

PROFESSIONAL **CMAKE**

A PRACTICAL GUIDE



CRAIG SCOTT

Professional CMake: A Practical Guide

18th Edition

ISBN 978-1-925904-30-7

© 2018-2024 by Craig Scott

This book or any portion thereof may not be reproduced in any manner or form without the express written permission of the author, with the following specific exceptions:

- The original purchaser may make personal copies exclusively for their own use on their electronic devices, provided that all reasonable steps are taken to ensure that only the original purchaser has access to such copies.
- Permission is given to use any of the code samples in this work without restriction. Attribution is not required.

For the removal of doubt, note that the above means this book may not be included in any data set used to train any sort of AI system or product, except if the original purchaser is the only person with access to that AI system or product.

The advice and strategies contained within this work may not be suitable for every situation. This work is sold with the understanding that the author is not held responsible for the results accrued from the advice in this book.

<https://crascit.com>

Table of Contents

Preface	1
Acknowledgments	2
I: Getting Started	4
1. Introduction	5
2. Setting Up A Project	7
2.1. In-source Builds	7
2.2. Out-of-source Builds	8
2.3. Generating Project Files	8
2.4. Running The Build Tool	9
2.5. Recommended Practices	10
3. A Minimal Project	11
3.1. Managing CMake Versions	11
3.2. The project() Command	13
3.3. Building A Basic Executable	14
3.4. Commenting	14
3.5. Recommended Practices	15
4. Building Simple Targets	16
4.1. Executables	16
4.2. Defining Libraries	17
4.3. Linking Targets	18
4.4. Linking Non-targets	20
4.5. Old-style CMake	20
4.6. Recommended Practices	21
5. Basic Testing And Deployment	23
5.1. Testing	23
5.2. Installing	25
5.3. Packaging	28
5.4. Recommended Practices	30
II: Fundamentals	31
6. Variables	32
6.1. Variable Basics	32
6.2. Environment Variables	34
6.3. Cache Variables	34
6.4. Scope Blocks	42
6.5. Printing Variable Values	44
6.6. String Handling	44
6.7. Lists	47
6.8. Math	50

6.9. Recommended Practices	50
7. Flow Control	52
7.1. The if() Command	52
7.2. Looping	60
7.3. Recommended Practices	65
8. Using Subdirectories	66
8.1. add_subdirectory()	66
8.2. include()	71
8.3. Project-relative Variables	73
8.4. Ending Processing Early	75
8.5. Recommended Practices	77
9. Functions And Macros	79
9.1. The Basics	79
9.2. Argument Handling Essentials	80
9.3. Keyword Arguments	82
9.4. Returning Values	88
9.5. Overriding Commands	91
9.6. Special Variables For Functions	93
9.7. Other Ways Of Invoking CMake Code	94
9.8. Problems With Argument Handling	97
9.9. Recommended Practices	105
10. Properties	107
10.1. General Property Commands	107
10.2. Global Properties	111
10.3. Directory Properties	111
10.4. Target Properties	112
10.5. Source Properties	113
10.6. Cache Variable Properties	115
10.7. Test Properties	116
10.8. Installed File Properties	116
10.9. Recommended Practices	117
11. Generator Expressions	118
11.1. Simple Boolean Logic	119
11.2. Target Details	121
11.3. General Information	122
11.4. Path Expressions	123
11.5. List Expressions	124
11.6. Utility Expressions	125
11.7. Recommended Practices	126
12. Modules	127
12.1. Checking Existence And Support	128

12.2. Other Modules	135
12.3. Recommended Practices	135
13. Policies	137
13.1. Policy Control	137
13.2. Policy Scope	140
13.3. Recommended Practices	142
14. Debugging And Diagnostics	144
14.1. Log Messages	144
14.2. Color Diagnostics	151
14.3. Print Helpers	152
14.4. Tracing Variable Access	153
14.5. Debugging Generator Expressions	154
14.6. Profiling CMake Calls	155
14.7. Discarding Previous Results	156
14.8. Interactive Debugging	156
14.9. Recommended Practices	157
15. Build Type	159
15.1. Build Type Basics	159
15.2. Common Errors	161
15.3. Custom Build Types	162
15.4. Recommended Practices	166
16. Compiler And Linker Essentials	168
16.1. Target Properties	168
16.2. Target Property Commands	172
16.3. Target Property Contexts	179
16.4. Directory Properties And Commands	182
16.5. De-duplicating Options	185
16.6. Compiler And Linker Variables	186
16.7. Language-specific Compiler Flags	190
16.8. Compiler Option Abstractions	191
16.9. Recommended Practices	198

III: Builds In Depth **201**

17. Language Requirements	202
17.1. Setting The Language Standard Directly	202
17.2. Setting The Language Standard By Feature Requirements	205
17.3. Requirements For C++20 Modules	207
17.4. Recommended Practices	209
18. Advanced Linking	211
18.1. Require Targets For Linking	211
18.2. Customize How Libraries Are Linked	212

18.3. Propagating Up Direct Link Dependencies	215
18.4. Recommended Practices	219
19. Target Types	220
19.1. Executables	220
19.2. Libraries	221
19.3. Promoting Imported Targets	229
19.4. Recommended Practices	231
20. Custom Tasks	233
20.1. Custom Targets	233
20.2. Adding Build Steps To An Existing Target	236
20.3. Commands That Generate Files	237
20.4. Configure Time Tasks	242
20.5. Platform Independent Commands	246
20.6. Combining The Different Approaches	248
20.7. Recommended Practices	249
21. Working With Files	251
21.1. Manipulating Paths	251
21.2. Copying Files	257
21.3. Reading And Writing Files Directly	265
21.4. File System Manipulation	269
21.5. File Globbing	272
21.6. Downloading And Uploading	273
21.7. Recommended Practices	275
22. Specifying Version Details	277
22.1. Project Version	277
22.2. Source Code Access To Version Details	279
22.3. Source Control Commits	282
22.4. Recommended Practices	285
23. Libraries	287
23.1. Build Basics	287
23.2. Linking Static Libraries	288
23.3. Shared Library Versioning	289
23.4. Interface Compatibility	291
23.5. Symbol Visibility	295
23.6. Mixing Static And Shared Libraries	304
23.7. Recommended Practices	307
24. Toolchains And Cross Compiling	310
24.1. Toolchain Files	311
24.2. Defining The Target System	312
24.3. Tool Selection	314
24.4. System Roots	317

24.5. Compiler Checks	318
24.6. Examples	319
24.7. Android	320
24.8. Recommended Practices	323
25. Apple Features	325
25.1. CMake Generator Selection	325
25.2. Application Bundles	328
25.3. Frameworks	331
25.4. Loadable Bundles	334
25.5. Build Settings	334
25.6. Signing And Capabilities	341
25.7. Creating And Exporting Archives	344
25.8. Universal Binaries	347
25.9. XCFrameworks	350
25.10. Linking Frameworks	351
25.11. Embedding Frameworks And Other Things	353
25.12. Limitations	355
25.13. Recommended Practices	356
26. Build Performance	358
26.1. Unity Builds	358
26.2. Precompiled Headers	362
26.3. Build Parallelism	364
26.4. Optimizing Build Dependencies	370
26.5. Compiler Caches	373
26.6. Debug-related Improvements	378
26.7. Alternative Linkers	379
26.8. Recommended Practices	380
IV: Testing And Analysis	382
27. Testing Fundamentals	383
27.1. Defining And Executing A Simple Test	383
27.2. Test Environment	386
27.3. Pass / Fail Criteria And Other Result Types	388
27.4. Test Grouping And Selection	392
27.5. Cross-compiling, Emulators, And Launchers	396
27.6. JUnit Output	397
27.7. Recommended Practices	398
28. Test Resources And Constraints	400
28.1. Simple Test Dependencies	400
28.2. Test Fixtures	400
28.3. Parallel Execution	402

28.4. Simple Resource Constraints	404
28.5. Resource Groups	405
28.6. Recommended Practices	410
29. Build And Test Mode	411
29.1. Using Build And Test Mode	411
29.2. Use Cases	412
29.3. Alternatives	413
29.4. Recommended Practices	414
30. Test Frameworks	415
30.1. GoogleTest	415
30.2. Catch2	420
30.3. doctest	421
30.4. Recommended Practices	421
31. CDash Integration	423
31.1. Key CDash Concepts	423
31.2. Executing Pipelines And Actions	424
31.3. CTest Configuration	426
31.4. Test Measurements And Results	430
31.5. Output Control	433
31.6. Recommended Practices	433
32. Static Code Analysis	435
32.1. clang-tidy	435
32.2. cppcheck	439
32.3. cpplint	441
32.4. Include What You Use	442
32.5. Potential Problems With Co-compilation	444
32.6. File Sets Header Verification	445
32.7. Disabling Checks For Some Files	446
32.8. Recommended Practices	446
33. Dynamic Code Analysis	448
33.1. Sanitizers	448
33.2. Code Coverage	455
33.3. Recommended Practices	463

V: Deployment And Dependencies **465**

34. Finding Things	466
34.1. Finding Files And Paths	466
34.2. Finding Programs	474
34.3. Finding Libraries	475
34.4. Finding Packages	477
34.5. Ignoring Search Paths	490

