variables` will print the names and values of any variables you give it.

## CheckCXXCompilerFlag

This checks to see if a flag is supported. For example:

```
include(CheckCXXCompilerFlag)
check_cxx_compiler_flag(-someflag OUTPUT_VARIABLE)
```

Note that `OUTPUT_VARIABLE` will also appear in the configuration printout, so choose a good name.

This is just one of many similar modules, such as `CheckIncludeFileCXX`, `CheckStructHasMember`, `TestBigEndian`, and `CheckTypeSize` that allow you to check for information about the system (and you can communicate that to your source code).

## WriteCompilerDetectionHeader

This is an amazing module similar to the ones listed above, but special enough to deserve its own section. It allows you to look for a list of features that some compilers support, and write out a C++ header file that lets you know whether that feature is available. It even can provide compatibility macros for features that have changed names!

To use:

```
write_compiler_detection_header(
  FILE myoutput.h
  PREFIX My
  COMPILERS GNU Clang MSVC Intel
  FEATURES cxx_variadic_templates
)
```

This supports compiler features (defined to 0 or 1), symbols (defined to empty or the symbol), and macros that support different names. They will be prefixed with the PREFIX you provide. You can separate compilers into different files using `OUTPUT_FILES_DIR`.

The downside is that you do have to list the compilers you expect to support. If you use the `ALLOW_UNKNOWN_COMPILERS` flag(s), you can keep this from erroring on unknown compilers, but it will still leave all features empty.

### `try_compile` / `try_run`

This is not exactly a module, but is crucial to many of the modules listed above. You can attempt to compile (and possibly run) a bit of code at configure time. This can allow you to get information about the capabilities of your system. The basic syntax is:

```
try_compile(
  RESULT_VAR
    bindir
  SOURCES
    source.cpp
)
```

There are lots of options you can add, like `COMPILE_DEFINITIONS`. In CMake 3.8+, this will honor the CMake C/C++/CUDA standard settings. If you use `try_run` instead, it will run the resulting program and give you the output in `RUN_OUTPUT_VARIABLE`.

## FeatureSummary

This is a fairly useful but rather odd module. It allows you to print out a list of packages what were searched for, as well as any options you explicity mark. It's partially but not completely tied into `find_package`. You first include the module, as always:

```
include(FeatureSummary)
```

Then, for any find packages you have run or will run, you can extend the default information:

```
set_package_properties(OpenMP PROPERTIES
    URL "http://www.openmp.org"
    DESCRIPTION "Parallel compiler directives"
    PURPOSE "This is what it does in my package")
```

You can also set the `TYPE` of a package to `RUNTIME`, `OPTIONAL`, `RECOMMENDED`, or `REQUIRED`; you can't, however, lower the type of a package; if you have already added a `REQUIRED` package through `find_package` based on an option, you'll see it listed as `REQUIRED`.

And, you can mark any options as part of the feature summary. If you choose the same name as a package, the two interact with each other.

```
add_feature_info(WITH_OPENMP OpenMP_CXX_FOUND "OpenMP (Thread safe FCNs only)")
```

Then, you can print out the summary of features, either to the screen or a log file:

```
if(CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME)
    feature_summary(WHAT ENABLED_FEATURES DISABLED_FEATURES PACKAGES_FOUND)
```

```
    feature_summary(FILENAME ${CMAKE_CURRENT_BINARY_DIR}/features.log WHAT ALL)
endif()
```

You can build any collection of `WHAT` items that you like, or just use `ALL`.

# Supporting IDEs

In general, IDEs are already supported by a standard CMake project. There are just a few extra things that can help IDEs perform even better.

# Folders for targets

Some IDEs, like Xcode, support folders. You have to manually enable the `USE_FOLDERS` global property to allow CMake to organize your files by folders:

```
set_property(GLOBAL PROPERTY USE_FOLDERS ON)
```

Then, you can add targets to folders after you create them:

```
set_property(TARGET MyFile PROPERTY FOLDER "Scripts")
```

Folders can be nested with `/`.

You can control how files show up in each folder with regular expressions or explicit listings in `source_group`:

# Folders for files

You can also control how the folders inside targets appear. There are two ways, both using the `source_group` command. The traditional way is

```
source_group("Source Files\\New Directory" REGULAR_EXPRESSION ".*\\.c[ucp]p?")
```

You can explicitly list files with `FILES`, or use a `REGULAR_EXPRESSION`. This way you have complete control over the folder structure. However, if your on-disk layout is well designed, you might just want to mimic that. In CMake 3.8+, you can do so very easily with a new version of the `source_group` command:

```
source_group(TREE "${CMAKE_CURRENT_SOURCE_DIR}/base/dir" PREFIX "Header Files" FILES ${FILE_LIST})
```

For the `TREE` option, you should usually give a full path starting with something like `${CMAKE_CURRENT_SOURCE_DIR}/` (because the command interprets paths relative to the build directory). The prefix tells you where it puts it into the IDE structure, and the `FILES` option takes a list of files. CMake will strip the `TREE` path from the `FILE_LIST` path, it will add `PREFIX`, and that will be the IDE folder structure.

> Note: If you need to support CMake < 3.8, I would recommend just protecting the above command, and only supporting nice folder layout on CMake 3.8+. For older methods to do this folder layout, see this blog post.

# Running with an IDE

To use an IDE, either pass `-G"name of IDE"` if CMake can produce that IDE's files (like Xcode, Visual Studio), or open the CMakeLists.txt file from your IDE if that IDE has built in support for CMake (CLion, QtCreator, many others).

# Debugging code

You might need to debug your CMake build, or debug your C++ code. Both are covered here.

# CMake debugging

First, let's look at ways to debug a CMakeLists or other CMake file.

## Printing variables

The time honored method of print statements looks like this in CMake:

```
message(STATUS "MY_VARIABLE=${MY_VARIABLE}")
```

However, a built in module makes this even easier:

```
include(CMakePrintHelpers)
cmake_print_variables(MY_VARIABLE)
```

If you want to print out a property, this is much, much nicer! Instead of getting the properties one by one of of each target (or other item with properties, such as `SOURCES`, `DIRECTORIES`, `TESTS`, or `CACHE_ENTRIES` - global properties seem to be missing for some reason), you can simply list them and get them printed directly:

```
cmake_print_properties(
    TARGETS my_target
    PROPERTIES POSITION_INDEPENDENT_CODE
)
```

## Tracing a run

Have you wanted to watch exactly what happens in your CMake file, and when? The `--trace-source="filename"` feature is fantastic. Every line run in the file that you give will be echoed to the screen when it is run, letting you follow exactly what is happening. There are related options as well, but they tend to bury you in output.

