Modern CMake

Henry Schreiner

Contents

1. Getting Started

1.1 Installing CMake

> **Tip**
>
> Your CMake version should be newer than your compiler. It should be newer than the libraries you are using. 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 3.15+ support. Maybe even if they want 4.4+ support…

1.1.1 Quick list (more info on each method below)

Ordered by author preference:

- All
  - Pip(x) (official, often updates same-day)
  - Anaconda / Conda-Forge
- Windows
  - Winget
  - Chocolatey
  - Scoop
  - MSYS2
  - Download binary (official)
- MacOS
  - Homebrew
  - MacPorts
  - Download binary (official)
- Linux
  - Snapcraft (official)
  - APT repository (Ubuntu/Debian only) (official)
  - Download binary (official)

1.1.2 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 (and a Universal2 version is supplied supporting both Intel and Apple Silicon), but using `brew install cmake` is much nicer if you use Homebrew (and you should; Apple even supports Homebrew such as during the Apple Silicon rollout). You can also get it on most other package managers, such as Chocolatey for Windows or MacPorts for macOS.

On Linux, there are several options. Kitware provides a Debian/Ubuntu 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/v4.4/cmake-4.4.0-linux-x86_64.tar.gz" |
tar --strip-components=1 -xz -C ~/.local
```

[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`!

The names changed in 3.20; older releases had names like `cmake-3.19.7-Linux-x86_64.tar.gz`. If you just want a local folder with CMake only:

```
~ $ mkdir -p cmake-4.4 && wget -qO- "https://cmake.org/files/v4.4/cmake-4.4.0-linux-x86_64.tar.gz" | tar --strip-components=1 -xz -C cmake-4.4
~ $ export PATH=`pwd`/cmake-4.4/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/v4.4/cmake-4.4.0-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.

1.1.3 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.

1.1.3.1 Windows

The winget package is a good way to get CMake. Other options:

Also Scoop is generally up to date. The normal installers from CMake.org are common on Windows, too.

1.1.3.2 MacOS

Homebrew is quite a bit more popular nowadays on macOS, at least according to Google Trends.

1.1.3.3 Linux

1.1.3.3.1 RHEL/Rocky/Alma/CentOS Stream

RHEL 9 and 10 (and the Rocky Linux / AlmaLinux rebuilds and CentOS Stream) ship a maintained CMake in AppStream and update it over the release's lifetime, so the system package is usable; you no longer need EPEL just to get a reasonable CMake (EPEL doesn't even carry one anymore).

1.1.3.3.2 Ubuntu

1.1.3.3.3 Debian

1.1.3.3.4 Other

1.1.3.4 General tools

Just `pip install cmake` on many systems. Add `--user` if you have to (modern pip does this for you if needed). Universal2 wheels are supplied for macOS, covering both Intel and Apple Silicon.

1.1.3.5 CI

Distribution	CMake version	Notes
Azure DevOps	3.31	kept up to date
GitHub Actions ubuntu-22.04	3.31	Same runners as Azure DevOps

Distribution	CMake version	Notes
GitHub Actions ubuntu-24.04	3.31	`ubuntu-latest`; same runners as Azure DevOps

If you are using GitHub Actions, also see the jwlawson/actions-setup-cmake action, which can install your selection of CMake, even in a docker action run.

1.1.3.6 Full list

Versions less than 3.15 are marked by a deeper color of red.

Also see pkgs.org/download/cmake.

1.1.4 Pip

This is also provided as an official package, maintained by the authors of CMake at KitWare and several PyPA members, including myself. It's now supported on special architectures, like PowerPC on Linux and Apple Silicon on macOS, and on MUSL systems like Alpine too. If you have pip (Python's package installer), you can do:

```
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 build from source (the package is built with `scikit-build-core`), which requires an existing copy of CMake. 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. It really shines when placed in a `pyproject.toml` file, however - it will only be installed to build your package, and will not remain afterwards! Fantastic.

This also, of course, works with pipx. So you can even use `pipx run cmake` to run CMake in a disposable virtual environment, without any setup - and this works out-of-the-box on GitHub Actions, since `pipx` is a supported package manager there!

> **Tip**
>
> 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.

1.2 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.

1.2.1 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. -Bbuild
~/package $ cmake --build build
```

(`-S` is the source location containing `CMakeLists.txt`, and `-B` is the build directory, which will be created.) 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 $ make -C build install
~/package $ cmake --build build --target install
~/package $ cmake --install build # CMake 3.15+ only
```

So which 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 <3.20) 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`. You can instead use the environment variables for these, as well, such as `CMAKE_BUILD_PARALLEL_LEVEL` (CMake 3.12+) and `VERBOSE` (CMake 3.14+).

1.2.2 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. If you are using Git Bash, you should use `export` to propagate the variables to subprocesses:

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

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.

1.2.3 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 explicitly 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.

1.2.4 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 .`).

1.2.5 Verbose and partial builds

Although not all build tools support it, you can get verbose builds (pick one):

```
~/package $ cmake --build build --verbose # CMake 3.14+ only
~/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.

1.2.6 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.

1.2.6.1 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.

1.2.7 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!

1.3 Do's and Don'ts

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.

1.3.1 CMake Antipatterns

- **Do not use global functions**: This includes `link_directories`, `include_directories`, 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.
- **Don't 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.

1.3.2 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 use 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.

1.3.3 Selecting a minimum in 2026:

What minimum CMake should you *run* locally, and what minimum should you *support* for people using your code? Since you are reading this, you should be able to get a release in the last few versions of CMake; do that, it will make your development easier. For support, there are two ways to pick minimums: based on features added (which is what a developer cares about), or on common pre-installed CMakes (which is what a user cares about).

Never select a minimum version older than the oldest compiler you support. CMake should always be at least as new as the CMake that supported your oldest compiler.

1.3.3.1 What minimum to choose - OS support:

- 3.19: First to support Apple Silicon.
- 3.22: Ubuntu 22.04, Amazon Linux 2023.
- 3.25: Debian 12 (11 backports)
- 3.26: Rocky Linux 9, AlmaLinux 9 (3.31 in current point releases)
- 3.28: Ubuntu 24.04.
- 3.30: Rocky Linux 10, AlmaLinux 10, CentOS Stream 10 (3.31 in current updates)
- 3.31: Debian 13 (12 backports)
- latest: pip/conda-forge/Homebrew/Chocolatey, etc.

1.3.3.2 What minimum to choose - Features:

- 3.11: `IMPORTED INTERFACE` setting, faster, FetchContent, `COMPILE_LANGUAGE` in IDEs
- 3.12: C++20, `cmake --build build -j N`, `SHELL:`, FindPython
- 3.14/3.15: CLI, FindPython updates
- 3.16: Unity builds / precompiled headers, CUDA meta features
- 3.17/3.18: Lots more CUDA, metaprogramming, FindPython updates

- 3.19: Presets
- 3.20: C++23, `cmake_path`
- 3.24: Package finder
- 3.25: Blocks for scoping
- 3.28: C++20 modules
- 3.29: Build before test target
- 3.30: Full C++26 support

1.4 What's new in CMake

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

1.4.1 CMake in development: WIP

CMake 4.5 is in development. Nothing major to report yet.

1.4.2 CMake 4.4: Diagnostics

This version introduces a structured diagnostics system that replaces the old warning variables and flags, a `SOURCES` file set type, test discovery at test time, and a batch of scripting conveniences.