34.6. Debugging find_...() Calls	491
34.7. Recommended Practices	492
35. Installing	496
35.1. Directory Layout	497
35.2. Installing Project Targets	501
35.3. Installing Exports	513
35.4. Installing Imported Targets	517
35.5. Installing Files	518
35.6. Installing C++20 Modules	524
35.7. Custom Install Logic	526
35.8. Installing Dependencies	528
35.9. Writing A Config Package File	531
35.10. Executing An Install	542
35.11. Recommended Practices	542
36. Packaging	546
36.1. Packaging Basics	546
36.2. Components	551
36.3. Multi Configuration Packages	556
36.4. Recommended Practices	558
37. Package Generators	560
37.1. Simple Archives	561
37.2. Qt Installer Framework (IFW)	562
37.3. WIX	568
37.4. NSIS	570
37.5. Inno Setup	572
37.6. DragNDrop	574
37.7. productbuild	575
37.8. RPM	578
37.9. DEB	582
37.10. FreeBSD	583
37.11. Cygwin	583
37.12. NuGet	583
37.13. External	584
37.14. Recommended Practices	584
38. ExternalProject	586
38.1. High Level Overview	586
38.2. Directory Layout	588
38.3. Built-in Steps	589
38.4. Step Management	595
38.5. Miscellaneous Features	598
38.6. Common Issues	600

38.7. ExternalData	603
38.8. Recommended Practices	604
39. FetchContent	605
39.1. Comparison With ExternalProject	605
39.2. Basic Usage	606
39.3. Resolving Dependencies	607
39.4. Integration With find_package()	609
39.5. Developer Overrides	616
39.6. Other Uses For FetchContent	618
39.7. Restrictions	620
39.8. Recommended Practices	621
40. Making Projects Consumable	623
40.1. Use Project-specific Names	623
40.2. Don't Assume A Top Level Build	625
40.3. Avoid Hard-coding Developer Choices	626
40.4. Avoid Package Variables If Possible	627
40.5. Use Appropriate Methods To Obtain Dependencies	628
40.6. Recommended Practices	629
41. Dependency Providers	630
41.1. Top Level Setup Injection Point	630
41.2. Dependency Provider Implementation	631
41.3. Recommended Practices	638

VI: Project Organization **640**

42. Presets	641
42.1. High Level Structure	641
42.2. Configure Presets	643
42.3. Build Presets	651
42.4. Test Presets	652
42.5. Package Presets	653
42.6. Workflow Presets	654
42.7. Recommended Practices	656
43. Project Structure	658
43.1. Superbuild Structure	658
43.2. Non-superbuild Structure	660
43.3. Common Top Level Subdirectories	662
43.4. IDE Projects	663
43.5. Defining Targets	666
43.6. Windows-specific Issues	671
43.7. Cleaning Files	673
43.8. Re-running CMake On File Changes	674

43.9. Injecting Files Into Projects	674
43.10. Recommended Practices	675
VII: Special Topics	678
44. Working With Qt	679
44.1. Making Qt Available To The Project	679
44.2. Standard Project Setup	681
44.3. Command And Target Names	681
44.4. Defining Targets	682
44.5. Autogen	682
44.6. Translations	691
44.7. Deployment	692
44.8. Recommended Practices	697
Appendix A: Full Compiler Cache Example	700
Appendix B: Sanitizers Example	703
Appendix C: Timer Dependency Provider	706
Index	707

Preface

Back around 2016, I was surprised at the lack of published material for learning how to use CMake. The official reference documentation was a useful resource for those willing to go exploring, but as a way of learning CMake in a progressive, structured manner, it was not ideal. There were some wikis and personal websites that had some useful contents, but there were also many that contained out-of-date or questionable advice and examples. There was a distinct gap, which meant those new to CMake had a hard time learning good practices, leading to many becoming overwhelmed or frustrated.

At the time, I had been writing some blog articles to do something more productive with my spare time and to deepen my own technical knowledge around software development. I frequently wrote about areas that came up in my interaction with colleagues at work or in my own development activities, and I found this to be both rewarding and useful to others. As that pattern repeated itself, the idea of writing a book was born. Fast-forward two and a half years and the result is this book.

Along the way, there was a pivotal moment I now look back on with a degree of amusement. A colleague bemoaned a particular feature that he wished CMake had. It burrowed its way into my brain and sat there for a few months until one day I decided to explore how hard it would be to add that feature myself. That culminated in the test fixtures feature that is now a part of CMake. Importantly, I was really struck by the positive experience I had making that contribution. The people, the tools and the processes in place made working on the project truly a pleasure. From there, I became more deeply involved and now fulfill the role of volunteer co-maintainer.

I have since created my own company, through which I provide consulting services based around CMake, build and release processes, C++, continuous integration, and other related areas. I didn't set out with that goal, but it is now my primary focus. I consider it a privilege to be involved in such a diverse cross-section of projects, organizations and platforms. It gives me some unique perspectives, which I can then feed back into my ongoing activities maintaining CMake. My consulting activities are also a strong driver of the updates and new material for each new edition of this book. It is my hope that you, the reader, can then benefit from these experiences, and the developments that evolve from them.

Acknowledgments

It is only when you come to thank all those who have contributed to the process of getting your book released that you realize just how many people have been involved. A work like this doesn't happen without the generosity, patience, and insight of others. Nor does it succeed without being challenged, tested, and reworked. It relies on those who were kind enough to (sometimes unknowingly!) get involved in these activities as much as it does on the author. I cannot thank these people enough for their kindness and wisdom.

The CMake community wouldn't be as strong and as vibrant as it is today without the ongoing support of Kitware and its staff, past and present. I'd like to make special mention of Brad King, the CMake project leader, who through his inclusive and encouraging approach to new CMake contributors has made people like myself very welcome and feel empowered to get involved. I have personally learned much from him just by observing the way he interacts with developers and users, providing strong leadership, and fostering an environment of respect for others. It also goes without saying that the numerous contributors to CMake over the years also deserve much praise for their efforts, often made on a purely voluntary basis. I'm humbled by the scale of contributions that have been made by so many, and by the positive impact on the world of software development.

A special mention is also much deserved for my past colleague, Mike Wake. Much of the material in the early editions of this book was thrashed out and tested in real, actively developed production projects. There were wrong turns, and many technical discussions on how to improve things from both a usability and a robustness perspective. His support in giving the space and encouragement to work through these things, and his willingness to wear some short-term (and sometimes not-so-short-term) pain were an instrumental part of distilling many processes and techniques down to what works in practice. I am also very grateful for the timely words of advice and encouragement delivered at just the right time during some of the more stressful and exhausting periods around those earlier editions.

Across the various editions, a number of people have generously agreed to review material in this book. Without these people, the book's technical accuracy and its readability would have suffered. Any remaining errors and deficiencies are my own. Fellow CMake developers Gregor Jasny and Christian Pfeiffer were valuable contributors throughout the review process for the first publication. I am truly grateful for their suggestions and insights. Thanks also to Nils Gladitz for his input, especially at such short notice before the first publication. I'd also like to thank my past colleagues, Matt Bolger and Lachlan Hetheron, both of whom provided constructive feedback and reminded me of the importance of a fresh set of eyes before the first release.

Many of the people who have generously reviewed chapters for subsequent editions are also CMake contributors or maintainers, and I am grateful for both their time reviewing this book, and in improving CMake itself. The same can be said for readers who have also provided corrections and suggestions for improvement. I'd like to acknowledge the following people who have provided reviews, corrections, or feedback that led to updates for one or more editions after the first publication:

Adriaan de Groot	Jesse Bollinger
Albert Neumüller	Jörg Bornemann
Alex Turbov	Lars Bilke
Alexandru Croitor	Lieven de Cock
Artur Ryt	Luis Caro Campos
Bo Rydberg	Luis Díaz Más
Burkhard Stubert	Lukas Oyen
Caleb Huitt	Marc Chevrier
Christopher McArthur	Mateusz Pusz
Cristian Adam	Michael Platings
Cristian Morales Vega	Nagy-Egri Máté Ferenc
Damon McDougall	Parker Coates
Declan Moran	Patrick Heyer
Elias Daler	Sebastian Holtermann
Francesco Giacomini	Tobias Hunger
Ganesh A. Hegde	Tristano Ajmone
Hidayat Rzaev	Vilas Chitrakaran
Jacek Galowicz	

I would also like to express my gratitude to the people behind AsciiDoctor, the software used to compile and prepare this book. Despite the size, complexity and technical nature of the material, I have been constantly amazed at how it has made self-publishing not just a viable option, but also an enjoyable experience with surprisingly few practical limitations. The path from author to reader is now so much shorter and simpler than when I initially began work on this book. Thanks for the awesome tool!

The book's cover and some supporting material on the website are the result of a better eye and understanding of graphical design than my own. To my friend and designer, V, the way you somehow managed to make sense of my random, disconnected ideas and conflicting snippets still baffles me. I don't understand how you do it, but I do like the result!