For example:

```
cmake -S . -B build --trace-source=CMakeLists.txt
```

If you add `--trace-expand`, the variables will be expanded into their values.

# Building in debug mode

For single-configuration generators, you can build your code with `-DCMAKE_BUILD_TYPE=Debug` to get debugging flags. In multi-configuration generators, like many IDEs, you can pick the configuration in the IDE. There are distinct flags for this mode (variables ending in `_DEBUG` as opposed to `_RELEASE`), as well as a generator expression value `CONFIG:Debug` or `CONFIG:Release`.

Once you make a debug build, you can run a debugger, such as gdb or lldb on it.

# Including Small Projects

This is where a good Git system plus CMake shines. You might not be able to solve all the world's problems, but this is pretty close for C++!

There are several methods listed in the chapters in this section.

# Git Submodule Method

If you want to add a Git repository on the same service (GitHub, GitLab, BitBucket, etc), the following is the correct Git command to set that up as a submodule in the `extern` directory:

```
gitbook $ git submodule add ../../owner/repo.git extern/repo
```

The relative path to the repo is important; it allows you to keep the same access method (ssh or https) as the parent repository. This works very well in most ways. When you are inside the submodule, you can treat it just like a normal repo, and when you are in the parent repository, you can "add" to change the current commit pointer.

But the traditional downside is that you either have to have your users know git submodule commands, so they can `init` and `update` the repo, or they have to add `--recursive` when they initially clone your repo. CMake can offer a solution:

```
find_package(Git QUIET)
if(GIT_FOUND AND EXISTS "${PROJECT_SOURCE_DIR}/.git")
# Update submodules as needed
    option(GIT_SUBMODULE "Check submodules during build" ON)
    if(GIT_SUBMODULE)
        message(STATUS "Submodule update")
        execute_process(COMMAND ${GIT_EXECUTABLE} submodule update --init --recursive
                        WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
                        RESULT_VARIABLE GIT_SUBMOD_RESULT)
        if(NOT GIT_SUBMOD_RESULT EQUAL "0")
            message(FATAL_ERROR "git submodule update --init failed with ${GIT_SUBMOD_RESULT}, please checkout submodules")
        endif()
    endif()
endif()

if(NOT EXISTS "${PROJECT_SOURCE_DIR}/extern/repo/CMakeLists.txt")
    message(FATAL_ERROR "The submodules were not downloaded! GIT_SUBMODULE was turned off or failed. Please update submodules
and try again.")
endif()
```

The first line checks for Git using CMake's built in `FindGit.cmake`. Then, if you are in a git checkout of your source, add an option (defaulting to `ON`) that allows developers to turn off the feature if they need to. We then run the command to get all repositories, and fail if that command fails, with a nice error message. Finally, we verify that the repositories exist before continuing, regardless of the method used to obtain them. You can use `OR` to list several.

Now, your users can be completely oblivious to the existence of the submodules, and you can still keep up good development practices! The only thing to watch out for is for developers; you will reset the submodule when you rerun CMake if you are developing inside the submodule. Just add new commits to the parent staging area, and you'll be fine.

You can then include projects that provide good CMake support:

```
add_subdirectory(extern/repo)
```

Or, you can build an interface library target yourself if it is a header only project. Or, you can use `find_package` if that is supported, probably preparing the initial search directory to be the one you've added (check the docs or the file for the `Find*.cmake` file you are using). You can also include a CMake helper file directory if you append to your `CMAKE_MODULE_PATH`, for example to add `pybind11`'s improved `FindPython*.cmake` files.

## Bonus: Git version number

Move this to Git section:

```
execute_process(COMMAND ${GIT_EXECUTABLE} rev-parse --short HEAD
                WORKING_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}"
```

```
                OUTPUT_VARIABLE PACKAGE_GIT_VERSION
                ERROR_QUIET
                OUTPUT_STRIP_TRAILING_WHITESPACE)
```

# GoogleTest: Download method

## Downloading Method: build time

Until CMake 3.11, the primary download method for packages was done at build time. This causes several issues; most important of which is that `add_subdirectory` doesn't work on a file that doesn't exist yet! The tool for this, ExternalProject, has to work around this by doing the build itself. (It can, however, build non-CMake packages as well).[1]

> [1]. Note that ExternalData is the tool for non-package data. ↩

## Downloading Method: configure time

If you prefer configure time, see the Crascit/DownloadProject repository for a drop-in solution. Submodules work so well, though, that I've discontinued most of the downloads for things like GoogleTest and moved them to submodules. Auto downloads are harder to mimic if you don't have internet access, and they are often implemented in the build directory, wasting time and space if you have multiple build directories.

# FetchContent (CMake 3.11+)

Often, you would like to do your download of data or packages as part of the configure instead of the build. This was invented several times in third party modules, but was finally added to CMake itself as part of CMake 3.11 as the FetchContent module.

The FetchContent module has excellent documentation that I won't try to repeat. The key ideas are:

- Use `FetchContent_Declare(MyName)` to get data or a package. You can set URLs, Git repositories, and more.
- Use `FetchContent_GetProperties(MyName)` on the name you picked in the first step to get `MyName_*` variables.
- Check `MyName_POPULATED`, and if not populated, use `FetchContent_Populate(MyName)` (and if a package, `add_subdirectory("${MyName_SOURCE_DIR}" "${MyName_BINARY_DIR}")`)

For example, to download Catch2:

```
FetchContent_Declare(
  catch
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
  GIT_TAG        v2.9.1
)

# CMake 3.14+
FetchContent_MakeAvailable(catch)
```

If you can't use CMake 3.14+, the classic way to prepare code was:

```
# CMake 3.11+
FetchContent_GetProperties(catch)
if(NOT catch_POPULATED)
  FetchContent_Populate(catch)
  add_subdirectory(${catch_SOURCE_DIR} ${catch_BINARY_DIR})
endif()
```

Of course, you could bundled this up into a macro:

```
if(${CMAKE_VERSION} VERSION_LESS 3.14)
    macro(FetchContent_MakeAvailable NAME)
        FetchContent_GetProperties(${NAME})
        if(NOT ${NAME}_POPULATED)
            FetchContent_Populate(${NAME})
            add_subdirectory(${${NAME}_SOURCE_DIR} ${${NAME}_BINARY_DIR})
        endif()
    endmacro()
endif()
```

Now you have the CMake 3.14+ syntax in CMake 3.11+.