- Initially released July 14, 2026
- New `cmake_diagnostic()` command gives per-category, structured control over warnings (the `CMAKE_WARN_DEPRECATED`/`CMAKE_ERROR_DEPRECATED` variables and the `-W[no-][error=]dev` flags are now deprecated in favor of `author`)
- New `SOURCES` file set type for specifying a target's sources
- New `discover_tests()` command registers tests discovered at test time; Ninja gained `test_prep/<test-name>` targets, and ctest can forward extra arguments to tests after `--`
- `cmake_parse_arguments()` gained a `PARSE_ARGN` signature
- `list(FILTER)`, `list(SORT)`, and `list(TRANSFORM)` accept callback predicates/comparators
- `execute_process()` gained `ENVIRONMENT` and `ENVIRONMENT_MODIFICATION`
- `cmake --install` supports multiple components and parallel jobs
- `--presets-file` added to `cmake`, `ctest`, and `cpack` (also presets schema v12)
- `write_basic_package_version_file()` gained `SamePatchVersion`, `SameFullVersion`, and `SemanticVersion` compatibility modes
- CUDA 23 support (NVCC 13.3+); `clang-cl` gained C++20 named modules
- FindPython supports the Stable ABI for free-threaded Python
- `macro()` now preserves backslashes in arguments (CMP0219)

1.4.3 CMake 4.3: Package specifications

This version adds support for the Common Package Specification (CPS), a language-agnostic package description format that packages can now produce and consume instead of hand-written CMake config files. It also adds an instrumentation framework for profiling your configure and build steps.

- Initially released March 17, 2026
- `find_package` can now search for and import CPS packages
- `install` and `export` gained a `PACKAGE_INFO` mode to generate CPS package descriptions
- `project` gained `COMPAT_VERSION` and `SPDX_LICENSE` options
- New cmake-instrumentation facility collects timing and diagnostic data during configure/generate/build/test/install, viewable in CDash or as Google Trace Event files
- `cmake --build` now accepts a build directory and `--preset` together
- `cmake --version` can print JSON, and `cmake -E bin2c` was added
- Archive handling (`cmake -E tar`, `file(ARCHIVE_CREATE)`, CPack) gained multithreading, compression levels, and more compression methods
- `get_property`/`set_property` support a `FILE_SET` scope
- New string comparison and `$<STRING:...>` generator expressions, plus file set and source file queries
- FindRuby and FindSQLite3 now provide imported targets

1.4.4 CMake 4.2: Emscripten

This version adds native Emscripten cross-compile support. There are lots of other smallish improvements too, like matching-case version variables from `find_package`, controls for object files, direct cache manipulation, and more.

- Initially released Nov 19, 2025
- File API updates (version 2.9), with more info on imported/interface targets
- Visual Studio 18 2026 added (14 2015 deprecated)
- Cross-compile support for Emscripten (WebAssembly)
- cmake -E copy_if_newer (and directory version) added
- set(CACHE{<variable>} (and unset) were added to directly manipulate the cache
- string(REGEX QUOTE) added (like re.escape in Python)
- New properties include OBJECT_NAME and INSTALL_OBJECT_* to control generated object files
- Matching case <PackageName>_VERSION variables added to all find modules
- NumPy no longer depends on Development.Module (CMP0201)
- ExternalProject can set environment variables

1.4.5 CMake 4.1: Bits of polish

This version improves the pkg-config integration, OPTIONAL for find commands, and other small changes. This release seemed to be focused on polishing up some edge cases.

- Initially released August 5, 2025
- cmake_pkg_config now supports IMPORT and POPULATE.
- Some configure log improvements related to find commands.
- CMAKE_FIND_REQUIRED added, along with new OPTIONAL keywords for all find commands.
- CMAKE_<LANG>_COMPILER_ARCHITECTURE_ID now public, and filled in for most compilers.
- NVIDIA support added to FindBLAS and FindLAPACK
- FindPython modules now enforce consistency when cross-compiling (CMP0190)
- Some GNUInstallDirs changes gated by CMP0192 and CMP0193

1.4.6 CMake 4.0: Out with the old

This version removes support for setting the CMake policy below 3.5. This means cmake_minimum_required(VERSION 3.4) is now an error, but cmake_minimum_required(VERSION 3.4...3.15), for example, is fine. Setting this below 3.10 has been a warning for quite some time now. You can use CMAKE_POLICY_VERSION_MINIMUM to work around third-party libraries that haven't updated.

- Initially released March 27, 2025
- A new CMAKE_POLICY_VERSION_MINIMUM variable to build old projects (support for <3.5 removed)
- Added a flag (--project-file) to control the CMakeLists.txt filename (for development only)
- Some new linker features, like LINKER: prefix and CMAKE_LINK_WARNING_AS_ERROR
- You can set the debugger working directory for targets
- FindPython can be run multiple times in a directory using new Python_ARTIFACTS_PREFIX
- Generator expressions can convert to NATIVE_PATH
- Visual Studio generators 2015/2017 work like the later ones now with platform selection
- macOS SDK selection simplified and less automatic

1.4.7 CMake 3.31: Better workflow presets

This release adds a collection of small features and additions, along with some experimental new technology. Some updates to presets, including a new shorter command for workflow presets.

- Initially released November 6, 2024
- LFortran now supported
- $comment supported in preset files
- cmake -LR <regex> to search the cache

- `cmake --workflow <name>` now supported (one less argument)
- `codegen` target and `CODEGEN` in `add_custom_command`
- Added `cmake_pkg_config` command to extract values from pkg-config files.
- `CMAKE_HOST_EXECUTABLE_SUFFIX` was added
- Better CUDA support, including getting the host compiler and OpenMP
- Support for CMake 3.10 and older deprecated
- Lots of smaller tweaks and cleanup
- Experimental CPS (common package specification) support

1.4.8 CMake 3.30: C++26

This release adds C++26 support and a way to get the latest supported standard for a compiler. This release makes a lot of small changes not listed below, like better TLS support, some generator expression updates, and some schema updates.

- Initially released July 2, 2024
- A `$<QUOTE>` generator expression was added to produce `"`
- C++26 compile features support fully implemented (partial since 3.25)
- `CMAKE_<LANG>_STANDARD_LATEST` holds the latest standard the current compiler supports
- Free-threaded Python 3.13+ support (3.30.3+ best)
- Better support (new variables and targets) for Windows Python debug builds, and `DEBUG_POSTFIX` now added by `python_add_library`
- FindBoost removed
- Visual Studio 2008 support removed

1.4.9 CMake 3.29: Build before testing

Finally you can make the `test` target depend on `ALL`, meaning `cmake --build build -t test` will rebuild as needed! You have to opt-into this, though, by setting `CMAKE_SKIP_TEST_ALL_DEPENDENCY` to false. Several improvements were made for scripting, linker selection, and support was improved for various compiler combinations on Windows.

- Initially released March 21, 2024
- Linker selection option (`CMAKE_LINKER_TYPE`/`LINKER_TYPE`)
- `CMAKE_INSTALL_PREFIX` can now be initialized by a matching environment variable
- If commands to check file permissions
- Select launcher for tests (`CMAKE_TEST_LAUNCHER`/`TEST_LAUNCHER`)
- You can now make tests depend on `all` with `CMAKE_SKIP_TEST_ALL_DEPENDENCY` set to `FALSE`!
- `cmake_language(EXIT)` for scripts with exit codes
- Select Intel OneAPI Fortran compiler with Visual Studio
- Compile CUDA on Windows with Clang

1.4.10 CMake 3.28: C++20 modules

This release adds C++ module support. This does not include C++23's `import std`, but is exciting step forward for this landmark C++20 feature. These will be scanned by default if using C++20+ and a new enough compiler and valid generator and if you have a CMake minimum or maximum that includes CMake 3.28.