In every book's acknowledgments section, the author invariably thanks family members and spouses, and there's good reason for that. It takes a huge amount of understanding and sacrifice to tolerate your tiredness, your unavailability to do many of the things ordinary people get to do, and your unreasonable decision to devote more time to a project than to them. I truly cannot express the depth of my gratitude to my wife for the way she has managed to be so supportive and patient throughout the process of getting this book written, published and regularly updated. I am indeed a very fortunate human being.

Part I: Getting Started

This first part of the book covers the most basic aspects of CMake. It will equip a developer who has never used CMake before with the necessary skills to be able to create, build, test, install, and package a simple executable or library. Rather than going into too much detail, these early chapters focus on getting a working project and introducing the basic commands needed to interact with it. It also establishes the importance of *targets*, one of the most fundamental concepts in CMake.

Attempting to use any tool before understanding at least the basics of what it does and how it is meant to be used is likely to result in much frustration. On the other hand, spending all one's time learning the theory about something without getting hands-on makes for a rather boring experience and often leads to an overly idealistic understanding. With that in mind, the reader is encouraged to try out the material in each chapter as they go. By the end of this part of the book, the reader should have a basic project they can use as a solid starting point for exploring the material in later chapters.

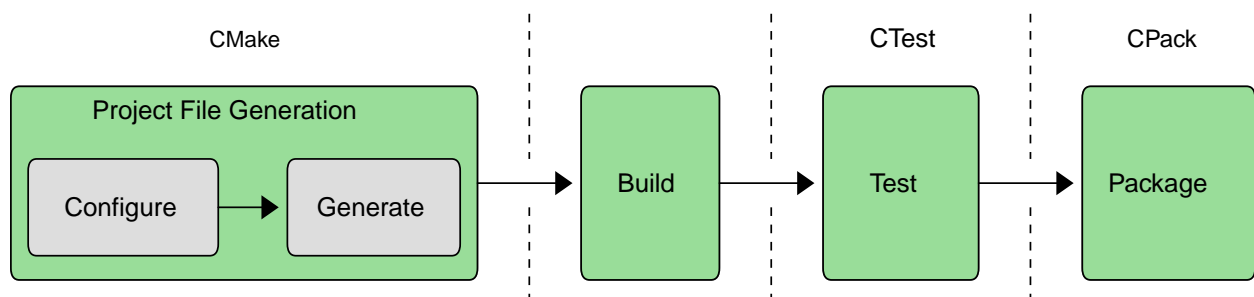
After completing this part of the book, the reader should continue on to the [Part II, “Fundamentals”](#) chapters. They go into more detail about key concepts and features that the rest of the book will rely on heavily.

Chapter 1. Introduction

Whether a seasoned developer or just starting out in a software career, one cannot avoid the process of becoming familiar with a range of tools in order to turn a project's source code into something an end user can actually use. Compilers, linkers, testing frameworks, packaging systems, and more all contribute to the complexity of deploying high quality, robust software. While some platforms have a dominant IDE environment that simplifies some aspects of this (e.g. Xcode and Visual Studio), projects that need to support multiple platforms cannot always make use of their features. Having to support multiple platforms adds more complications that can affect everything from the set of available tools through to the different capabilities available and restrictions enforced. A typical developer could be forgiven for losing at least some of their sanity trying to keep on top of the whole picture.

Fortunately, there are tools that make taming the process more manageable. CMake is one such tool, or more accurately, CMake is a suite of tools which covers everything from setting up a build right through to producing packages ready for distribution. Not only does it cover the process from start to end, it also supports a wide range of platforms, tools, and languages.

When working with CMake, it helps to understand its view of the world. Loosely speaking, the start to end process according to CMake looks something like this:



The first stage takes a generic project description and generates platform-specific project files suitable for use with the developer's regular build tool of choice (make, Xcode, Visual Studio, etc.). While this setup stage is what CMake is best known for, the CMake suite of tools also includes CTest and CPack. These manage the testing and packaging stages respectively. The entire process from start to finish can be driven from CMake itself, and each stage can also be executed individually through CMake. This abstracts away platform differences, making some tasks much simpler.

Before jumping in and getting their hands dirty with CMake, developers will first need to ensure CMake is installed on their system. Some platforms may typically come with CMake already installed (most Linux distributions have CMake available through their package manager), but these versions are often quite old. Where possible, it is recommended that developers work with a recent CMake release. This is particularly true when developing for Apple platforms, where tools like Xcode and its SDKs change rapidly, and where app store requirements evolve over time. The official CMake packages can be downloaded and unpacked to a directory on the developer's machine without interfering with any system-wide CMake installation. Developers are encouraged to take advantage of this and remain relatively close to the most recent stable CMake release.

These days, CMake also comes with fairly extensive [reference documentation](#), which is accessible from the official CMake site. This useful resource is very helpful for looking up the various commands, options, keywords, and so on. Developers will likely want to bookmark it for quick reference. The [CMake forum](#) is also a great source of advice and is the recommended place for asking CMake-related questions where the documentation doesn't provide sufficient guidance.

Chapter 2. Setting Up A Project

Without a build system, a project is just a collection of files. CMake brings some order to this, starting with a human-readable file called `CMakeLists.txt` that defines what should be built and how, what tests to run, and what package(s) to create. This file is a platform-independent description of the whole project, which CMake then turns into platform-specific build tool project files. As its name suggests, it is just an ordinary text file which developers edit in their favorite text editor or development environment. The contents of this file are covered in great detail in subsequent chapters, but for now, it is enough to know that this is what controls everything that CMake will do in setting up and performing the build.

A fundamental part of CMake is the concept of a project having both a source directory and a binary directory. The source directory is where the `CMakeLists.txt` file is located, and the project's source files and all other files needed for the build are organized under that location. The source directory is frequently under version control with a tool like git, subversion, or similar.

The binary directory is where everything produced by the build is created. It is often also called the build directory. For reasons that will become clear in later chapters, CMake generally uses the term *binary* directory, but among developers, the term build directory tends to be in more common use. This book tends to prefer the latter term since it is generally more intuitive. CMake, the chosen build tool (make, Visual Studio, etc.), CTest, and CPack will all create various files within the build directory and subdirectories below it. Executables, libraries, test output, and packages are all created within the build directory. CMake also creates a special file called `CMakeCache.txt` in the build directory to store various information for reuse on subsequent runs. Developers won't normally need to concern themselves with the `CMakeCache.txt` file, but later chapters will discuss situations where this file is relevant. The build tool's project files (Xcode or Visual Studio project files, Makefiles, etc.) are also created in the build directory, and those project files are not intended to be put under version control. The `CMakeLists.txt` files are the canonical description of the project, and the generated project files should be considered part of the build output.

When a developer commences work on a project, they must decide where they want their build directory to be in relation to their source directory. There are essentially two approaches: *in-source* and *out-of-source* builds.

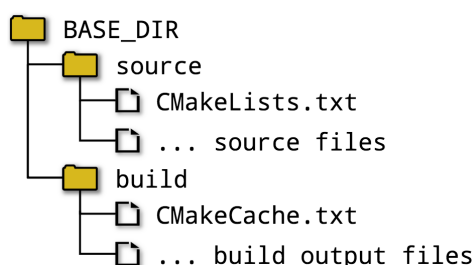
2.1. In-source Builds

It is possible, though discouraged, for the source and build directories to be the same. This arrangement is called an *in-source* build. Developers at the beginning of their career often start out using this approach because of the perceived simplicity. A significant drawback to in-source builds is that all the build outputs are intermixed with the source files. This lack of separation causes directories to become cluttered with all sorts of files and subdirectories, making it harder to manage the project sources and running the risk of build outputs overwriting source files. It also makes working with version control systems more difficult, since there are lots of files created by the build which the source control tool has to know to ignore, or the developer has to manually exclude during commits. One other drawback to in-source builds is that it can be non-trivial to clear out all build output and start again with a clean source tree. For these reasons, developers are discouraged from using in-source builds where possible, even for simple projects.

2.2. Out-of-source Builds

The more preferable arrangement is for the source and build directories to be different, which is called an *out-of-source* build. This keeps the sources and the build outputs completely separate from each other, thus avoiding the intermixing problems experienced with in-source builds. Out-of-source builds also have the advantage that the developer can create multiple build directories for the same source directory. This allows builds to be set up with different sets of options, such as debug and release versions.

This book always uses out-of-source builds. It also generally follows the pattern of the source and build directories being under a common parent. The build directory will be called *build*, or some variation thereof. For example:



A variation on this is to make the build directory a subdirectory of the source directory. This offers most of the advantages of an out-of-source build, but isn't as isolated from the sources. Some continuous integration systems more or less require this structure due to how they limit access outside the source directory. IDEs also often use such an arrangement by default. If the build directory is placed under the source directory, prefer to give the build directory a name starting with *build*. This minimizes the chance of it clashing with a source directory, and it makes it easy to set up source repository ignore rules for all build directories with a simple wildcard pattern.

2.3. Generating Project Files

Once the choice of directory structure has been made, the developer runs CMake, which reads in the `CMakeLists.txt` file and creates project files in the build directory. The developer selects the type of project file to be created by choosing a particular project file *generator*. A range of different generators are supported, with the more commonly used ones listed in the table below.