# Testing

## General Testing Information

In your main CMakeLists.txt you need to add the following function call (not in a subfolder):

```
if(CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME)
    include(CTest)
endif()
```

Which will enable testing and set a `BUILD_TESTING` option so users can turn testing on and off (Along with a few other things). Or you can do this yourself by directly calling `enable_testing()`.

When you add your test folder, you should do something like this:

```
if(CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME AND BUILD_TESTING)
    add_subdirectory(tests)
endif()
```

The reason for this is that if someone else includes your package, and they use `BUILD_TESTING`, they probably do not want your tests to build. In the rare case that you really do want to enable testing on both packages, you can provide an override:

```
if((CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME OR MYPROJECT_BUILD_TESTING) AND BUILD_TESTING)
    add_subdirectory(tests)
endif()
```

The main use case for the override above is actually in this book's own examples, as the master CMake project really does want to run all the subproject tests.

You can register targets with:

```
add_test(NAME TestName COMMAND TargetName)
```

If you put something else besides a target name after COMMAND, it will register as a command line to run. It would also be valid to put the generator expression:

```
add_test(NAME TestName COMMAND $<TARGET_FILE:${TESTNAME}>)
```

which would use the output location (thus, the executable) of the produced target.

## Building as part of a test

If you want to run CMake to build a project as part of a test, you can do that too (in fact, this is how CMake tests itself). For example, if your master project was called `MyProject` and you had an `examples/simple` project that could build by itself, this would look like:

```
add_test(
  NAME
    ExampleCMakeBuild
  COMMAND
    "${CMAKE_CTEST_COMMAND}"
            --build-and-test "${My_SOURCE_DIR}/examples/simple"
                             "${CMAKE_CURRENT_BINARY_DIR}/simple"
            --build-generator "${CMAKE_GENERATOR}"
            --test-command "${CMAKE_CTEST_COMMAND}"
```

```
)
```

# Testing Frameworks

Look at the subchapters for recipes for popular frameworks.

- GoogleTest: A popular option from Google. Development can be a bit slow.
- Catch2: A modern, PyTest-like framework with clever macros.
- DocTest: A replacement for Catch2 that is supposed to compile much faster and be cleaner. See Catch2 chapter and replace with DocTest.

# GoogleTest

## Submodule method (preferred)

To use this method, just checkout GoogleTest as a submodule:[1]

```
git submodule add --branch=release-1.8.0 ../../google/googletest.git extern/googletest
```

Then, in your main `CMakeLists.txt`:

```
option(PACKAGE_TESTS "Build the tests" ON)
if(PACKAGE_TESTS)
    enable_testing()
    include(GoogleTest)
    add_subdirectory(tests)
endif()
```

I would recommend using something like `PROJECT_NAME STREQUAL CMAKE_PROJECT_NAME` to set the default for the `PACKAGE_TESTS` option, since this should only build by default if this is the current project. As mentioned before, you have to do the `enable_testing` in your main CMakeLists.

Now, in your tests directory:

```
add_subdirectory("${PROJECT_SOURCE_DIR}/extern/googletest" "extern/googletest")
```

If you did this in your main CMakeLists, you could use a normal `add_subdirectory`; the extra path here is needed to correct the build path because we are calling it from a subdirectory.

The next line is optional, but keeps your `CACHE` cleaner:

```
mark_as_advanced(
    BUILD_GMOCK BUILD_GTEST BUILD_SHARED_LIBS
    gmock_build_tests gtest_build_samples gtest_build_tests
    gtest_disable_pthreads gtest_force_shared_crt gtest_hide_internal_symbols
)
```

If you are interested in keeping IDEs that support folders clean, I would also add these lines:

```
set_target_properties(gtest PROPERTIES FOLDER extern)
set_target_properties(gtest_main PROPERTIES FOLDER extern)
set_target_properties(gmock PROPERTIES FOLDER extern)
set_target_properties(gmock_main PROPERTIES FOLDER extern)
```

Then, to add a test, I'd recommend the following macro:

```
macro(package_add_test TESTNAME)
    # create an exectuable in which the tests will be stored
    add_executable(${TESTNAME} ${ARGN})
    # link the Google test infrastructure, mocking library, and a default main fuction to
    # the test executable.  Remove g_test_main if writing your own main function.
    target_link_libraries(${TESTNAME} gtest gmock gtest_main)
    # gtest_discover_tests replaces gtest_add_tests,
    # see https://cmake.org/cmake/help/v3.10/module/GoogleTest.html for more options to pass to it
    gtest_discover_tests(${TESTNAME}
        # set a working directory so your project root so that you can find test data via paths relative to the project root
        WORKING_DIRECTORY ${PROJECT_DIR}
        PROPERTIES VS_DEBUGGER_WORKING_DIRECTORY "${PROJECT_DIR}"
```

```
    )
    set_target_properties(${TESTNAME} PROPERTIES FOLDER tests)
endmacro()

package_add_test(test1 test1.cpp)
```

This will allow you to quickly and simply add tests. Feel free to adjust to suit your needs. If you haven't seen it before, `ARGN` is "every argument after the listed ones". Modify the macro to meet your needs. For example, if you're testing libraries and need to link in different libraries for different tests, you might use this:

```
macro(package_add_test_with_libraries TESTNAME FILES LIBRARIES TEST_WORKING_DIRECTORY)
    add_executable(${TESTNAME} ${FILES})
    target_link_libraries(${TESTNAME} gtest gmock gtest_main ${LIBRARIES})
    gtest_discover_tests(${TESTNAME}
        WORKING_DIRECTORY ${TEST_WORKING_DIRECTORY}
        PROPERTIES VS_DEBUGGER_WORKING_DIRECTORY "${TEST_WORKING_DIRECTORY}"
    )
    set_target_properties(${TESTNAME} PROPERTIES FOLDER tests)
endmacro()

package_add_test_with_libraries(test1 test1.cpp lib_to_test "${PROJECT_DIR}/european-test-data/")
```

# Download method

You can use the downloader in my CMake helper repository, using CMake's `include` command.