- Initially released December 6, 2023
- C++20 named modules supported by Ninja 1.11+ and MSVC 17.4+.
- `HIP` supported for NVIDIA.
- Apple's VisionOS added.
- `CMAKE_CROSSCOMPILING_EMULATOR` environment variable added.
- Get/set properties TEST supports other DIRECTORY's
- Some support for job servers added
- Support for passing variables to pkg-config
- Generator expressions now support short-circuting

1.4.11 CMake 3.27: Debugger

This release adds the new CMake debugger! This should improve support for debugging your CMake code in something like VSCode. This release also “removes” FindPythonLibs/FindPythonInterp/FindCUDA; if the min or max version is set to 3.27 or higher, the modules will be missing.

- Initially released July 19, 2023
- C++ Modules extensions (`.ccm`, `.cxxm`, `.c++m`) are treated as C++
- `COMPILE_ONLY`, `LIST`, and `PATH` generator expressions added, along with a few more specific ones.
- New `SKIP_LINTING`, as well as more generator expression support in things like <LANG>_CPPCHECK, etc.
- `find_package` now searches for uppercase <PACKAGENAME>_ROOT CMake/Environment variables.
- Added `add_custom_command(... DEPENDS_EXPLICIT_ONLY` & variable for Ninja dependency control.
- CMake build verbose now prints the working dir and command line used to build.
- Better support for versions of MSVC.
- Several new CUDA properties related to targeting `.cubin`/`.fatbin`/`.optixir`.
- Setting `cmake_minimum_required` less than 3.5 is now deprecated.
- FindCUDA semi-removed, use CUDA language and FindCUDAToolkit.
- FindPythonLibs & FindPythonInterp semi-removed, use FindPython.

1.4.12 CMake 3.26: Logging & Python

Two important additions for FindPython, PyPy SOABI support & LimitedAPI/StableABI support, really enhance FindPython’s use. There are quite a few nice fixes and new warnings, such as if you reverse the order of `project()` and `cmake_minimum_required()`. Logging has been moved from `CMakeOutput.log` and `CMakeError.log` to a new `CMakeConfigureLog.yaml` log.

- Initially released March 14, 2023
- FindPython generates the correct PyPy SOABI (finally!)
- FindPython supports LimitedAPI/StableABI with a new flag.
- CMake has a new YAML log of configure time checks in the output directory (also `message(CONFIGURE_LOG ...)`).
- `ASM_MARMASM` language added for Microsoft ARM assembler.
- `CMAKE_VS_VERSION_BUILD_NUMBER` added for the VS version number.
- `USE_FOLDERS` is on by default
- "<LANG>_CLANG_TIDY_EXPORT_FIXES_DIR" for clang-tidy suggested fix output.
- CMake’s copy CLI tool supports updates only if different
- `target_compile_options` now come after `target_compile_features` / CMAKE_<LANG>_STANDARD

1.4.13 CMake 3.25: Blocks and SYSTEM

CMake has new block scoping commands selectively controlling variables and policies. It also has a lot more control over SYSTEM. The functional features of CMake introduced a few releases ago are now usable in `find_` commands with `VALIDATOR`. Workflows got an upgrade, too.

- Initially released November 16, 2022
- C++26 support
- LTO for CUDA with nvcc
- Workflow presets added, package presets too.
- `SYSTEM` added to `add_subdirectory`, `FetchContent`, and as a directory property.
- `block()`/`endblock()` for policy/variable scopes, also `PROPAGATE` in `return()`
- `BSD` & `LINUX` variables added
- `VALIDATOR` function for `find_*` commands.
- Several improvements to `try_*` commands.

- `SYSTEM` target/directory property and `EXPORT_NO_SYSTEM` added, also for FetchContent.

1.4.14 CMake 3.24: Package Finder

This is a fantastic release. Package writers are getting integration between `find_package` and `FetchContent` that will allow "download if missing" workflows, and is configurable by packagers. Similarly, warnings as errors can be set by a package and removed by packagers, as well (still make sure not to do this unless you are being build as the main project!).

- Initially released August 4, 2022
- `--fresh` option removes the old cache when running.
- `find_package` and `FetchContent` now have integration - you have options to download missing dependencies.
- `find_package` has a new `GLOBAL` option.
- `CMAKE_PROJECT_TOP_LEVEL_INCLUDES` allows a user (like packagers) to inject pre-project code.
- `PATH` management for generator expressions.
- `CMAKE_COLOR_DIAGNOSTICS` env var & variable added, replacing `CMAKE_COLOR_MAKEFILE`.
- You can disable `find_*` searching the install prefix.
- `COMPILE_WARNING_AS_ERROR` property and `CMAKE_` variable, and `--compile-no-warning-as-error` to disable it.
- CUDA supports `native` to compile for the current GPUs detected.
- `SYSTEM` includes now are respected on MSVC generators.
- Better support for MSVC, XCode, and others.
- `LLVMFlang` compiler support.

1.4.15 CMake 3.23: Header only libraries

A solid release focused on header only libraries, more user control, CMake presets, and better CUDA support. There are some powerful new features for header only libraries, like the various `*_SETS` target properties. There are new controls like the ability to restrict paths for `find_` commands and the ability to remove `SYSTEM` from an existing target. You also get expanded debugging features, and the ability to force all links to be to targets. Presets can include other files. CUDA and C# received new updates, and a couple of compilers were added.

- Initially released March 29, 2022
- CMake presets are a bit nicer, with the ability to include other files.
- A couple of new supported compilers, and better C# support.
- `FILE_SET` for `install` and `target_sources` header-only source files.
- `<INTERFACE_>HEADER_SETS`, `<INTERFACE_>HEADER_DIRS` for target headers.
- `CUDA_ARCHITECTURES` support for all and all-major.
- DEBUG messages from can be enabled for `find_*` or find modules.
- `define_property()` has a handy `INITIALIZE_FROM_VARIABLE` option.
- `CMAKE_<SYSTEM_>IGNORE_PREFIX_PATH` to control `find_*` commands.
- `<CMAKE_>LINK_LIBRARIES_ONLY_TARGETS` added to force only targets linked (nice for finding mistakes!).
- `IMPORTED_NO_SYSTEM`, a new property to forcibly remove SYSTEM from a target.
- `FindGTest` now adds a `GMock` target if found.

1.4.16 CMake 3.22: Handy env vars

A smaller release with some nice improvements all around focused on supporting common build situations. You can finally set `CMAKE_BUILD_TYPE` in your environment to set a default build type. There are several other new env vars and variables too. Compiler flags related to standards have been improved. `cmake_host_system_information` got improved further (from 3.10) with OS information.