Category	Generator Examples	Multi-config
Visual Studio	Visual Studio 17 2022	Yes
	Visual Studio 16 2019	
Xcode	Xcode	Yes
Ninja	Ninja	No
	Ninja Multi-Config	Yes
Makefiles	Unix Makefiles	No
	NMake Makefiles	

Some generators produce projects which support multiple configurations (Debug, Release, and so on). These allow the developer to choose between different build configurations without having to re-run CMake. This is more convenient for generators that create projects for use in IDE environments like Xcode and Visual Studio. For generators which do not support multiple configurations, the developer has to re-run CMake to switch the build between Debug, Release, etc. These are simpler, and they often have good support in IDE environments not so closely associated with a particular compiler (CLion, Qt Creator, KDevelop, etc.).

The most basic way to run CMake is via the `cmake` command line utility. The simplest way to invoke it is to change to the build directory and pass options to `cmake` for the generator type and location of the source tree. For example:

```
mkdir build
cd build
cmake -G "Unix Makefiles" ../source
```

If the `-G` option is omitted, CMake will choose a default generator type based on the host platform. If using CMake 3.15 or later, this default can be overridden by setting the `CMAKE_GENERATOR` environment variable to the desired default instead. For all generator types, CMake will carry out a series of tests to determine how to set up the project files. This includes things like verifying that the compilers work, determining the set of supported compiler features, and various other tasks. A variety of information will be logged before CMake finishes with lines like the following upon success:

```
-- Configuring done
-- Generating done
-- Build files have been written to: /some/path/build
```

The above highlights that project file creation actually involves two steps; configuring and generating. During the *configuring* phase, CMake reads in the `CMakeLists.txt` file and builds up an internal representation of the entire project. After this is done, the *generation* phase creates the project files. The distinction between configuring and generating doesn't matter so much for basic CMake usage, but in later chapters, the separation of configuration and generation becomes important. This is covered in more detail in [Chapter 11, Generator Expressions](#).

When CMake has completed its run, it will have saved a `CMakeCache.txt` file in the build directory. CMake uses this file to save details so that if it is run again, it can re-use information computed the first time and speed up the project generation. As covered in later chapters, it also allows developer options to be saved between runs. A GUI application, `cmake-gui`, is available as an alternative to running the `cmake` command line tool. Discussion of that GUI application is deferred to [Chapter 6, Variables](#) where its usefulness is more clearly evident.

2.4. Running The Build Tool

At this point, with project files now available, the developer can use their selected build tool in the way to which they are accustomed. The build directory will contain the necessary project files, which can be loaded into an IDE and read by command line tools. Alternatively, `cmake` can invoke the build tool on the developer's behalf:

```
cmake --build /pathTo/build --config Debug --target MyApp
```

This works even for project types the developer may be more accustomed to using through an IDE like Xcode or Visual Studio. The `--build` option points to the build directory used by the CMake project generation step. For multi-configuration generators, the `--config` option specifies which configuration to build, whereas single-configuration generators will ignore the `--config` option and rely instead on information provided when the CMake project generation step was performed. Specifying the build configuration is covered in depth in [Chapter 15, Build Type](#). The `--target` option can be used to tell the build tool what to build, or if omitted, the default target will be built. With CMake 3.15 or later, multiple targets can be listed after the `--target` option, separated by spaces.

While developers will typically invoke their selected build tool directly in day-to-day development, invoking it via the `cmake` command as shown above can be more useful in scripts driving an automated build. Using this approach, a simple scripted build might look something like this:

```
mkdir build
cd build
cmake -G "Unix Makefiles" ../source
cmake --build . --config Release --target MyApp
```

If the developer wishes to experiment with different generators, the argument given to the `cmake -G` option is the only thing that needs to be changed. The correct build tool will be automatically invoked. The build tool doesn't even have to be on the user's `PATH` for `cmake --build` to work, although it may need to be for the initial configuration step when `cmake` is first invoked.

2.5. Recommended Practices

Even when first starting out using CMake, it is advisable to make a habit of keeping the build directory completely separate from the source tree. A good way to get early experience of the benefits of such an arrangement is to set up two or more different builds for the same source directory. One build could be configured with Debug settings, the other for a Release build. Another option is to use different project generators for the different build directories, such as Unix Makefiles and Xcode. This can help to catch any unintended dependencies on a particular build tool, or to check for differing compiler settings between generator types.

It can be tempting to focus on using one particular type of project generator in the early stages of a project, especially if the developer is not accustomed to writing cross-platform software. Projects do, however, have a habit of growing beyond their initial scope, and it is relatively common to need to support additional platforms and generator types. Periodically checking the build with a different project generator than the one a developer usually uses can save considerable future pain by discouraging generator-specific code where it isn't required. This also has the benefit of making the project well-placed to take advantage of any new generator type in the future. A good strategy is to ensure the project builds with the default generator type on each platform of interest, plus one other type. The Ninja generator is an excellent choice for the latter, since it has the broadest platform support of all the generators, and it also creates very efficient builds. If the project is being scripted, invoke the build tool via `cmake --build` instead of invoking the build tool directly. This allows the script to easily switch between generator types without having to be modified.

Chapter 3. A Minimal Project

All CMake projects start with a file called `CMakeLists.txt`, which is required to be at the top of the source tree. Think of it as the CMake project file, defining everything about the build, from sources and targets through to testing, packaging, and other custom tasks. It can be as simple as a few lines, or it can be quite complex and pull in more files from other directories. `CMakeLists.txt` is just an ordinary text file and is usually edited directly, just like any other source file in the project.

Continuing the analogy with sources, CMake defines its own language which has many things a programmer would be familiar with, such as variables, functions, macros, conditional logic, looping, code comments, and so on. These various concepts and features are covered in [Part II, “Fundamentals”](#), but for now, the goal is just to get a simple build working as a starting point. The following is a minimal, well-formed `CMakeLists.txt` file that defines a basic executable.

```
cmake_minimum_required(VERSION 3.2)
project(MyApp)
add_executable(MyExe main.cpp)
```

Each line in the above example executes a built-in CMake *command*. In CMake, commands are similar to other languages’ function calls, except that while they support arguments, they do not return values directly (but a later chapter shows how to pass values back to the caller in other ways). Arguments are separated from each other by spaces and may be split across multiple lines:

```
add_executable(MyExe
    main.cpp
    src1.cpp
    src2.cpp
)
```

Command names are also case-insensitive, so the following are all equivalent:

```
add_executable(MyExe main.cpp)
ADD_EXECUTABLE(MyExe main.cpp)
Add_Executable(MyExe main.cpp)
```

Typical style varies, but the accepted convention these days is to use all lowercase for command names. This is also the convention followed by the CMake documentation for built-in commands.

3.1. Managing CMake Versions

CMake is continually updated and extended to add support for new tools, platforms and features. The developers behind CMake are very careful to maintain backwards compatibility with each new release, so when users update to a newer version of CMake, projects should continue to build as they did before. Sometimes, a particular CMake behavior needs to change, or more stringent checks and warnings may be introduced in newer versions. Rather than requiring all projects to immediately deal with this, CMake provides *policy* mechanisms which allow the project to say

“Behave like CMake version X.Y.Z”. This allows CMake to fix bugs internally and introduce new features, but still maintain the expected behavior of any particular past release.

The primary way a project specifies details about its expected CMake version behavior is with the `cmake_minimum_required()` command. This should always be the first line of the `CMakeLists.txt` file. It ensures that the project’s requirements are checked and established before anything else. This command does two things:

- It specifies the minimum version of CMake the project needs. If the `CMakeLists.txt` file is processed with a CMake version older than the one specified, it will halt immediately with an error. This ensures that a particular minimum set of CMake functionality is available before proceeding.
- It enforces policy settings to match CMake behavior to the specified version.

Using this command is so important that CMake 3.26 and later will issue a warning if the top level `CMakeLists.txt` file does not call `cmake_minimum_required()` before calling `project()`. CMake needs to know how to set up the policy behavior for all subsequent processing. For most projects, it is enough to treat `cmake_minimum_required()` as simply specifying the minimum required CMake version, as its name suggests. The fact that it also implies CMake should behave the same as that particular version can be considered a useful side benefit. [Chapter 13, Policies](#) discusses policy settings in more detail and explains how to tailor this behavior as needed.

The typical form of the `cmake_minimum_required()` command is straightforward:

```
cmake_minimum_required(VERSION major.minor[.patch[.tweak]])
```

The `VERSION` keyword must always be present, and the version details provided must have at least the `major.minor` part. In most projects, specifying the patch and tweak parts is not necessary, since new features typically only appear in minor version updates (this is the official CMake behavior from version 3.0 onward). Only if a specific bug fix is needed should a project specify a patch part. Furthermore, since no CMake release in the 3.x series has used a tweak number, projects should not need to specify one either.

Developers should think carefully about what minimum CMake version their project should require. Version 3.5 is the absolute oldest any new project should consider. Anything older will trigger deprecation warnings if the developer uses the latest CMake version on the project. If working with fast-moving platforms such as iOS, quite recent versions of CMake may be needed to support the latest OS and Xcode releases.