This is a downloader for GoogleTest, based on the excellent DownloadProject tool. Downloading a copy for each project is the recommended way to use GoogleTest (so much so, in fact, that they have disabled the automatic CMake install target), so this respects that design decision. This method downloads the project at configure time, so that IDE's correctly find the libraries. Using it is simple:

```
cmake_minimum_required(VERSION 3.10)
project(MyProject CXX)
list(APPEND CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/cmake)

enable_testing() # Must be in main file

include(AddGoogleTest) # Could be in /tests/CMakeLists.txt
add_executable(SimpleTest SimpleTest.cu)
add_gtest(SimpleTest)
```

> Note: `add_gtest` is just a macro that adds `gtest`, `gmock`, and `gtest_main`, and then runs `add_test` to create a test with the same name:
>
> ```
> target_link_libraries(SimpleTest gtest gmock gtest_main)
> add_test(SimpleTest SimpleTest)
> ```

# FetchContent: CMake 3.11

The example for the FetchContent module is GoogleTest:

```
include(FetchContent)

FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG        release-1.8.0
)

FetchContent_GetProperties(googletest)
if(NOT googletest_POPULATED)
```

```
  FetchContent_Populate(googletest)
  add_subdirectory(${googletest_SOURCE_DIR} ${googletest_BINARY_DIR})
endif()
```

[1]. Here I've assumed that you are working on a GitHub repository by using the relative path to googletest. ↩

# Catch

Catch and Catch2 (C++11 only version) are powerful, idomatic testing solutions similar in philosophy to PyTest for Python. To use Catch in a CMake project, there are several options.

## Vendoring

If you simply drop in the single include release of Catch into your project, this is what you would need to add Catch:

```
# Prepare "Catch" library for other executables
set(CATCH_INCLUDE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/extern/catch)
add_library(Catch2::Catch IMPORTED INTERFACE)
set_property(Catch2::Catch PROPERTY INTERFACE_INCLUDE_DIRECTORIES "${CATCH_INCLUDE_DIR}")
```

Then, you would link to Catch2::Catch. This would have been okay as an INTERFACE target since you won't be exporting your tests.

## Direct inclusion

If you add the library using ExternalProject, FetchContent, or git submodules, you can also `add_subdirectory` Catch (CMake 3.1+).

Catch also provides two CMake modules that you can use to register the individual tests with CMake.

# Exporting and Installing

There are three good ways and one bad way to allow others use your library:

## Find module (the bad way)

If you are the library author, don't make a `Find<mypackage>.cmake` script! These were designed for libraries whose authors did not support CMake. Use a `Config<mypackage>.cmake` instead as listed below.

## Add Subproject

A package can include your project in a subdirectory, and then use `add_directory` on the subdirectory. This useful for header-only and quick-to-compile libraries. Note that the install commands may interfere with the parent project, so you can add `EXCLUDE_FROM_ALL` to the `add_subdirectory` command; the targets you explicitly use will still be built.

In order to support this as a library author, make sure you use `CMAKE_CURRENT_SOURCE_DIR` instead of `PROJECT_SOURCE_DIR` (and likewise for other variables, like binary dirs). You can check `CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME` to only add options or defaults that make sense if this is a project.

Also, since namespaces are a good idea, and the usage of your library should be consistent with the other methods below, you should add

```
add_library(MyLib::MyLib ALIAS MyLib)
```

to standardise the usage across all methods. This ALIAS target will not be exported below.

## Exporting

The third way is `*Config.cmake` scripts; that will be the topic of the next chapter in this session.

# Installing

Install commands cause a file or target to be "installed" into the install tree when you `make install`. Your basic target install command looks like this:

```
install(TARGETS MyLib
        EXPORT MyLibTargets
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib
        RUNTIME DESTINATION bin
        INCLUDES DESTINATION include
        )
```

The various destinations are only needed if you have a library, static library, or program to install. The includes destination is special; since a target does not install includes. It only sets the includes destination on the exported target (which is often already set by `target_include_directories`, so check the MyLibTargets file and make sure you don't have the include directory included twice if you want clean cmake files).

It's usually a good idea to give CMake access to the version, so that `find_package` can have a version specified. That looks like this:

```
include(CMakePackageConfigHelpers)
write_basic_package_version_file(
    MyLibConfigVersion.cmake
    VERSION ${PACKAGE_VERSION}
    COMPATIBILITY AnyNewerVersion
    )
```

You have two choices next. You need to make a `MyLibConfig.cmake`, but you can do it either by exporting your targets directly to it, or by writing it by hand, then including the targets file. The later option is what you'll need if you have any dependencies, even just OpenMP, so I'll illustrate that method.

First, make an install targets file (very similar to the one you made in the build directory):

```
install(EXPORT MyLibTargets
        FILE MyLibTargets.cmake
        NAMESPACE MyLib::
        DESTINATION lib/cmake/MyLib
         )
```

This file will take the targets you exported and put them in a file. If you have no dependencies, just use `MyLibConfig.cmake` instead of `MyLibTargets.cmake` here. Then write a custom `MyLibConfig.cmake` file in your source tree somewhere. If you want to capture configure time variables, you can use a `.in` file, and you will want to use the `@var@` syntax. The contents that look like this:

```
include(CMakeFindDependencyMacro)

# Capturing values from configure (optional)
set(my-config-var @my-config-var@)

# Same syntax as find_package
find_dependency(MYDEP REQUIRED)

# Any extra setup

# Add the targets file
include("${CMAKE_CURRENT_LIST_DIR}/MyLibTargets.cmake")
```

Now, you can use configure file (if you used a `.in` file) and then install the resulting file. Since we've made a `ConfigVersion` file, this is a good place to install it too.

```
configure_file(MyLibConfig.cmake.in MyLibConfig.cmake @ONLY)
install(FILES "${CMAKE_CURRENT_BINARY_DIR}/MyLibConfig.cmake"
              "${CMAKE_CURRENT_BINARY_DIR}/MyLibConfigVersion.cmake"
        DESTINATION lib/cmake/MyLib
        )
```

That's it! Now once you install a package, there will be files in `lib/cmake/MyLib` that CMake will search for (specifically, `MyLibConfig.cmake` and `MyLibConfigVersion.cmake`), and the targets file that config uses should be there as well.

When CMake searches for a package, it will look in the current install prefix and several standard places. You can also add this to your search path manually, including `MyLib_PATH`, and CMake gives the user nice help output if the configure file is not found.