- Initially released November 18, 2021

- New environment variables for defaults, `CMAKE_BUILD_TYPE` and `CMAKE_CONFIGURATION_TYPES`
- New environment variable `CMAKE_INSTALL_MODE` for install types (symlinks)
- New CMAKE_REQUIRE_FIND_PACKAGE_<PackageName> variable to convert an optional find to a required one
- CMAKE_<LANG>_EXTENSIONS_DEFAULT comes from the compiler
- `CMakeDependentOption` uses normal conditional syntax now
- CTest can now modify environment variables
- Some generators now use external (system) markers on includes for MSVC

1.4.17 CMake 3.21 : Colors

Different message types now have different colors! There's now a nice variable to see if you are in the top level project. Lots of continued cleanup and specialized new features, such as adding the HIP language and C17 and C23 support. Presets continue to be improved.

- Initially released July 14, 2021
- Preliminary support for MSVC 2022
- CMAKE_<LANG>_LINKER_LAUNCHER added for make and ninja
- HIP added as a language
- C17 and C23 support added
- --install-prefix <dir> and --toolchain <file> added when running CMake
- Messages printed are colored by message type!
- Support for MSYS, including `FindMsys`
- The `file(` command got several updates, including `EXPAND_TILDE`
- Support for runtime dependencies and artifacts added to `install`
- `PROJECT_IS_TOP_LEVEL` and <PROJECT-NAME>_IS_TOP_LEVEL finally added
- Caching improvements for the `find_` commands

1.4.18 CMake 3.20 : Docs

The CMake docs received a major boost in productivity by adding "new in" tags to quickly see what was added without having to toggle documentation versions! C++ 23 support added. Source files must have the extension listed now, and LANGUAGE is always respected. Quite a bit of cleanup was done; make sure your code is tested with `...3.20` before deploying that as your maximum. Presets continue to be improved.

- Initially released March 23, 2021
- Support added for C++23
- CUDAARCHS environment variable for setting CUDA architectures
- The new `IntelLLVM` compilers are now supported (OneAPI 2021.1), and `NVHPC` NVIDIA HPC SDK, as well
- Some expanded generator expression support in custom commands/targets, install renaming
- New `cmake_path` command for working with paths
- `try_run` now has a `WORKING_DIRECTORY`
- More features for the `file(GENERATE` command
- Several removals, like `cmake-server`, `WriteCompilerDetectionHeader` (if policy set to 3.20+), and a few things that have newer methods now.
- Source files must include the extension

1.4.19 CMake 3.19 : Presets

You can now add presets in JSON form, and users will get the preset default. `find_package` can now take a version range, and some specialty find modules, like FindPython, have custom support for it. A lot of new controls were added for permissions. Further support for generator expressions in more places.

- Initially released November 18, 2020

- New CMake presets files now supported - you can set defaults for your project per generator, or you can make User presets. PSA: Please add `CMakeUserPresets.json` to your `.gitignore`, even if you do not use `CMakePresets.json`.
- CMake now uses the new build system introduced in XCode 12+
- MSVC for Android now supported
- `cmake -E create_hardlink` was added
- `add_test` finally properly supports whitespace in test names
- You can now `DEFER` `cmake_language` to run at the end of the directory processing
- Lots of new `file` options, like temporary downloads and `COMPRESSION_LEVEL` for `ARCHIVE_CREATE`
- `find_package` supports a version range
- `DIRECTORY` can now include a binary directory in property commands
- New `JSON` commands for `string`
- New `OPTIMIZE_DEPENDENCIES` property and `CMAKE_*` variable for smartly dropping dependencies of static and object libraries.
- PCH support expanded with `PCH_INSTANTIATE_TEMPLATES` property and `CMAKE_*` variable.
- Check modules have been expanded with `CUDA` and `ISPC` languages
- FindPython: `Python*_LINK_OPTIONS` added
- `compute-sanitizer` for ctest now supports CUDA for memcheck

1.4.20 CMake 3.18 : CUDA with Clang & CMake macro language

CUDA now supports Clang (without separable compilation). A new `CUDA_ARCHITECTURES` property was implemented to better support targeting CUDA hardware. A new `cmake_language` command supports calling cmake commands and expressions from strings. Lots of other meta changes that could make new designs available; calling functions by variable, evaluating arbitrary CMake by string, and configure files directly from strings. Many other nice tiny features and papercut fixes are sprinkled throughout, a small selection is below.

- Initially released July 15, 2020
- `cmake` can `cat` files together now
- New profiling mode for `cmake`
- `cmake_language` with `CALL` and `EVAL`
- `export` requires `APPEND` if used multiple times (in CMake language level 3.18+)
- You can archive directly from `file()`
- `file(CONFIGURE` is a nicer form of `configure_file` if you already have a string to produce
- Other `find_*` commands gain `find_package`'s `REQUIRED` flag
- `NATURAL` sorting in `list(SORT` added
- More options for handling properties with DIRECTORY scope
- `CUDA_ARCHITECTURES` was added
- New `LINK_LANGUAGE` generator expressions (`DEVICE`/`HOST` versions too)
- Source can be a subdirectory for `FetchContent`

1.4.21 CMake 3.17 : More CUDA

A FindCUDAToolkit was finally added, which allows finding and using the CUDA toolkit without enabling the CUDA language! CUDA now is a bit more configurable, such as linking to shared libraries. Quite a bit more polish in the expected areas, as well, like FindPython. Finally, you can now iterate over multiple lists at a time.

- Initially released March 20, 2020
- `CUDA_RUNTIME_LIBRARY` can finally be set to Shared!
- FindCUDAToolkit finally added
- `cmake -E rm` replaces older remove commands
- CUDA has meta features like `cuda_std_03`, etc.
- You can track the searches for packages with `--debug-find`
- 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)

1.4.22 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.

- Initially released November 26, 2019
- 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

1.4.23 CMake 3.15 : CLI upgrade

This release has many smaller polishing changes, including several 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.

- Initially released July 17, 2019
- `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

1.4.24 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.

- Initially released March 14, 2019
- 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.

1.4.25 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).

- Initially released November 20, 2018
- 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

1.4.26 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.

- Initially released July 17, 2018
- 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

1.4.27 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).

- Initially released March 28, 2018
- 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)

1.4.28 CMake 3.10 : CppCheck

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

- Initially released November 20, 2017
- 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_host_system_information` can access much more information.

1.4.29 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.

- Initially released July 18, 2017
- 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

1.4.30 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+!

- Initially released April 10, 2017
- 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 IDEs to reflect the project folder structure!)

1.4.31 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.

- Initially released November 11, 2016
- `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

1.4.32 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.

- Initially released July 7, 2016
- 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

1.4.33 CMake 3.5 : ARM

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

- Initially released March 8, 2016
- 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

1.4.34 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.

- Initially released November 12, 2015
- 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

1.4.35 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.

- Initially released July 23, 2015
- IN_LIST added to if
- *_INCLUDE_WHAT_YOU_USE property added
- COMPILE_LANGUAGE generator expression (limited support in some generators)

1.4.36 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.

- Initially released March 11, 2015
- `continue()` inside loops
- File and directory locks added

1.4.37 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.

- Initially released December 17, 2014
- C++11 Support
- Compile features support
- Sources can be added later with `target_sources`
- Better support for generator expressions and INTERFACE targets

1.4.38 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.