As a general rule of thumb, choose the most recent CMake version that won’t present significant problems for those building the project. The greatest difficulty is typically experienced by projects that need to support older platforms where the system-provided version of CMake may be quite old. For such cases, if at all possible, developers should consider installing a more recent release rather than restricting themselves to very old CMake versions. On the other hand, if the project will itself be a dependency for other projects, then choosing a more recent CMake version may present a hurdle for adoption. In such cases, it may be beneficial to instead require the oldest CMake version that still provides the minimum CMake features needed, but make use of features from later CMake versions if available ([Chapter 13, Policies](#) presents techniques for achieving this). This

will prevent other projects from being forced to require a more recent version than their target environment typically allows or provides. Dependent projects can always require a more recent version if they so wish, but they cannot require an older one. The main disadvantage of using the oldest workable version is that it may result in more deprecation warnings, since newer CMake versions will warn about older behaviors to encourage maintainers to update their project.

3.2. The `project()` Command

Every CMake project should contain a `project()` command, and it should appear after `cmake_minimum_required()` has been called. The command with its most common options has the following form:

```
project(projectName
    [VERSION major[.minor[.patch[.tweak]]]]
    [LANGUAGES languageName ...]
)
```

The `projectName` is required and may only contain letters, numbers, underscores (`_`) and hyphens (`-`), although typically only letters and perhaps underscores are used in practice. Since spaces are not permitted, the project name does not have to be surrounded by quotes. This name is used for the top level of a project with some project generators (Xcode and Visual Studio). It is also used in various other parts of the project, such as to act as defaults for packaging and documentation metadata, and to provide project-specific variables. The name is the only mandatory argument for the `project()` command.

The optional `VERSION` details are only supported in CMake 3.0 and later. Like the `projectName`, the version details are used by CMake to populate some variables and as default package metadata, but other than that, the version details don't have any other significance. Nonetheless, a good habit to establish is to define the project's version here so that other parts of the project can refer to it. [Chapter 22, *Specifying Version Details*](#) covers this in depth and explains how to refer to this version information later in the `CMakeLists.txt` file.

The optional `LANGUAGES` argument defines the programming languages that should be enabled for the project. Supported values include `C`, `CXX`, `Fortran`, `ASM`, `CUDA`, and others, depending on the CMake version. If specifying multiple languages, separate each with a space. In some special situations, projects may want to indicate that no languages are used, which can be done using `LANGUAGES NONE`. Techniques introduced in later chapters take advantage of this particular form. If no `LANGUAGES` option is provided, CMake will default to `C` and `CXX`. CMake versions prior to 3.0 do not support the `LANGUAGES` keyword, but languages can still be specified after the project name using the older form of the command like so:

```
project(MyProj C CXX)
```

New projects are encouraged to specify a minimum CMake version of at least 3.0 and use the new form with the `LANGUAGES` keyword instead.

The `project()` command does much more than populate a few variables. One of its important responsibilities is to check the compilers for each enabled language and ensure they are able to compile and link successfully. Problems with the compiler and linker setup are then caught very early. Once these checks have passed, CMake sets up a number of variables and properties which control the build for the enabled languages. [Chapter 24, Toolchains And Cross Compiling](#) discusses this area in much greater detail, including the various ways to influence toolchain selection and configuration. [Chapter 8, Using Subdirectories](#) also discusses additional considerations and requirements that affect the use of the `project()` command.

When the compiler and linker checks performed by CMake are successful, their results are cached so that they do not have to be repeated in subsequent CMake runs. These cached details are stored in the `CMakeCache.txt` file in build directory. Additional details about the checks can be found in subdirectories within the build area, but developers would typically only need to look there if working with a new or unusual compiler, or when setting up toolchain files for cross-compiling.

3.3. Building A Basic Executable

To complete our minimal example, the `add_executable()` command tells CMake to create an executable from a set of source files. The basic form of this command is:

```
add_executable(targetName source1 [source2 ...])
```

This creates an executable which can be referred to within the CMake project as `targetName`. This name may contain letters, numbers, underscores and hyphens. When the project is built, an executable will be created in the build directory with a platform-dependent name, the default name being based on the target name. Consider the following simple example command:

```
add_executable(MyApp main.cpp)
```

By default, the name of the executable would be `MyApp.exe` on Windows and `MyApp` on Unix-based platforms like macOS, Linux, etc. The executable name can be customized with target properties, a CMake feature introduced in [Chapter 10, Properties](#). Multiple executables can also be defined within the one `CMakeLists.txt` file by calling `add_executable()` multiple times with different target names. If the same target name is used in more than one `add_executable()` command, CMake will fail and highlight the error.

3.4. Commenting

Before leaving this chapter, it is useful to demonstrate how to add comments to a `CMakeLists.txt` file. Comments are used extensively throughout this book, and developers are encouraged to also get into the habit of commenting their projects just as they would for ordinary source code.

CMake follows commenting conventions similar to Unix shell scripts. Any line beginning with a `#` character is treated as a comment. Except within a quoted string, anything after a `#` on a line within a `CMakeLists.txt` file is also treated as a comment.

CMake 3.0 also added support for lua-style block comments. The start of the block comment has the form `#[==[` where there can be any number of `=` characters between the square brackets, including none. Everything up until a matching `]==]` is treated as a comment, where the number of `=` characters must match the number at the start of the comment. This can be a useful way to temporarily comment out a block of code.

The following shows a few comment examples and brings together the concepts introduced in this chapter:

```
cmake_minimum_required(VERSION 3.2)

# We don't use the C++ compiler, so don't let project()
# test for it in case the platform doesn't have one
project(MyApp VERSION 4.7.2 LANGUAGES C)

# Primary tool for this project
add_executable(MainTool
    main.c
    debug.c # Optimized away for release builds
)

# Helpful diagnostic tool for development and testing
add_executable(TestTool testTool.c)

# These tools are not ready yet, disable them
#[=[
add_executable(NewTool1 tool1.c)
add_executable(NewTool2
    tool2.c
    extras.c
)
]=]
```

3.5. Recommended Practices

Ensure every CMake project has a `cmake_minimum_required()` command as the first line of its top level `CMakeLists.txt` file. When deciding the minimum required version number to specify, keep in mind that later versions will give more freedom in using newer CMake features. It will also mean the project is likely to be better placed to adapt to new platform or operating system releases, which inevitably introduce new things for build systems to deal with. Conversely, if the project is intended to be built and distributed as part of the operating system (common for Linux), the minimum CMake version is likely to be dictated by the version of CMake provided by that same distribution.

If the project can require CMake 3.0 or later, it is also good to force thinking about project version numbers early and start incorporating version numbering into the `project()` command as soon as possible. It can be very hard to overcome the inertia of existing processes and change how version numbers are handled later in the life of a project. Consider popular practices such as [Semantic Versioning](#) when deciding on a versioning strategy.

Chapter 4. Building Simple Targets

As shown in the previous chapter, it is relatively straightforward to define a simple executable in CMake. The simple example given previously required defining a target name for the executable and listing the source files to be compiled:

```
add_executable(MyApp main.cpp)
```

This assumes the developer wants a basic console executable to be built. But CMake also allows the developer to define other types of executables, such as app bundles on Apple platforms, and Windows GUI applications. This chapter discusses additional options which can be given to `add_executable()` to specify these details.

In addition to executables, developers also frequently need to build and link libraries. CMake supports a few different kinds of libraries, including static, shared, modules, and frameworks. CMake also offers very powerful features for managing dependencies between targets and how libraries are linked. This whole area of libraries and how to work with them in CMake forms the bulk of this chapter. The concepts covered here are used extensively throughout this book. Some very basic use of variables and properties are also given to provide a flavor for how these CMake features relate to libraries and targets in general.

4.1. Executables

The more complete form of the basic `add_executable()` command is as follows:

```
add_executable(targetName [WIN32] [MACOSX_BUNDLE]
                    [EXCLUDE_FROM_ALL]
                    source1 [source2 ...]
)
```

The only differences to the form shown previously are the new optional keywords.

WIN32

When building the executable on a Windows platform, this option instructs CMake to build the executable as a Windows GUI application. In practice, this means it will be created with a `WinMain()` entry point instead of just `main()`, and it will be linked with the `/SUBSYSTEM:WINDOWS` option. On all other platforms, the `WIN32` option is ignored.

MACOSX_BUNDLE

When present, this option directs CMake to build an app bundle when building on an Apple platform. Contrary to what the option name suggests, it applies not just to macOS, but also to other Apple platforms like iOS as well. The exact effects of this option vary somewhat between platforms. For example, on macOS, the app bundle layout has a very specific directory structure, whereas on iOS, the directory structure is flattened. CMake will also generate a basic `Info.plist` file for bundles. These and other details are covered in more detail in [Section 25.2, “Application Bundles”](#). On non-Apple platforms, the `MACOSX_BUNDLE` keyword is ignored.

EXCLUDE_FROM_ALL

Sometimes, a project defines a number of targets, but by default only some of them should be built. When no target is specified at build time, the default ALL target is built. Depending on the CMake generator used, the name may be slightly different, such as ALL_BUILD for Xcode, or all when using one of the Makefiles or Ninja generators. If an executable is defined with the EXCLUDE_FROM_ALL option, it will not be included in that default ALL target. The executable will then only be built if it is explicitly requested by the build command, or if it is a dependency for another target that is part of the default ALL build. A common situation where it can be useful to exclude a target from ALL is where the executable is a developer tool that is only needed occasionally.