# Exporting

The default behavior for exporting changed in CMake 3.15. Since changing files in a user's home directory is considered "surprising" (and it is, which is why this chapter exists), it is no longer the default behavior. If you set a minimum or maximum CMake version of 3.15 or better, this will no longer happen unless you set `CMAKE_EXPORT_PACKAGE_REGISTRY`, as mentioned below.

There are three ways to access a project from another project: subdirectory, exported build directories, and installing. To use the build directory of one project in another project, you will need to export targets. Exporting targets is needed for a proper install, allowing the build directory to be used as well is just two added lines. It is not generally a way to work that I would recommend, but can be useful for development and as way to prepare the installation procedure discussed later.

You should make an export set, probably near the end of your main `CMakeLists.txt`:

```
export(TARGETS MyLib1 MyLib2 NAMESPACE MyLib:: FILE MyLibTargets.cmake)
```

This puts the targets you have listed into a file in the build directory, and optionally prefixes them with a namespace. Now, to allow CMake to find this package, export the package into the `$HOME/.cmake/packages` folder:

```
set(CMAKE_EXPORT_PACKAGE_REGISTRY ON)
export(PACKAGE MyLib)
```

Now, if you `find_package(MyLib)`, CMake can find the build folder. Look at the generated `MyLibTargets.cmake` file to help you understand exactly what is created; it's just a normal CMake file, with the exported targets.

Note that there's a downside: if you have imported dependencies, they will need to be imported before you `find_package`. That will be fixed in the next method.

# Packaging

There are two ways to instruct CMake to build your package; one is to use a CPackConfig.cmake file, and the other is to integrate the CPack variables into your CMakeLists.txt file. Since you want variables from your main build to be included, like version number, you'll want to make a configure file if you go that route. I'll show you the integrated version:

```
# Packaging support
set(CPACK_PACKAGE_VENDOR "Vendor name")
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY "Some summary")
set(CPACK_PACKAGE_VERSION_MAJOR ${PROJECT_VERSION_MAJOR})
set(CPACK_PACKAGE_VERSION_MINOR ${PROJECT_VERSION_MINOR})
set(CPACK_PACKAGE_VERSION_PATCH ${PROJECT_VERSION_PATCH})
set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/LICENCE")
set(CPACK_RESOURCE_FILE_README "${CMAKE_CURRENT_SOURCE_DIR}/README.md")
```

These are the most common variables you'll need to make a binary package. A binary package uses the install mechanism of CMake, so anything that is installed will be present.

You can also make a source package. You should set `CMAKE_SOUCE_IGNORE_FILES` to regular expressions that ensure you don't pick up any extra files (like the build directory or git details); otherwise `make package_source` will bundle up literally everything in the source directory. You can also set the source generator to make your favorite types of files for source packages:

```
set(CPACK_SOURCE_GENERATOR "TGZ;ZIP")
set(CPACK_SOURCE_IGNORE_FILES
    /.git
    /dist
    /.*build.*
    /\\\\.DS_Store
)
```

Note that this will not work on Windows, but the generated source packages work on Windows.

Finally, you need to include the CPack module:

```
include(CPack)
```

# Finding Package

There are two ways to find packages in CMake.

# CUDA (in progress)

CUDA support is available in two flavors. The new method, introduced in CMake 3.8 (3.9 for Windows), will be what I focus on first. The old method will be covered afterwards, but as you'll see, it's uglier and harder to get right. I'd stick to requiring CMake 3.8 or 3.9 for CUDA (and CMake 3.11 for IDEs like Xcode and Visual Studio).

A good resource for CUDA and Modern CMake is this talk by CMake developer Robert Maynard at GTC 2017.

# Method 1: CUDA as a First Class Language

This method is quite new, and doesn't seem to have much documentation. There are several issues you need to watch out for when using it, but overall is should be a much nicer and cleaner way to use CUDA.

## Adding the CUDA Language

There are two ways to enable CUDA support. If CUDA is not optional:

```
project(MY_PROJECT LANGUAGES CUDA CXX)
```

You'll probably want `CXX` listed here also. And, if CUDA is optional, you'll want to put this in somewhere conditionally:

```
enable_language(CUDA)
```

To check to see if CUDA is available, use CheckLanuage:

```
include(CheckLanguage)
check_language(CUDA)
```

You can check the version of the NVCC toolkit with `CMAKE_CUDA_COMPILER_VERSION` (for now, only NVCC is supported, but just to be sure, check `CMAKE_CUDA_COMPILER_ID STREQUAL "NVIDIA"`).

## Variables for CUDA

Many variables with `CXX` in the name have a CUDA version with `CUDA` instead. For example, to set the C++ standard required for CUDA,

```
if(NOT DEFINED CMAKE_CUDA_STANDARD)
    set(CMAKE_CUDA_STANDARD 11)
    set(CMAKE_CUDA_STANDARD_REQUIRED ON)
endif()
```

## Adding a library

This is the easy part; as long as you use `.cu` for CUDA files, you can just add libraries like you normally would.

You can also use separable compilation:

```
set_target_properties(mylib PROPERTIES
                            CUDA_SEPERABLE_COMPILATION ON)
```

You can also direclty make a PTX file with the `CUDA_PTX_COMPILATION` property.

### Working with targets

Using targets should work similarly to CXX, but there's a problem. If you include a target that includes compiler options (flags), most of the time, the options will not be protected by the correct includes (and the chances of them having the correct CUDA wrapper is even smaller). Here's what a correct compiler options line should look like:

```
"$<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CXX>>:-fopenmp>$<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CUDA>>:-Xcompiler=-fopenmp>"
```

However, if you using almost any find_package, and using the Modern CMake methods of targets and inheritance, everything will break. I've learned that the hard way.

For now, here's a pretty reasonable solution, *as long as you know the un-aliased target name*. It's a function that will fix a C++ only target by wrapping the flags if using a CUDA compiler:

```
function(CUDA_CONVERT_FLAGS EXISTING_TARGET)
    get_property(old_flags TARGET ${EXISTING_TARGET} PROPERTY INTERFACE_COMPILE_OPTIONS)
    if(NOT "${old_flags}" STREQUAL "")
        string(REPLACE ";" "," CUDA_flags "${old_flags}")
        set_property(TARGET ${EXISTING_TARGET} PROPERTY INTERFACE_COMPILE_OPTIONS
            "$<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CXX>>:${old_flags}>$<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CUDA>>:-Xcompile
r=${CUDA_flags}>"
            )
    endif()
endfunction()
```

### Useful variables

- `CMAKE_CUDA_TOOLKIT_INCLUDE_DIRECTORIES` : Place for built-in Thrust, etc
- `CMAKE_CUDA_COMPILER` : NVCC with location

> **Note that FindCUDA is deprecated, but for now, the following functions require FindCUDA:**
>
> - CUDA version checks / picking a version
> - Architecture detection (Note: 3.12 fixes this partially)
> - Linking to CUDA libraries from non-.cu files

## Method 2: FindCUDA

If you want to support an older version of CMake, I recommend at least including the FindCUDA from CMake version 3.9 in your cmake folder (see the CLIUtils github organization for a git repository). You'll want two features that were added: `CUDA_LINK_LIBRARIES_KEYWORD` and `cuda_select_nvcc_arch_flags`, along with the newer architectures and CUDA versions.