- Initially released June 10, 2014
- New documentation
- INTERFACE libraries
- Project VERSION support
- Exporting build trees easily
- Bracket arguments and comments available (not widely used)
- Lots of improvements

2. The Basics

2.1 Introduction to the basics

2.1.1 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 3.15)
```

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.[3] 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![4] The version of CMake will also dictate the policies, which define behavior changes. CMake no longer keeps the very oldest behaviors around: starting in CMake 4.0, setting `cmake_minimum_required` (or the start of a range) below 3.5 is a hard error, and values below roughly 3.10 produce a deprecation warning. So pick a modern minimum, like the range recommended below. A list of policies and versions is available at policies.

Starting in CMake 3.12, this supports a range, such as `VERSION 3.15...4.4`; this means you support as low as 3.15 but have also tested it with the new policy settings up to 4.4. 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 3.1-3.11 will only set the old version of the policies, since those versions didn't treat this specially). 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 3.15...4.4)
```

> **Tip**
>
> 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.

2.1.2 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+), `Swift` (3.15+), `OBJC` and `OBJCXX` (3.16+), `ISPC` (3.18+), and `HIP` (3.21+). `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.

[3] 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.

[4] 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.

> **Tip**
>
> You can add comments with the # character. CMake does have an inline syntax for comments too, but it's rarely used.

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

2.1.3 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.

2.1.4 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).[5]

2.1.5 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

[5]The `::` syntax was originally intended for `INTERFACE IMPORTED` libraries, which were explicitly supposed to be libraries defined outside the current project. Historically, most of the `target_*` commands didn't work on `IMPORTED` libraries, so an `ALIAS` target was the only way to get a `::` name for your own targets; that limitation was removed in CMake 3.11, but making a namespaced `ALIAS` for your own library is still good practice (it matches what users get from `find_package`).

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++ standards 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!

2.1.6 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 3.15...4.4)

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 PRIVATE calclib)
```

2.2 Variables and the Cache

2.2.1 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}`.[6] 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}"`).

2.2.2 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 (the `INTERNAL` type implies `FORCE` and hides the entry from listings and GUIs):

```
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.

2.2.3 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.

---

[6] `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.

2.2.4 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.

2.2.5 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.[7]

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

2.3 Programming in CMake

2.3.1 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 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

2.3.2 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.[8] 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.[9]

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-

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

[9] The CMake docs splits expressions into Informational, Logical, and Output.

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>
)
```

2.3.3 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 ${ARGN}")
    set(${REQUIRED_ARG} "From SIMPLE" PARENT_SCOPE)
endfunction()

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

The output would be:

```
-- Simple arguments: This, followed by Foo;Bar
Output: From SIMPLE
```

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.

2.3.4 Arguments

CMake has a named variable system that you've already seen in most of the built-in CMake functions. You can use it with the `cmake_parse_arguments` function. 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`.

2.4 Communication with your code

2.4.1 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 `COPYONLY` 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`:

2.4.1.1 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@"
```

2.4.1.2 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).

2.4.2 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.

2.5 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 patterns,
- 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
  - LICENSE.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:

```cmake
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 build subdirectory. Feel
free to remove CMakeCache.txt and CMakeFiles.")
endif()
```

See the extended code example here.

2.6 Running other programs

2.6.1 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 --recursive failed
with ${GIT_SUBMOD_RESULT}, please checkout submodules")
    endif()
endif()
```

2.6.2 Running a command at build time

Build time commands are a bit trickier. 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(Python REQUIRED COMPONENTS Interpreter)
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.

2.6.3 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.

2.7 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.

```cmake
# 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 3.15...4.4)

# 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.

3. Extra features

3.1 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.

3.1.1 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()
```

3.2 C++ standards

CMake has excellent support for selecting the C++ standard, all the way through C++26. There are two recommended ways to do it: per-target compile features, and the standard variables/properties. Both have been around since well before this book's minimum CMake version, so use whichever fits.

3.2.1 Meta compiler features

The most powerful way, with the nicest syntax and the best target behavior for mixing standards and options, is `target_compile_features` with a `cxx_std_*` meta-feature. 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, you can pick from `cxx_std_11`, `cxx_std_14`, `cxx_std_17` (all CMake 3.8+), `cxx_std_20` (CMake 3.12+), `cxx_std_23` (CMake 3.20+), and `cxx_std_26` (CMake 3.25+). The second line is optional, but will avoid extensions being added; without it you'd get things like `-std=gnu++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; the standard is treated as a minimum, and the highest one requested wins.

> **Note**
>
> CMake 3.1 originally supported granular compile features, like `cxx_constexpr`, that let you request individual language features instead of a standard (list here). This didn't scale; CMake stopped adding granular features after C++14, and the docs now steer you to the `cxx_std_*` meta-features instead.

3.2.2 Global and property settings

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

```
set(CMAKE_CXX_STANDARD 11 CACHE STRING "The C++ standard to use")
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=gnu++11`. This method isn't bad for a final package, but shouldn't be used by a library. You should always set this as a cached variable, so you can override it to try a new version easily (or if this gets used as a library, this is the only way to override it - but again, don't use this for libraries). 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 compile features have for populating `PRIVATE` and `INTERFACE` properties, so it really is only useful on final targets.

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

> **Danger**
>
> Don't set manual flags yourself. You'll then become responsible for maintaining 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.

3.3 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 its own chapter.

3.3.1 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.

3.3.2 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.

3.3.3 Interprocedural optimization

`INTERPROCEDURAL_OPTIMIZATION`, best known as *link time optimization* and the `-flto` flag, is well supported by CMake. You can turn this on globally with `CMAKE_INTERPROCEDURAL_OPTIMIZATION` or per-target with the `INTERPROCEDURAL_OPTIMIZATION` property. Setting this to `ON` on a target is an error if the compiler doesn't support it (see CMP0069). You can use `check_ipo_supported()`, from the built-in CheckIPOSupported module, to check support beforehand:

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

3.4 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.

3.4.1 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()
```

3.4.2 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`

3.4.3 Clang tidy

This is the command line for running clang-tidy, as a list (remember, a semicolon separated string is a list).

Here is a simple example of using Clang-Tidy:

```
~/package # cmake -S . -B build-tidy -DCMAKE_CXX_CLANG_TIDY="$(which clang-tidy);-fix"
~/package # cmake --build build-tidy -j 1
```

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 (maybe something like `DO_CLANG_TIDY`) instead of the `CMAKE_CXX_CLANG_TIDY` variable, then add it to your target properties as you create them. You can find clang-tidy in your path like this:

```
find_program(
    CLANG_TIDY_EXE
    NAMES "clang-tidy"
    DOC "Path to clang-tidy executable"
)
```

3.4.4 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 or with brew (macOS) with `brew install include-what-you-use`. Then, you can pass this into your build without modifying the source:

```
~/package # cmake -S . -B build-iwyu -DCMAKE_CXX_INCLUDE_WHAT_YOU_USE=include-what-you-use
```

Finally, you can collect the output and (optionally) apply the fixes:

```
~/package # cmake --build build-iwyu 2> iwyu.out
~/package # fix_includes.py < iwyu.out
```

(You should check the fixes first, or touch them up after applying!)

3.4.5 Link what you use

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

3.4.6 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):

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

3.5 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.

3.5.1 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)
```

When the condition (here, `VAL1 AND VAL2`) is true, this behaves like a normal `option(BUILD_TESTS "Build your tests" ON)`. When the condition is false, however, it does not just hide the option; it sets `BUILD_TESTS` to the fallback value (`OFF` here) as a normal (non-cache) variable, shadowing any value the user may have set in the cache. So `BUILD_TESTS` is only user-configurable while the condition holds; roughly:

```
if(VAL1 AND VAL2)
    option(BUILD_TESTS "Build your tests" ON)
else()
    set(BUILD_TESTS OFF)
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`.

3.5.2 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.

3.5.3 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).

3.5.4 `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`.

3.5.5 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 explicitly 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`.

3.6 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.

3.6.1 Folders for targets

Some IDEs, like Xcode, support folders. Since CMake 3.26 (see policy CMP0143), the `USE_FOLDERS` global property defaults to `ON`; before that, you had to enable it manually to allow CMake to organize your files by folders:

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

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

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

Folders can be nested with `/`.

3.6.2 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

```cmake
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:

```cmake
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).

3.7 Debugging code

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

3.7.1 CMake debugging

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

3.7.1.1 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 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
)
```

3.7.1.2 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.