In addition to the above, there are other forms of the `add_executable()` command which produce a kind of reference to an existing executable or target rather than defining a new one to be built. These alias executables are covered in detail in [Chapter 19, Target Types](#).

4.2. Defining Libraries

Creating simple executables is a fundamental need of any build system. For many larger projects, the ability to create and work with libraries is also essential to keep the project manageable. CMake supports building a variety of different kinds of libraries, taking care of many of the platform differences, but still supporting the native idiosyncrasies of each. Library targets are defined using the `add_library()` command, of which there are a number of forms. The most basic of these is the following:

```
add_library(targetName [STATIC | SHARED | MODULE]
    [EXCLUDE_FROM_ALL]
    source1 [source2 ...]
)
```

This form is analogous to how `add_executable()` is used to define a simple executable. The `targetName` is used within the `CMakeLists.txt` file to refer to the library, with the name of the built library on the file system being derived from this name by default. The `EXCLUDE_FROM_ALL` keyword has exactly the same effect as it does for `add_executable()`, namely to prevent the library from being included in the default ALL target. The type of library to be built is specified by one of the remaining three keywords `STATIC`, `SHARED`, or `MODULE`.

STATIC

This specifies a static library or archive. On Windows, the default library name would be `targetName.lib`, while on Unix-like platforms, it would typically be `libtargetName.a`.

SHARED

This specifies a shared or dynamically linked library. On Windows, the default library name would be `targetName.dll`, on Apple platforms it would be `libtargetName.dylib`, and on other Unix-like platforms it would typically be `libtargetName.so`. On Apple platforms, shared libraries can also be marked as frameworks, a topic covered in [Section 25.3, “Frameworks”](#).

MODULE

Specifies a library that is somewhat like a shared library, but is intended to be loaded dynamically at run-time rather than being linked directly to a library or executable. These are typically plugins or optional components the user may choose to have loaded or not. On Windows platforms, no import library is created for the DLL.

It is possible to omit the keyword defining what type of library to build. Unless the project specifically requires a particular type of library, the preferred practice is to not specify it and leave the choice up to the developer when building the project. In such cases, the library will be either `STATIC` or `SHARED`, with the choice determined by the value of a CMake variable called `BUILD_SHARED_LIBS`. If `BUILD_SHARED_LIBS` has been set to `true`, the library target will be a shared library, otherwise it will be static. Working with variables is covered in detail in [Chapter 6, Variables](#), but for now, one way to set this variable is by including a `-D` option on the `cmake` command line like so:

```
cmake -DBUILD_SHARED_LIBS=YES /path/to/source
```

It could be set in the `CMakeLists.txt` file instead with the following placed before any `add_library()` commands, but that would then require developers to modify it if they wanted to change it (i.e. it would be less flexible):

```
set(BUILD_SHARED_LIBS YES)
```

Just as for executables, library targets can also be defined to refer to some existing binary or target rather than being built by the project. Another type of pseudo-library is also supported for collecting together object files without going as far as creating a static library. These are all discussed in detail in [Chapter 19, Target Types](#).

4.3. Linking Targets

When considering the targets that make up a project, developers are typically used to thinking in terms of library A needing library B, so A is linked to B. This is the traditional way of looking at library handling, where the idea of one library needing another is very simplistic. In reality, there are a few different types of dependency relationships that can exist between libraries:

PRIVATE

Private dependencies specify that library A uses library B in its own internal implementation. Anything else that links to library A doesn't need to know about B because it is an internal implementation detail of A.

PUBLIC

Public dependencies specify that not only does library A use library B internally, it also uses B in its interface. This means that A cannot be used without B, so anything that uses A will also have a direct dependency on B. An example of this would be a function defined in library A which has at least one parameter of a type defined and implemented in library B, so code cannot call the function from A without providing a parameter whose type comes from B.

INTERFACE

Interface dependencies specify that in order to use library A, parts of library B must also be used. This differs from a public dependency in that library A doesn't require B internally, it only uses B in its interface. An example of where this is useful is when working with library targets defined using the `INTERFACE` form of `add_library()`, such as when using a target to represent a header-only library's dependencies (see [Section 19.2.4, “Interface Libraries”](#)).

CMake captures this richer set of dependency relationships with its `target_link_libraries()` command, not just the simplistic idea of needing to link. The general form of the command is:

```
target_link_libraries(targetName
    <PRIVATE|PUBLIC|INTERFACE> item1 [item2 ...]
    [<PRIVATE|PUBLIC|INTERFACE> item3 [item4 ...]]
    ...
)
```

This allows projects to precisely define how one library depends on others. CMake then takes care of managing the dependencies throughout the chain of libraries linked in this fashion. For example, consider the following:

```
add_library(Collector src1.cpp)
add_library(Algo src2.cpp)
add_library(Engine src3.cpp)
add_library(Ui src4.cpp)
add_executable(MyApp main.cpp)

target_link_libraries(Collector
    PUBLIC Ui
    PRIVATE Algo Engine
)
target_link_libraries(MyApp PRIVATE Collector)
```

In this example, the `Ui` library is linked to the `Collector` library as `PUBLIC`, so even though `MyApp` only directly links to `Collector`, `MyApp` will also be linked to `Ui` because of that `PUBLIC` relationship. The `Algo` and `Engine` libraries, on the other hand, are linked to `Collector` as `PRIVATE`, so `MyApp` will not be directly linked to them. [Section 19.2, “Libraries”](#) discusses additional behaviors for static libraries which may result in further linking to satisfy dependency relationships, including cyclic dependencies.

Later chapters present a few other `target_...()` commands which further enhance the dependency information carried between targets. These allow compiler/linker flags and header search paths to also carry through from one target to another when they are connected by `target_link_libraries()`. These features were added progressively from CMake 2.8.11 through to 3.2, and they lead to considerably simpler and more robust `CMakeLists.txt` files.

Later chapters also discuss the use of more complex source directory hierarchies. In such cases, if using CMake 3.12 or earlier, the `targetName` used with `target_link_libraries()` must have been defined by an `add_executable()` or `add_library()` command in the same directory from which `target_link_libraries()` is being called (this restriction was removed in CMake 3.13).

4.4. Linking Non-targets

In the preceding section, all the items being linked to were existing CMake targets, but the `target_link_libraries()` command is more flexible than that. In addition to CMake targets, the following things can also be specified as items in a `target_link_libraries()` command:

Full path to a library file

CMake will add the library file to the linker command. If the library file changes, CMake will detect that change and re-link the target. Note that from CMake version 3.3, the linker command always uses the full path specified, but prior to version 3.3, there were some situations where CMake may ask the linker to search for the library instead (e.g. replace `/usr/lib/libfoo.so` with `-lfoo`). The reasoning and details of the pre-3.3 behavior are non-trivial and are largely historical, but for the interested reader, the full set of information is available in the CMake documentation under the `CMP0060` policy.

Plain library name

If just the name of the library is given with no path, the linker command will search for that library (e.g. `foo` becomes `-lfoo` or `foo.lib`, depending on the platform). This would be common for libraries provided by the system.

Link flag

As a special case, items starting with a hyphen other than `-l` or `-framework` will be treated as flags to be added to the linker command. The CMake documentation warns that these should only be used for `PRIVATE` items, since they would be carried through to other targets if defined as `PUBLIC` or `INTERFACE`, and this may not always be safe.

4.5. Old-style CMake

For historical reasons, any link item specified in `target_link_libraries()` may be preceded by one of the keywords `debug`, `optimized`, or `general`. The effect of these keywords is to further refine when the item following it should be included based on whether the build is configured as a debug build (see [Chapter 15, Build Type](#)). If an item is preceded by the `debug` keyword, then it will only be added if the build is a debug build. If an item is preceded by the `optimized` keyword, it will only be added if the build is not a debug build. The `general` keyword specifies that the item should be added for all build configurations, which is the default behavior anyway if no keyword is used. The `debug`, `optimized` and `general` keywords should be avoided for new projects, as there are clearer, more flexible, and more robust ways to achieve the same thing with today's CMake features.

The `target_link_libraries()` command also has a few other forms, some of which have been part of CMake from well before version 2.8.11. These forms are discussed here for the benefit of understanding older CMake projects, but their use is generally discouraged for new projects. The full form shown previously with `PRIVATE`, `PUBLIC` and `INTERFACE` sections should be preferred, as it expresses the nature of dependencies with more accuracy.

```
target_link_libraries(targetName item [item...])
```

The above form is generally equivalent to the items being defined as `PUBLIC`, but in certain situations, they may instead be treated as `PRIVATE`. In particular, if a project defines a chain of library dependencies with a mix of old and new command forms, the old-style form will generally be treated as `PRIVATE`.

Another supported but deprecated form is the following:

```
target_link_libraries(targetName
    LINK_INTERFACE_LIBRARIES item [item...]
)
```

This is a pre-cursor to the `INTERFACE` keyword of the newer form covered above, but its use is discouraged by the CMake documentation. Its behavior can affect different target properties, with the policy settings controlling that behavior. This is a potential source of confusion for developers, which can be avoided by using the newer `INTERFACE` form instead.

```
target_link_libraries(targetName
    <LINK_PRIVATE|LINK_PUBLIC> lib [lib...]
    [<LINK_PRIVATE|LINK_PUBLIC> lib [lib...]]
)
```

Similar to the previous old-style form, this one is a pre-cursor to the `PRIVATE` and `PUBLIC` keyword versions of the newer form. Again, the old-style form has the same confusion over which target properties it affects, and the `PRIVATE/PUBLIC` keyword form should be preferred for new projects.