To use the old CUDA support, you use `find_package`:

```
find_package(CUDA 7.0 REQUIRED)
message(STATUS "Found CUDA ${CUDA_VERSION_STRING} at ${CUDA_TOOLKIT_ROOT_DIR}")
```

You can control the CUDA flags with `CUDA_NVCC_FLAGS` (list append) and you can control separable compilation with `CUDA_SEPARABLE_COMPILATION`. You'll also want to make sure CUDA plays nice and adds keywords to the targets (CMake 3.9+):

```
set(CUDA_LINK_LIBRARIES_KEYWORD PUBLIC)
```

You'll also might want to allow a user to check for the arch flags of their current hardware:

```
cuda_select_nvcc_arch_flags(ARCH_FLAGS) # optional argument for arch to add
```

# OpenMP

OpenMP support was drastically improved in CMake 3.9+. The Modern(TM) way to add OpenMP to a target is:

```
find_package(OpenMP)
if(OpenMP_CXX_FOUND)
    target_link_libraries(MyTarget PUBLIC OpenMP::OpenMP_CXX)
endif()
```

This not only is cleaner than the old method, it will also correctly set the library link line differently from the compile line if needed. In CMake 3.12+, this will even support OpenMP on macOS (if the library is available, such as with `brew install libomp`). However, if you need to support older CMake, the following works on CMake 3.1+:

```
# For CMake < 3.9, we need to make the target ourselves
if(NOT TARGET OpenMP::OpenMP_CXX)
    find_package(Threads REQUIRED)
    add_library(OpenMP::OpenMP_CXX IMPORTED INTERFACE)
    set_property(TARGET OpenMP::OpenMP_CXX
                 PROPERTY INTERFACE_COMPILE_OPTIONS ${OpenMP_CXX_FLAGS})
    # Only works if the same flag is passed to the linker; use CMake 3.9+ otherwise (Intel, AppleClang)
    set_property(TARGET OpenMP::OpenMP_CXX
                 PROPERTY INTERFACE_LINK_LIBRARIES ${OpenMP_CXX_FLAGS} Threads::Threads)

endif()
target_link_libraries(MyTarget PUBLIC OpenMP::OpenMP_CXX)
```

Warning: CMake < 3.4 has a bug in the Threads package that requires you to have the `C` language enabled.

# Boost library

The Boost library is included in the find packages that CMake provides, but it has a couple of oddities in how it works. See FindBoost for a full description; this will just give a quick overview and provide a recipe. Be sure to check the page for the minimum required version of CMake you are using and see what options you have.

First, you can customize the behavior of the Boost libraries selected using a set of variables that you set before searching for Boost. There are a growing number of settings, but here are the three most common ones:

```
set(Boost_USE_STATIC_LIBS OFF)
set(Boost_USE_MULTITHREADED ON)
set(Boost_USE_STATIC_RUNTIME OFF)
```

In CMake 3.5, imported targets were added. These targets handle dependencies for you as well, so they are a very nice way to add Boost libraries. However, CMake has the dependency information baked into it for all known versions of Boost, so CMake must be newer than Boost for these to work. In a recent merge request, CMake started assuming that the dependencies hold from the last version it knows about, and will use that (along with giving a warning). This functionality was backported into CMake 3.9.

The import targets are in the `Boost::` namespace. `Boost::boost` is the header only part. The other compiled libraries are available, and include dependencies as needed.

Here is an example for using the `Boost::filesystem` library:

```
set(Boost_USE_STATIC_LIBS OFF)
set(Boost_USE_MULTITHREADED ON)
set(Boost_USE_STATIC_RUNTIME OFF)
find_package(Boost 1.50 REQUIRED COMPONENTS filesystem)
message(STATUS "Boost version: ${Boost_VERSION}")

# This is needed if your Boost version is newer than your CMake version
# or if you have an old version of CMake (<3.5)
if(NOT TARGET Boost::filesystem)
    add_library(Boost::filesystem IMPORTED INTERFACE)
    set_property(TARGET Boost::filesystem PROPERTY
        INTERFACE_INCLUDE_DIRECTORIES ${Boost_INCLUDE_DIR})
    set_property(TARGET Boost::filesystem PROPERTY
        INTERFACE_LINK_LIBRARIES ${Boost_LIBRARIES})
endif()

target_link_libraries(MyExeOrLibrary PUBLIC Boost::filesystem)
```

# MPI

To add MPI, like OpenMP, you'll be best off with CMake 3.9+.

```
find_package(MPI REQUIRED)
message(STATUS "Run: ${MPIEXEC} ${MPIEXEC_NUMPROC_FLAG} ${MPIEXEC_MAX_NUMPROCS} ${MPIEXEC_PREFLAGS} EXECUTABLE ${MPIEXEC_POSTF
LAGS} ARGS")
target_link_libraries(MyTarget PUBLIC MPI::MPI_CXX)
```

However, you can imitate this on CMake 3.1+ with:

```
find_package(MPI REQUIRED)

# For supporting CMake < 3.9:

if(NOT TARGET MPI::MPI_CXX)
    add_library(MPI::MPI_CXX IMPORTED INTERFACE)

    set_property(TARGET MPI::MPI_CXX
                 PROPERTY INTERFACE_COMPILE_OPTIONS ${MPI_CXX_COMPILE_FLAGS})
    set_property(TARGET MPI::MPI_CXX
                 PROPERTY INTERFACE_INCLUDE_DIRECTORIES "${MPI_CXX_INCLUDE_PATH}")
    set_property(TARGET MPI::MPI_CXX
                 PROPERTY INTERFACE_LINK_LIBRARIES ${MPI_CXX_LINK_FLAGS} ${MPI_CXX_LIBRARIES})
endif()

message(STATUS "Run: ${MPIEXEC} ${MPIEXEC_NUMPROC_FLAG} ${MPIEXEC_MAX_NUMPROCS} ${MPIEXEC_PREFLAGS} EXECUTABLE ${MPIEXEC_POSTF
LAGS} ARGS")
target_link_libraries(MyTarget PUBLIC MPI::MPI_CXX)
```

# ROOT

ROOT is a C++ Toolkit for High Energy Physics. It is huge. There are really a lot of ways to use it in CMake, though many/most of the examples you'll find are probably wrong. Here's my recommendation.