3.7.1.3 Debug and trace messages

The `message` command supports log levels beyond `STATUS`, including `VERBOSE`, `DEBUG`, and `TRACE` (CMake 3.15+). Messages at the quieter levels are hidden unless you raise the log level when configuring:

```
cmake -S . -B build --log-level=DEBUG
```

You can also set the `CMAKE_MESSAGE_LOG_LEVEL` variable (cache it if you want it to persist across reruns). This lets you leave detailed logging in your CMake code with `message(DEBUG ...)` without cluttering a normal configure.

3.7.1.4 The CMake debugger

CMake 3.27+ has an interactive debugger speaking the Debug Adapter Protocol, the same protocol most editors use for debugging code. You can launch it directly:

```
cmake -S . -B build --debugger
```

though it's nicest from a tool with DAP support, such as the CMake Tools extension for VS Code; you can set breakpoints in your CMakeLists, step through the configure, and inspect variables and properties.

3.7.2 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 generator expressions like `$<CONFIG:Debug>` or `$<CONFIG:Release>`.

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

4. Using Other Projects

4.1 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.

4.2 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:

```
$ 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 --recursive
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.

4.2.1 Bonus: Git version number

Since you've already found Git with `find_package(Git)` in the snippet above, you can also use `${GIT_EXECUTABLE}` to capture the current commit hash into a variable, for example to embed it in your build:

```cmake
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)
```

4.3 Downloading Projects

4.3.1 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 built-in tool for this, ExternalProject, has to work around this by doing the build itself. (It can, however, build non-CMake packages as well).[10]

4.3.2 Downloading Method: configure time

If you prefer configure time, the modern answer is the built-in FetchContent module (CMake 3.11+, much improved in 3.14). If you are stuck supporting very old CMake, the Crascit/DownloadProject repository was a drop-in solution, but it has been superseded by FetchContent. 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.

[10] Note that ExternalData is the tool for non-package data.

4.4 FetchContent (CMake 3.14+)

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 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_MakeAvailable(MyName)` to make it available; if it is a package with a `CMakeLists.txt`, this adds it to your build (like `add_subdirectory` would).

For example, to download Catch2:

```
FetchContent_Declare(
  catch
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
  GIT_TAG        v3.15.0
)

FetchContent_MakeAvailable(catch)
```

If you are maintaining older code, you may still run across the original pattern of checking `MyName_POPULATED` and calling `FetchContent_Populate(MyName)` by hand; that was deprecated in CMake 3.30 (CMP0169) and warns on current CMake, so just use `FetchContent_MakeAvailable`.

See the example here.

5. Testing

5.1 Testing

5.1.1 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:TargetName>)
```

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

5.1.2 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
    "${CMAKE_CTEST_COMMAND}"
             --build-and-test "${MyProject_SOURCE_DIR}/examples/simple"
                              "${CMAKE_CURRENT_BINARY_DIR}/simple"
             --build-generator "${CMAKE_GENERATOR}"
             --test-command "${CMAKE_CTEST_COMMAND}"
)
```

5.1.3 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.

5.2 GoogleTest

GoogleTest and GoogleMock are classic options; personally, I would recommend Catch2 instead, as GoogleTest heavily follows the Google development philosophy; it drops old compilers very quickly, it assumes users want to live at HEAD, etc. Adding GoogleMock is also often painful - and you need GoogleMock to get matchers, which are a default feature in Catch2 (but not doctest).

5.2.1 FetchContent method

The FetchContent module's own documentation uses GoogleTest as its example, and it is the simplest way to get it:

```
include(FetchContent)

FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG        v1.17.0
)

FetchContent_MakeAvailable(googletest)
```

This provides the namespaced targets `GTest::gtest`, `GTest::gtest_main`, `GTest::gmock`, and `GTest::gmock_main`; these are the same ones `find_package(GTest)` gives you for an installed copy (modern GoogleTest supports a normal CMake install, so using a packaged copy works fine too).

5.2.2 Submodule method

To use this method, just checkout GoogleTest as a submodule:[11]

```
git submodule add --branch=v1.17.x ../../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
)
```

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

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)
```

5.2.3 Adding tests

Whichever way you added GoogleTest, to add a test, I'd recommend the following macro:

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

5.3 Catch

Catch2 is a powerful, idiomatic testing solution similar in philosophy to PyTest for Python. It supports a wider range of compilers than GTest, and is quick to support new things, like M1 builds on macOS. It also has a smaller but faster twin, doctest, which is quick to compile but misses features like matchers. The current Catch2 v3 line is a normal library (the old v2 single-header design is gone), and it requires C++14 or better. To use Catch2 in a CMake project, there are several options.

5.3.1 Configure methods

Catch2 has nice CMake support; you can get it with FetchContent (or submodules, or an installed copy via `find_package(Catch2 3)`). Both the `extended-project` and `fetch` examples use FetchContent. See the docs.

```
include(FetchContent)

FetchContent_Declare(
  catch
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
  GIT_TAG        v3.15.0
)
FetchContent_MakeAvailable(catch)
```

This provides two targets: `Catch2::Catch2`, and `Catch2::Catch2WithMain`, which also provides `main()` for you; unlike v2, you no longer write a `main.cpp` with `CATCH_CONFIG_MAIN`:

```
add_executable(my_tests test.cpp)
target_link_libraries(my_tests PRIVATE Catch2::Catch2WithMain)
```

Your test files then just include what they use, like `<catch2/catch_test_macros.hpp>` (the v2 catch-all `<catch2/catch.hpp>` is gone).

5.3.2 Registering tests with CTest

Catch2 also ships CMake modules in its `extras` folder that register each `TEST_CASE` with CTest individually. The one to use is `catch_discover_tests` from the `Catch` module (`ParseAndAddCatchTests` is deprecated in v3):

```
list(APPEND CMAKE_MODULE_PATH ${catch_SOURCE_DIR}/extras)
include(CTest)
include(Catch)
catch_discover_tests(my_tests)
```

If you installed Catch2 instead of adding it as a subproject, the module is available directly after `find_package(Catch2 3 REQUIRED)`.

5.3.3 Vendoring

If you can't (or don't want to) download at configure time, Catch2 v3 provides an amalgamated distribution: a pair of files, `extras/catch_amalgamated.hpp` and `extras/catch_amalgamated.cpp`, that you can drop into your project and build as a normal library:

```
add_library(catch2 STATIC extern/catch2/catch_amalgamated.cpp)
target_include_directories(catch2 PUBLIC extern/catch2)
target_compile_features(catch2 PUBLIC cxx_std_14)
```

The amalgamated files include the default `main()` as well. This is fine as a plain (non-exported) target since you won't be exporting your tests.