4.6. Recommended Practices

Target names need not be related to the project name. While they are sometimes the same, the two things are separate concepts. Changing one shouldn't imply that the other must also change. Project and target names should rarely change anyway, since doing so would break any downstream consumer that relied on the existing names. Therefore, set the project name directly rather than via a variable. Choose a target name according to what the target does rather than the project it is part of. Assume the project will eventually need to define more than one target. These practices reinforce better habits, which will be important when working on more complex, multi-target projects.

It is common to see tutorials and examples define a variable for the project name, and then reuse that variable for the name of an executable or library target. Another common variation uses the `PROJECT_NAME` variable, which is set automatically by the `project()` command. Both practices should be avoided.

```
# BAD: Don't use a variable to set the project name, set it directly
set(projectName MyExample)
project(${projectName})

# BAD: Don't set the target name from the project name
add_executable(${projectName} ...)
```

```
# GOOD: Set the project name directly
project(MyProj)

# BAD: Don't set the target name from the project name
add_executable(${PROJECT_NAME} ...)
```

```
# GOOD: Set the project name directly
project(MyProj)

# GOOD: Target name is independent of the project name
add_executable(MyThing ...)
```

When naming targets for libraries, resist the temptation to start or end the name with `lib`. On just about all platforms except Windows, a leading `lib` will be prefixed automatically when constructing the actual library name to make it conform to the platform's usual convention. If the target name already begins with `lib`, the library file names end up with the form `liblibsomething...`, which people often assume to be a mistake.

Unless there are strong reasons to do so, avoid specifying the `STATIC` or `SHARED` keyword for a library until it is known to be needed. This allows greater flexibility in choosing between static or dynamic libraries as an overall project-wide strategy. The `BUILD_SHARED_LIBS` variable can be used to change the default in one place instead of having to modify every call to `add_library()`.

Always specify `PRIVATE`, `PUBLIC` and/or `INTERFACE` keywords when calling the `target_link_libraries()` command, rather than following the old-style CMake syntax which assumed everything was `PUBLIC`. As a project grows in complexity, these three keywords have a stronger impact on how inter-target dependencies are handled. Using them from the beginning of a project also forces developers to think about the dependencies between targets, which can help to highlight structural problems within the project much earlier.

Chapter 5. Basic Testing And Deployment

CMake provides a variety of features for testing a project, installing it, and producing packages. The features associated with each of these activities can be overwhelming, in large part because the activities themselves are complex. The sheer number of different things done by various platforms, testing tools, and packaging systems is often underappreciated. CMake aims to simplify that complexity by presenting a more consistent interface and set of controls, while still providing access to low-level features where needed. As a result, understanding a few basics is generally enough to start exploring each of these topics and gain useful results.

5.1. Testing

CMake provides a separate command-line tool called `ctest`. It can be thought of as a test scheduling and reporting tool, offering close integration with CMake for defining tests in a convenient and flexible way. Typically, CMake will generate the input file necessary for `ctest` based on details provided by the project.

The following minimal example shows how to define a project with a couple of simple test cases:

```
cmake_minimum_required(VERSION 3.19)
project(MyProj VERSION 4.7.2)

enable_testing()

add_executable(testSomething testSomething.cpp)

add_test(NAME SomethingWorks COMMAND testSomething)
add_test(NAME ExternalTool COMMAND /path/to/tool someArg moreArg)
```

The `enable_testing()` call is needed to instruct CMake to produce an input file for `ctest`. It should generally be called just after `project()`.

The `add_test()` command is how a project can define a test case. It supports a couple of different forms, but the one shown above with `NAME` and `COMMAND` keywords is recommended.

The argument following `NAME` should generally contain only letters, numbers, hyphens, and underscores. Other characters may be supported if using CMake 3.19 or later, but projects should avoid anything complicated and stick with these basic characters in most cases.

The `COMMAND` can be any arbitrary command that could be run from a shell or command prompt. As a special case, it can also be the name of an executable target defined by the project. CMake will then translate that target name into the location of the binary built for that target. In the above example, the `SomethingWorks` test will run the executable built for the `testSomething` CMake target. The project doesn't have to care where the build will create the binary in the file system, CMake will provide that information to `ctest` automatically.

By default, a test is deemed to pass if it returns an exit code of 0. Much more detailed and flexible criteria can be defined, which is covered in [Section 27.3, “Pass / Fail Criteria And Other Result Types”](#), but a simple check of the exit code is often sufficient.

The following sequence of steps will configure, build, and test a project. Any CMake generator could be used, but this example uses Ninja. It configures the build to use the Debug configuration, and ctest will detect that automatically.

```
mkdir build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug ../source
cmake --build .
ctest
```

Some generators are multi-configuration, like Xcode, Visual Studio, and Ninja Multi-Config. When using such generators, the configuration to build and test needs to be provided at build and test time:

```
mkdir build
cd build
cmake -G "Ninja Multi-Config" ../source
cmake --build . --config Debug
ctest --build-config Debug
```

For convenience, `-C` can be used instead of the longer `--build-config` with the `ctest` command.

If there are many tests, and they take non-trivial time to run, they can be executed in parallel:

```
ctest --parallel 16
```

The shorter `-j` option can also be used instead of `--parallel`. The number after the keyword specifies how many tests can run at the same time. In more advanced scenarios, a test can be allocated more than one CPU, which is discussed in [Section 28.3, “Parallel Execution”](#).

The default output from `ctest` is fairly concise. Output from passing and failing tests will be hidden, with only the results being shown. Full output can be obtained with the `-V` or `--verbose` option, or just the output of failing tests with `--output-on-failure`.

As a convenience, CMake also defines a test build target, which runs `ctest` with a default set of options. It always runs all tests, and it provides only very limited control over the test output and the way tests are run. It can be useful as a way to run tests in IDEs which don't provide a dedicated feature for running `ctest` tests, but generally developers are better off learning to run `ctest` directly and take advantage of all its various options.

CMake and `ctest` offer much more functionality than discussed above. [Part IV, “Testing And Analysis”](#) includes a number of chapters that walk through many of the powerful and flexible features available.

5.2. Installing

While things built by a project can often be used directly from the build directory, that's often just a stepping stone on the way to deploying the project. Deployment can take different forms, but a common element to most of them is an install step. During installation, files are copied from the build directory (and possibly the source directory) to the install location. Files may be transformed in some way before or after the copy.

CMake provides direct support for installing different types of artifacts. The `install()` command provides the majority of that functionality, and it has a number of different forms. The following minimal example uses the `install(TARGETS)` form to install a few CMake targets.

```
cmake_minimum_required(VERSION 3.14)
project(MyProj VERSION 4.7.2)

add_executable(MyApp ...)
add_library(AlgoRuntime SHARED ...)
add_library(AlgoSDK STATIC ...)

# This concise form requires CMake 3.14 or later
install(TARGETS MyApp AlgoRuntime AlgoSDK)
```

The above example takes advantage of features added in CMake 3.14. When no destinations are given to tell CMake where to install the targets, CMake will use default locations that correspond to the convention used on most Unix systems. The same layout also typically works fine on Windows, so it can generally be used everywhere except for application bundles on Apple platforms, which have their own unique directory structure (discussed in detail in [Section 25.2.1, “Bundle Structure”](#) and [Section 25.3.1, “Framework Structure”](#)). CMake's default layout will install executables to a `bin` subdirectory below the base install location, libraries in a `lib` subdirectory, and headers in an `include` subdirectory. On Windows, DLL libraries would be installed to `bin` rather than `lib`.

If using CMake 3.13 or earlier, no default destinations are provided, and the project must specify them explicitly. The equivalent `install()` command to the previous example would then look like this:

```
install(TARGETS MyApp AlgoRuntime AlgoSDK
  RUNTIME DESTINATION bin
  LIBRARY DESTINATION lib
  ARCHIVE DESTINATION lib
)
```

[Section 35.2, “Installing Project Targets”](#) goes into detail about what `RUNTIME`, `LIBRARY`, and `ARCHIVE` mean, along with a range of other types of installable entities associated with a target. It also discusses the handling of symbolic links created for shared libraries when they have version details associated with them, which is a separate topic covered in [Section 23.3, “Shared Library Versioning”](#).

Files and directories can also be installed. The following demonstrates how header files have traditionally been installed until more recent CMake versions:

```
install(FILES things.h algo.h DESTINATION include/myproj)
install(DIRECTORY headers/myproj DESTINATION include)
```

The `install(FILES)` form requires each file to be listed individually. This is useful when only some files within a directory should be installed. The `install(DIRECTORY)` form recursively copies the specified directory to the destination. To copy the *contents* of the directory rather than the directory itself, append a trailing `/` to the directory name. For example, if the only thing in the `headers` directory was a `myproj` subdirectory, the following command would be equivalent to the one above:

```
install(DIRECTORY headers/ DESTINATION include/myproj)
```

With CMake 3.23 or later, a more powerful, more convenient way of handling headers is to use *file sets*. A file set can associate headers with a target, and the headers can be installed along with the target in an `install(TARGETS)` call. No separate `install(FILES)` or `install(DIRECTORY)` call is needed. In addition, a file set provides information about whether a header is private or public, and also about the header search paths a consumer of the target must use.