Most importantly, there are *lots of improvements* in CMake support in more recent versions of ROOT - Using 6.16+ is much, much easier! If you really must support 6.14 or earlier, see the section at the end.

## Finding ROOT

ROOT 6.10+ supports config file discovery, so you can just do:

```
find_package(ROOT 6.16 CONFIG REQUIRED)
```

to attempt to find ROOT. If you don't have your paths set up, you can pass `-DROOT_DIR=$ROOTSYS/cmake` to find ROOT. (But, really, you should source `thisroot.sh`).

## The right way (Targets)

ROOT 6.12 and earlier do not add the include directory for imported targets. ROOT 6.14+ has corrected this error, and required target properties have been getting better. This method is rapidly becoming easier to use (see the example at the end of this page for the older ROOT details).

To link, just pick the libraries you want to use:

```
add_executable(RootSimpleExample SimpleExample.cxx)
target_link_libraries(RootSimpleExample PUBLIC ROOT::Physics)
```

If you'd like to see the default list, run `root-config --libs` on the command line. In Homebrew ROOT 6.18 this would be:

- `ROOT::Core`
- `ROOT::Gpad`
- `ROOT::Graf3d`
- `ROOT::Graf`
- `ROOT::Hist`
- `ROOT::Imt`
- `ROOT::MathCore`
- `ROOT::Matrix`
- `ROOT::MultiProc`
- `ROOT::Net`
- `ROOT::Physics`
- `ROOT::Postscript`
- `ROOT::RIO`
- `ROOT::ROOTDataFrame`
- `ROOT::ROOTVecOps`
- `ROOT::Rint`
- `ROOT::Thread`
- `ROOT::TreePlayer`
- `ROOT::Tree`

# The old global way

ROOT provides a utility to set up a ROOT project, which you can activate using `include("${ROOT_USE_FILE}")`. This will automatically make ugly directory level and global variables for you. It will save you a little time setting up, and will waste massive amounts of time later if you try to do anything tricky. As long as you aren't making a library, it's probably fine for simple scripts. Includes and flags are set globally, but you'll still need to link to `${ROOT_LIBRARIES}` yourself, along with possibly `ROOT_EXE_LINKER_FLAGS` (You will have to `separate_arguments` first before linking or you will get an error if there are multiple flags, like on macOS). Also, before 6.16, you have to manually fix a bug in the spacing.

Here's what it would look like:

```
# Sets up global settings
include("${ROOT_USE_FILE}")

# This is required for ROOT < 6.16
# string(REPLACE "-L " "-L" ROOT_EXE_LINKER_FLAGS "${ROOT_EXE_LINKER_FLAGS}")

# This is required on if there is more than one flag (like on macOS)
separate_arguments(ROOT_EXE_LINKER_FLAGS)

add_executable(RootUseFileExample SimpleExample.cxx)
target_link_libraries(RootUseFileExample PUBLIC ${ROOT_LIBRARIES} ${ROOT_EXE_LINKER_FLAGS})
```

# Components

Find ROOT allows you to specify components. It will add anything you list to `${ROOT_LIBRARIES}`, so you might want to build your own target using that to avoid listing the components twice. This did not solve dependencies; it was an error to list `RooFit` but not `RooFitCore`. If you link to `ROOT::RooFit` instead of `${ROOT_LIBRARIES}`, then `RooFitCore` is not required.

# Dictionary generation

Dictionary generation is ROOT's way of working around the missing reflection feature in C++. It allows ROOT to learn the details of your class so it can save it, show methods in the Cling interpreter, etc. You'll need three things in your source code to make it work for classes:

- Your class definition should end with `ClassDef(MyClassName, 1)`
- Your class implementation should have `ClassImp(MyClassName)` in it
- You should have a file with a name that ends with `LinkDef.h`

The `LinkDef.h` file follows a specific formula and tells ROOT what parts to generate dictionaries for.

To generate, you should include the following in your CMakeLists:

```
include("${ROOT_DIR}/modules/RootNewMacros.cmake")

# Uncomment for ROOT versions than 6.16
# They break if nothing is in the global include list!
# include_directories(ROOT_BUG)
```

The second line is due to a bug in the NewMacros file that causes dictionary generation to fail if there is not at least one global include directory or a `inc` folder. Here I'm including a non-existent directory just to make it work. There is no `ROOT_BUG` directory.

To generate a file:

```
root_generate_dictionary(G__Example Example.h LINKDEF ExampleLinkDef.h)
```

The final argument, listed after `LINKDEF`, must have a name that ends in `LinkDef.h`. This command will create three files. If you started output name with `G__`, that will be removed from the name, otherwise it will use the name given; this must match the final output library name you will soon be creating. Assuming this is `${NAME}`:

- `${NAME}.cxx`: This file should be included in your sources when you make the library.
- `lib{NAME}.rootmap` (`G__` prefix removed): The rootmap file in plain text
- `lib{NAME}_rdict.pcm` (`G__` prefix removed): A ROOT file

The final two output files must sit next to the library output. This is done by checking `CMAKE_LIBRARY_OUTPUT_DIRECTORY` (it will not pick up local target settings). If you have a libdir set but you don't have (global) install locations set, you'll also need to set `ARG_NOINSTALL` to `TRUE`.