6. Exporting and Installing

6.1 Exporting and Installing

There are three good ways and one bad way to allow others to use your library. The good ways are including your project as a subdirectory, exporting your build tree, and installing `*Config.cmake` scripts:

6.1.1 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 `<mypackage>Config.cmake` instead as listed below.

6.1.2 Add Subproject

A package can include your project in a subdirectory, and then use `add_subdirectory` on the subdirectory. This is 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.

6.1.3 Exporting and Installing

The other two good ways, exporting your build tree and installing `*Config.cmake` scripts, share most of their machinery; they are the topic of the next chapters in this section.

6.2 Installing

Install commands cause a file or target to be "installed" into the install tree when you `make install`. Rather than hard-coding destinations like `lib` (wrong on distributions that use `lib64`, for example), use the standard GNUInstallDirs module, which provides cache variables like `CMAKE_INSTALL_LIBDIR` with platform-appropriate defaults. Your basic target install command looks like this:

```cmake
include(GNUInstallDirs)
install(TARGETS MyLib
        EXPORT MyLibTargets
        LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
        ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
        RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
        INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
        )
```

The various destinations are only needed if you have a library, static library, or program to install; since CMake 3.14, you can omit them entirely and these sensible defaults are used. 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. This uses the project version set by your `project()` command:

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

(You can also leave out the `VERSION` argument entirely; it defaults to the project version.)

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 latter 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):

```cmake
install(EXPORT MyLibTargets
        FILE MyLibTargets.cmake
        NAMESPACE MyLib::
        DESTINATION ${CMAKE_INSTALL_LIBDIR}/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 then look like this:

```cmake
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 ${CMAKE_INSTALL_LIBDIR}/cmake/MyLib
        )
```

That's it! Now once you install a package, there will be files in `${CMAKE_INSTALL_LIBDIR}/cmake/MyLib` (usually `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 point CMake at the package manually, by adding the install prefix to `CMAKE_PREFIX_PATH`, or by setting `MyLib_DIR` to the directory containing the config file (or `MyLib_ROOT`, CMake 3.12+), and CMake gives the user nice help output if the configure file is not found.

The CMakePackageConfigHelpers module mentioned above has additional tools to help write a more relocatable `Config.cmake` file. Refer to the CMake documentation on configure_package_config_file (used instead of `configure_file`) and the `@PACKAGE_INIT@` substitution string to get

- a set of automatically defined `PACKAGE_<var>` variables (for relative path versions of `<var>`) and
- a `set_and_check()` alternative to `set()` to automatically check for path existence.

6.3 Exporting

Danger

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 later, this will no longer happen unless you set `CMAKE_EXPORT_PACKAGE_REGISTRY` as shown 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 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.

6.4 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:

```cmake
# 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}/LICENSE")
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 `CPACK_SOURCE_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:

```cmake
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:

```cmake
include(CPack)
```

7. Libraries

7.1 Finding Packages

There are two ways to find packages in CMake: "Module" mode and "Config" mode.

In Module mode, `find_package(XYZ)` looks for a file named `FindXYZ.cmake`, either shipped with CMake itself or provided by your project (via `CMAKE_MODULE_PATH`). Find modules exist because the package being searched for doesn't know anything about CMake; the module pokes around the system looking for the right files. The following pages cover several of the most useful find modules CMake ships with.

In Config mode, the package itself ships an `XYZConfig.cmake` (or `xyz-config.cmake`) file describing exactly how to use it, as shown in the previous chapter. This is the preferred mechanism for packages that support CMake; CMake searches the standard install locations, and you can point it at unusual ones with `CMAKE_PREFIX_PATH` or `XYZ_DIR`.

By default, `find_package` tries Module mode first, then falls back to Config mode; you can restrict it to one or the other with the `MODULE` or `CONFIG` keywords. Either way, a well-behaved package provides imported targets, usually namespaced like `XYZ::XYZ` — linking to those with `target_link_libraries` is the modern interface, rather than using `XYZ_INCLUDE_DIRS`/`XYZ_LIBRARIES` style variables.

7.2 CUDA

CUDA support is available in two flavors. The new method, introduced in CMake 3.8 (3.9 for Windows), should be strongly preferred over the old, hacky method - I only mention the old method due to the high chances of an old package somewhere having it. Unlike the older languages, CUDA support has been rapidly evolving, and building CUDA is hard, so I would recommend you *require a very recent version* of CMake! CMake 3.17 and 3.18 have a lot of improvements directly targeting CUDA.

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

7.2.1 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 `check_language`:

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

You can see if CUDA is present by checking `CMAKE_CUDA_COMPILER` (was missing until CMake 3.11).

You can check variables like `CMAKE_CUDA_COMPILER_ID` (for nvcc, this is `"NVIDIA"`, Clang was added in CMake 3.18). You can check the version with `CMAKE_CUDA_COMPILER_VERSION`.

7.2.2 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()
```

If you are looking for CUDA's standard level, in CMake 3.17 a new collection of compiler features were added, like `cuda_std_11`. These have the same benefits that you are already used to from the `cxx` versions.

7.2.2.1 Adding a library / executable

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

You can also use separable compilation:

```
set_target_properties(mylib PROPERTIES
                            CUDA_SEPARABLE_COMPILATION ON)
```

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

7.2.2.2 Targeting architectures

When you build CUDA code, you generally should be targeting an architecture. If you don't, you compile PTX for the lowest supported architecture, which provide the basic instructions but is compiled at runtime, making it potentially much slower to load.

All cards have an architecture level, like "7.2". You have two choices; the first is the code level; this will report to the code being compiled a version, like "5.0", and it will take advantage of all the features up to 5.0 but not past (assuming well written code / standard libraries). Then there's a target architecture, which must be equal or greater to the code architecture. This needs to have the same major number as your target card, and be equal to or less than the target card. So 7.0 would be a common choice for our 7.2 card. Finally, you can also generate PTX; this will work on all future cards, but will compile just in time.

In CMake 3.18, it became very easy to target architectures. If you have a version range that includes 3.18 or newer, you will be using `CMAKE_CUDA_ARCHITECTURES` variable and the `CUDA_ARCHITECTURES` property on targets. You can list values (without the `.`), like 50 for arch 5.0. This will generate for both the real (SASS) and virtual architecture (PTX). Passing values of '50-real' will only generate for SASS, while passing '50-virtual' will only generate for PTX. If set to OFF, it will not pass architectures.

In CMake 3.23, the architectures values were extended to support the user friendly values 'all' and 'all-major'; CMake 3.24 added 'native'.