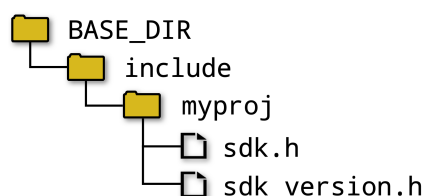
File sets are defined by the `target_sources()` command. [Section 16.2.7, “File Sets”](#), [Section 32.6, “File Sets Header Verification”](#), and [Section 35.5.1, “File Sets”](#) discuss the topic in detail, but for now, the following example shows how a public headers file set might be defined and installed:

```
add_library(AlgoSDK ...)

target_sources(AlgoSDK
    PUBLIC
        FILE_SET api
        TYPE HEADERS
        BASE_DIRS headers
        FILES
            headers/myproj/sdk.h
            headers/myproj/sdk_version.h
)

install(TARGETS AlgoSDK FILE_SET api)
```

As mentioned earlier, CMake 3.14 and later will use `include` as a default destination for headers when no destination is provided. And when file sets are installed as part of a target, the relative structure below the file set’s `BASE_DIRS` will be preserved. Thus, the directory structure of the installed headers resulting from the above example would be:



When installing libraries and headers for other projects to build against, it is recommended to provide a set of CMake-specific config package files as well. These files give the consuming project a CMake target they can link against, and that target will include header search path details to be applied to the consumer. The `install(EXPORT)` sub-command is typically used to produce some of these config package files. This more complex topic is discussed in [Section 35.3, “Installing Exports”](#) and [Section 35.9, “Writing A Config Package File”](#). The consumer imports the package with a command called `find_package()`, which is covered in detail in [Section 34.4, “Finding Packages”](#).

The following sequence of commands shows how to configure, build and install a project:

```
mkdir build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug ../source
cmake --build .
cmake --install . --prefix /path/to/somewhere
```

These steps are very similar to those given earlier for the example in [Section 5.1, “Testing”](#), except the last command is different. The `cmake --install <buildDir>` form is available with CMake 3.15 or later. The `--prefix <installDir>` option specifies the base install location to install the project to. If `--prefix` is not given, the install location is taken from a platform-dependent value set during the configure step. See the discussion of `CMAKE_INSTALL_PREFIX` in [Section 35.1.2, “Base Install Location”](#) for further details on that aspect.

As was the case for testing, multi-configuration generators are also supported for installs too. The configuration must be specified as part of the build and install steps:

```
mkdir build
cd build
cmake -G "Ninja Multi-Config" ../source
cmake --build . --config Debug
cmake --install . --config Debug --prefix /path/to/somewhere
```

CMake also provides an `install` build target, which can be used to install the project with default options. It isn’t as flexible as running `cmake --install`, but it is supported for all CMake releases, not just CMake 3.15 or later. The `cmake --install` command accepts a few other options not shown above (see [Section 35.10, “Executing An Install”](#)), whereas the `install` build target has limited customizability.

Once a project grows beyond a single application or library, installing everything in a single package may no longer be appropriate. The project may want to split up the installation into separate *components*. CMake has extensive support for this, but it is not a simple topic. Splitting up a project into multiple components affects not just what is installed, but also raises questions like "What dependencies exist between the components?", and "How should components map to separate packages, or installable units within a packaged product?". Components are discussed throughout [Chapter 35, *Installing*](#), and [Section 36.2, “Components”](#) focuses specifically on the packaging aspects.

5.3. Packaging

Installing a project built from sources was once a widespread way for users to install software. However, many projects can't provide source files for the user to build themselves. These days, users tend to expect pre-built packages anyway.

CMake provides the `cpack` tool, which can produce binary packages in a variety of formats. These include simple archives like `.zip`, `.tar.gz` and `.7z`, packages for platform-specific packaging systems like RPM, DEB, and MSI, and even standalone graphical installers. These are all based on installing a project using the information provided through `install()` commands, and others. Internally, `cpack` effectively does one or more `cmake --install` commands with `--prefix` set to a temporary staging area. The contents of that staging area are then used to create a package in the relevant format.

Basic packaging is implemented by setting some relevant CMake variables, then including a CMake module called `CPack`, which writes out an input file for the `cpack` tool. CMake modules are introduced in [Chapter 12, Modules](#), CMake variables in [Chapter 6, Variables](#), and [Chapter 36, Packaging](#) covers packaging in detail. For now, the following minimal example shows how to put these things together in a fairly simple way:

```
cmake_minimum_required(VERSION 3.14)
project(MyProj VERSION 4.7.2)

add_executable(MyApp ...)
add_library(AlgoRuntime SHARED ...)
add_library(AlgoSDK STATIC ...)

install(TARGETS MyApp AlgoRuntime AlgoSDK)

# These are project-specific
set(CPACK_PACKAGE_NAME MyProj)
set(CPACK_PACKAGE_VENDOR MyCompany)
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY "An example project")

# These lines tend to be the same for every project
set(CPACK_PACKAGE_INSTALL_DIRECTORY ${CPACK_PACKAGE_NAME})
set(CPACK_VERBATIM_VARIABLES TRUE)

# This is what writes out the input file for cpack
include(CPack)
```

The various `set(...)` lines in the above example demonstrate setting CMake variables. The values only need to be surrounded in quotes if they contain spaces. A real project would set more variables than these, but the above is enough to get started producing a working package. [Section 36.1, “Packaging Basics”](#) provides more complete guidance on a minimum set of variables a project should set for production-grade packages.

Note the `VERSION` keyword in the `project()` call. It is a convenient way of providing a default value for the package version, if using CMake 3.12 or later. As noted back in [Section 3.2, “The project\(\) Command”](#), it can also be used in other ways by the project, such as embedding the version number in source files compiled for the project (covered in [Chapter 22, Specifying Version Details](#)).

The following familiar set of steps demonstrates how to configure, build, and package a project:

```
mkdir build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Release ../source
cmake --build .
cpack -G "ZIP;WIX"
```

Once again, only the last line is significantly different to earlier examples (the above also configures for Release rather than Debug, since that's more typical for pre-built packages). The `cpack -G` option specifies the package formats to generate. When more than one format is given, they must be separated by a semicolon. The list of supported formats varies by platform and can be obtained by running `cpack --help`.

As expected, multi-configuration generators are supported, and the configuration must be specified when building and when packaging:

```
mkdir build
cd build
cmake -G "Ninja Multi-Config" ../source
cmake --build . --config Release
cpack -G "ZIP;WIX" --config Release
```

Continuing the familiar pattern, CMake provides a package build target, which runs `cpack` with the options specified by the project. Again, customizability is very limited when building the package target instead of running `cpack` directly. When using the package build target, a default set of package generators will be used, but that default set is unlikely to be appropriate. The project can override the default set by setting the `CPACK_GENERATORS` variable before calling `include(CPack)`. The list of generators will typically be different for each major platform, so some conditional logic is likely to be needed (see [Chapter 7, Flow Control](#)). The following example taken from [Section 36.1, “Packaging Basics”](#) provides a good starting point.

```
if(WIN32)
    set(CPACK_GENERATOR ZIP WIX)
elseif(APPLE)
    set(CPACK_GENERATOR TGZ productbuild)
elseif(CMAKE_SYSTEM_NAME STREQUAL "Linux")
    set(CPACK_GENERATOR TGZ RPM)
else()
    set(CPACK_GENERATOR TGZ)
endif()
```

The `cpack -G` option overrides any list set by the project with the `CPACK_GENERATOR` variable, so the command line still has full control.

5.4. Recommended Practices

Get familiar with the `ctest` and `cpack` command line tools. These support various options which are not available when building the corresponding test and package targets. The `ctest` tool in particular has many useful options that are instrumental in day-to-day development when running tests.

When developing the project and modifying its `install()` commands, doing test installs to a temporary staging area is a useful technique. Use commands like `cmake --install` with the `--prefix` option pointing at the staging area to confirm the installed contents match the expected set of files. The temporary staging area should first be wiped to ensure no contents from any previous test install are left behind.

Avoid doing direct installs from the build tree to permanent or system-wide locations. Consider producing a binary package and installing that instead as two separate steps. Installing directly from a build directory may require administrative privileges, and some aspects of doing an install may modify the contents of the build directory and change file or directory ownership. This can cause hard-to-trace build errors later when the build runs under the non-administrative account, but is unable to modify things it normally would be able to.

If the project can set its minimum CMake version to 3.23 or higher, invest some time learning about CMake file sets (see [Section 16.2.7, “File Sets”](#) and [Section 35.5.1, “File Sets”](#)). Put all the project’s header files in file sets, and use separate `PRIVATE` and `PUBLIC` file sets to clearly define which ones are meant to be installed, and which ones are not. This will also simplify the handling of header search paths, both when building the project, and when it is installed.