---

# Using Old ROOT

If you really have to use older ROOT, you'll need something like this:

```
# ROOT targets are missing includes and flags in ROOT 6.10 and 6.12
set_property(TARGET ROOT::Core PROPERTY
    INTERFACE_INCLUDE_DIRECTORIES "${ROOT_INCLUDE_DIRS}")

# Early ROOT does not include the flags required on targets
add_library(ROOT::Flags_CXX IMPORTED INTERFACE)


# ROOT 6.14 and earlier have a spacing bug in the linker flags
string(REPLACE "-L " "-L" ROOT_EXE_LINKER_FLAGS "${ROOT_EXE_LINKER_FLAGS}")

# Fix for ROOT_CXX_FLAGS not actually being a CMake list
separate_arguments(ROOT_CXX_FLAGS)
set_property(TARGET ROOT::Flags_CXX APPEND PROPERTY
    INTERFACE_COMPILE_OPTIONS ${ROOT_CXX_FLAGS})

# Add definitions
separate_arguments(ROOT_DEFINITIONS)
foreach(_flag ${ROOT_EXE_LINKER_FLAG_LIST})
    # Remove -D or /D if present
    string(REGEX REPLACE [=[^[-//]D]=] "" _flag ${_flag})
    set_property(TARGET ROOT::Flags APPEND PROPERTY INTERFACE_LINK_LIBRARIES ${_flag})
endforeach()

# This also fixes a bug in the linker flags
separate_arguments(ROOT_EXE_LINKER_FLAGS)
set_property(TARGET ROOT::Flags_CXX APPEND PROPERTY
    INTERFACE_LINK_LIBRARIES ${ROOT_EXE_LINKER_FLAGS})

# Make sure you link with ROOT::Flags_CXX too!
```

# A Simple ROOT Project

This is a minimal example of a ROOT project using the UseFile system and without a dictionary.

## examples/root-usefile/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1...3.16)

project(RootUseFileExample LANGUAGES CXX)

find_package(ROOT 6.16 CONFIG REQUIRED)
# Sets up global settings
include("${ROOT_USE_FILE}")

# This is required for ROOT < 6.16
# string(REPLACE "-L " "-L" ROOT_EXE_LINKER_FLAGS "${ROOT_EXE_LINKER_FLAGS}")

# This is required on if there is more than one flag (like on macOS)
separate_arguments(ROOT_EXE_LINKER_FLAGS)

add_executable(RootUseFileExample SimpleExample.cxx)
target_link_libraries(RootUseFileExample PUBLIC ${ROOT_LIBRARIES} ${ROOT_EXE_LINKER_FLAGS})
```

## examples/root-usefile/SimpleExample.cxx

```
#include <TLorentzVector.h>

int main() {
    TLorentzVector v(1,2,3,4);
    v.Print();
    return 0;
}
```

# A Simple ROOT Project

This is a minimal example of a ROOT project using the target system and without a dictionary.

## examples/root-simple/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.1...3.16)

project(RootSimpleExample LANGUAGES CXX)

# Finding the ROOT package
find_package(ROOT 6.16 CONFIG REQUIRED)


# Adding an executable program and linking to needed ROOT libraries
add_executable(RootSimpleExample SimpleExample.cxx)
target_link_libraries(RootSimpleExample PUBLIC ROOT::Physics)
```

## examples/root-simple/SimpleExample.cxx

```
#include <TLorentzVector.h>

int main() {
    TLorentzVector v(1,2,3,4);
    v.Print();
    return 0;
}
```

# Dictionary Example

This is an example of building a module that includes a dictionary in CMake. Instead of using the ROOT suggested flags, we will manually add threading via `find_package`, which is the only important flag in the list on most systems.

### examples/root-dict/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.4...3.16)

project(RootDictExample LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_PLATFORM_INDEPENDENT_CODE ON)

find_package(ROOT CONFIG REQUIRED)
include("${ROOT_DIR}/modules/RootNewMacros.cmake")

root_generate_dictionary(G__DictExample DictExample.h LINKDEF DictLinkDef.h)

add_library(DictExample SHARED DictExample.cxx DictExample.h G__DictExample.cxx)
target_include_directories(DictExample PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}")
target_link_libraries(DictExample PUBLIC ROOT::Core)
```

# Supporting files

This is just a simple-as-possible class definition, with one method:

### examples/root-dict/DictExample.cxx

```
#include "DictExample.h"

Double_t Simple::GetX() const {return x;}

ClassImp(Simple)
```

### examples/root-dict/DictExample.h

```
#pragma once

#include <TROOT.h>

class Simple {
    Double_t x;

public:
    Simple() : x(2.5) {}
    Double_t GetX() const;

    ClassDef(Simple,1)
};
```

We need a `LinkDef.h`, as well.

### examples/root-dict/DictLinkDef.h

```
// See: https://root.cern.ch/selecting-dictionary-entries-linkdefh
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;
#pragma link C++ nestedclasses;

#pragma link C++ class Simple+;

#endif
```

# Testing it

This is an example of a macro that tests the correct generation from the files listed above.

## examples/root-dict/CheckLoad.C

```
{
gSystem->Load("libDictExample");
Simple s;
cout << s.GetX() << endl;
TFile *_file = TFile::Open("tmp.root", "RECREATE");
gDirectory->WriteObject(&s, "MyS");
Simple *MyS = nullptr;
gDirectory->GetObject("MyS", MyS);
cout << MyS->GetX() << endl;
_file->Close();
}
```

# Minuit2

Minuit2 is available in standalone mode, for use in cases where ROOT is either not available or not built with Minuit2 enabled. This will cover recommended usages, as well as some aspects of the design.

# Usage

Minuit2 can be used in any of the standard CMake ways, either from the ROOT source or from a standalone source distribution:

```
# Check for Minuit2 in ROOT if you want
# and then link to ROOT::Minuit2 instead

add_subdirectory(minuit2) # or root/math/minuit2
# OR
find_package(Minuit2 CONFIG) # Either build or install

target_link_libraries(MyProgram PRIVATE Minuit2::Minuit2)
```

# Development

Minuit2 is a good example of potential solutions to the problem of integrating a modern (CMake 3.1+) build into an existing framework.

To handle the two different CMake systems, the main `CMakeLists.txt` defines common options, then calls a `Standalone.cmake` file if this is not building as part of ROOT.

The hardest part in the ROOT case is that Minuit2 requires files that are outside the `math/minuit2` directory. This was solved by adding a `copy_standalone.cmake` file with a function that takes a filename list and then either returns a list of filenames inplace in the original source, or copies files into the local source and returns a list of the new locations, or returns just the list of new locations if the original source does not exist (standalone).

```
# Copies files into source directory
cmake /root/math/minuit2 -Dminuit2-standalone=ON

# Makes .tar.gz from source directory
make package_source

# Optional, clean the source directory
make purge
```

This is only intended for developers wanting to produce source packages - a normal user *does not pass this option* and will not create source copies.

You can use `make install` or `make package` (binary packages) without adding this `standalone` option, either from inside the ROOT source or from a standalone package.