7.2.2.3 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:

```
set(opt "$<$<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>>:-Xcompiler=${CUDA_flags}>"
            )
    endif()
endfunction()
```

7.2.2.4 Useful variables

You can use `FindCUDAToolkit` to find a variety of useful targets and variables even without enabling the CUDA language.

```
cmake_minimum_required(VERSION 3.17)
project(example LANGUAGES CXX)

find_package(CUDAToolkit REQUIRED)
add_executable(uses_cublas source.cpp)
target_link_libraries(uses_cublas PRIVATE CUDA::cublas)
```

Variables that using `find_package(CUDAToolkit)` provides:

- `CUDAToolkit_BIN_DIR`: Directory that holds the `nvcc` executable
- `CUDAToolkit_INCLUDE_DIRS`: Lists of directories containing headers for built-in Thrust, etc
- `CUDAToolkit_LIBRARY_DIR`: Directory that holds the CUDA runtime library

Variables that enabling the `CUDA` language provides:

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

> **Warning**
>
> The classic FindCUDA module was deprecated for years and was finally removed in CMake 3.27 (policy CMP0146); on any modern CMake, `find_package(CUDA)` is a hard error. Before CMake 3.18, the following did still require FindCUDA:
>
> - CUDA version checks / picking a version
> - Architecture detection (Note: 3.12 fixes this partially)
> - Linking to CUDA libraries from non-.cu files

7.2.3 Classic FindCUDA [WARNING: REMOVED in CMake 3.27] (for reference only)

The recipe below only works on CMake older than 3.27, since the module no longer exists after that. If you are stuck supporting such an old 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
```

7.3 OpenMP

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 flag-based method, it will also correctly set the library link line differently from the compile line if needed. It supports OpenMP on macOS, too (if the library is available, such as with `brew install libomp`).

7.4 Boost library

Boost has shipped its own CMake configuration file, `BoostConfig.cmake`, since Boost 1.70. CMake used to provide a FindBoost module as well, but it has not kept up and was disabled in CMake 3.30 (policy CMP0167); on current CMake, `find_package(Boost)` simply uses the config file Boost installs. A nice consequence of config mode is that Boost describes its own targets and dependencies, so it doesn't matter if your Boost is newer than your CMake.

You can customize which flavor of the libraries gets selected using a set of variables that you set before searching for Boost; here are the three most common ones:

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

The imported targets are in the `Boost::` namespace. `Boost::headers` is the header-only part (it was called `Boost::boost` historically). The compiled libraries, like `Boost::filesystem`, are available as well, and pull in their dependencies as needed.

Here is an example for using the Boost.Filesystem library:

```
find_package(Boost 1.70 REQUIRED COMPONENTS filesystem)
message(STATUS "Boost version: ${Boost_VERSION}")

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

7.5 MPI

MPI, like OpenMP, is best used through the imported target its find module provides:

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

7.6 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, any current version of ROOT (the 6.3x series at the time of writing) behaves like a proper CMake project: it provides imported targets, exports C++ standard features for those targets, and loads its CMake macros automatically. If you really must support an ancient ROOT (pre-6.20), see the appendix at the end.

7.6.1 Finding ROOT

ROOT 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`).

7.6.2 The right way (Targets)

ROOT's imported targets carry the include directories, flags, and dependencies you need. 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. For a typical ROOT install this would be something like:

- `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`

7.6.3 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).

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})
```

7.6.4 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.

7.6.5 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. Your source code will need the following modifications to support dictionary generation:

- Your class definition should end with `ClassDef(MyClassName, 1)`
- Your class implementation should have `ClassImp(MyClassName)` in it

ROOT provides `rootcling` and `genreflex` (a legacy interface to `rootcling`) binaries which produce the source files required to build the dictionary. It also defines `root_generate_dictionary`, a CMake function to invoke `rootcling` during the build process. The macros file defining it is loaded automatically, so the function is available as soon as `find_package(ROOT)` runs. (Only ancient ROOT required manually including the macros file; it was renamed to `RootMacros.cmake` and auto-loaded in 6.20 — see the appendix.)

`rootcling` uses a special header file with a specific formula to determine which parts to generate dictionaries for. The name of this file may have any prefix, but **must** end with `LinkDef.h`. Once you have written this header file, the dictionary generation function can be invoked.

7.6.5.1 Manually building the dictionary

Sometimes, you might want to ask ROOT to generate the dictionary, and then add the source file to your library target yourself. You can call the `root_generate_dictionary` with the name of the dictionary, e.g. `G__Example`, any required header files, and finally the special `LinkDef.h` file, listed after `LINKDEF`:

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

This command will create three files:

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

The name (`${NAME}`) of the target that you must create is determined by the dictionary name; if the dictionary name starts with `G__`, it will be removed. Otherwise, the name is used directly.

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`.

7.6.5.2 Building the dictionary with an existing target

Instead of manually adding the generated dictionary to your library sources, you can ask ROOT to do this for you by passing a `MODULE` argument. This argument should specify the name of an existing build target:

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

The full name of the dictionary (e.g. `G__Example`) should not be identical to the `MODULE` argument.

7.6.6 Using Old ROOT

If you really have to use an ancient ROOT (pre-6.20), you'll need to load the dictionary macros manually before using `root_generate_dictionary`:

```
include("${ROOT_DIR}/modules/RootNewMacros.cmake")
# For ROOT versions older than 6.16, things break
# if nothing is in the global include list!
if (${ROOT_VERSION} VERSION_LESS "6.16")
    include_directories(ROOT_NONEXISTENT_DIRECTORY_HACK)
endif()
```

(The `include_directories` line works around a bug in the NewMacros file that broke dictionary generation unless there was at least one global include directory or `inc` folder; the directory listed doesn't need to exist.)

You'll also need to patch up the missing target properties and flags yourself, with 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_DEFINITIONS})
    # Remove -D or /D if present
    string(REGEX REPLACE [=[^[-//]D]=] "" _flag ${_flag})
    set_property(TARGET ROOT::Flags_CXX APPEND PROPERTY
INTERFACE_COMPILE_DEFINITIONS ${_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!
```

7.6.7 A Simple ROOT Project

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

7.6.7.1 examples/root-usefile/CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.15...4.4)

project(RootUseFileExample LANGUAGES CXX)

find_package(ROOT 6.16 CONFIG REQUIRED)

## [core]
# 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})
## [core]
```

7.6.7.2 examples/root-usefile/SimpleExample.cxx

```cpp
#include <TLorentzVector.h>

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

7.6.8 A Simple ROOT Project

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

7.6.8.1 examples/root-simple/CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 3.15...4.4)

project(RootSimpleExample LANGUAGES CXX)

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

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

7.6.8.2 examples/root-simple/SimpleExample.cxx

```cpp
#include <TLorentzVector.h>

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

7.6.9 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.

7.6.9.1 examples/root-dict/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.15...4.4)

project(RootDictExample LANGUAGES CXX)

set(CMAKE_CXX_STANDARD
    11
    CACHE STRING "C++ standard to use")
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_PLATFORM_INDEPENDENT_CODE ON)

find_package(ROOT 6.20 CONFIG REQUIRED)
# If you want to support <6.20, add this line:
# include("${ROOT_DIR}/modules/RootNewMacros.cmake")
# However, it was moved and included by default in 6.201

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)

## Alternative to add the dictionary to an existing target:
# add_library(DictExample SHARED DictExample.cxx DictExample.h)
# target_include_directories(DictExample PUBLIC "${CMAKE_CURRENT_SOURCE_DIR}")
# target_link_libraries(DictExample PUBLIC ROOT::Core)
# root_generate_dictionary(G__DictExample DictExample.h MODULE DictExample
LINKDEF DictLinkDef.h)
```

7.6.9.2 Supporting files

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

7.6.9.2.1 examples/root-dict/DictExample.cxx

```
#include "DictExample.h"

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

ClassImp(Simple)
```

7.6.9.2.2 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.

7.6.9.2.3 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
```

7.6.9.3 Testing it

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

7.6.9.3.1 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();
}
```

7.7 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.

7.7.1 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)
```

7.7.2 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.